The TPTP Typed First-order Form with Arithmetic

Geoff Sutcliffe[1], Stephan Schulz[2], Koen Claessen[3], and Peter Baumgartner[4]

[1] University of Miami, USA
[2] Technische Universität München, Germany
[3] Chalmers University, Sweden
[4] NICTA and ANU, Australia

Abstract. The TPTP World is a well established infrastructure supporting research, development, and deployment of Automated Theorem Proving systems. Recently, the TPTP World has been extended to include a typed first-order logic, which in turn has enabled the integration of arithmetic. This paper describes these developments.

1 Motivation and History

The TPTP World [32] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. The TPTP World is based on the Thousands of Problems for Theorem Provers (TPTP) problem library [30], and includes the TPTP language, the SZS ontologies, the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries, and the CADE ATP System Competition (CASC). This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems – most state of the art systems natively read the TPTP language, many produce proofs or models in the TSTP format, much testing and development is done using the TPTP problem library, and CASC is an annual focal point where developers meet to discuss new ideas and advances in ATP techniques.

Originally the TPTP supported only first-order problems in clause normal form (CNF). Over the years support for the full first-order form (FOF) and typed higher-order form (THF) have been added. Recently the simply typed first-order form (TFF) has been added. TFF has in turn been used as the basis for supporting arithmetic. Problems that use these new features have been added to the TPTP problem library, and ATP systems that can solve these problems have been developed. This paper describes the key steps of these developments.

While the development of the TPTP World for typed first-order logic is new, several similar logics have been described previously, e.g., [36, 26, 13]. However, there are no contemporary ATP systems that implement those logics. There is active related research in the SMT community, which started in 2003 [24]. There are high performance systems for the various logics of the SMT-LIB[1], e.g., those

[1] See the SMT-LIB web page `http://combination.cs.uiowa.edu/smtlib/`

that performed well in the SMT-COMP competitions.[2] While the TFF language was designed independently, there are inevitable parallels between the TFF and SMT languages. The TPTP and SMT languages both fully support a typed first-order logic, and both have specific features for arithmetic theories. Some features of the TPTP TFF language were adopted from SMT, and some differences were motivated by differences between the two communities' idioms (e.g., the TPTP arithmetic includes the Euclidean quotient used in the SMT-LIB `Ints` theory, but also other quotients requested by TPTP users). Salient commonalities and differences between the two languages are evident in this paper. At the level of the TPTP and SMT-LIB problem collections, the problems in SMT-LIB are categorized with respect to their underlying logics and theories (e.g., the admissible quantifier prefixes and the kind of arithmetic used). Those categories and the problems in them typically reflect the capabilities of the available SMT solvers. The TPTP library uses its "specialist problem class" categorization (e.g., the use of equality and the SZS status of problems [29]) only for the analysis of results, and in this way encourages the submission of problems and the development of tools without a specific reasoner or language fragment in mind. There is a growing linkage between the SMT and TPTP worlds, stimulated and made possible by the developments described in this paper. One example was the entry of the SMT-based systems CVC3 and Z3 in the TFA division of CASC-23 [34]. Tools for translating between the TPTP and SMT formats have (or should be by the time this paper is published) been developed – the TPTP2X utility distributed as part of the TPTP is used to translate TPTP TFF problems to SMT2 format for input by CVC3 (see Section 4).

2 The TPTP Typed First-order Form Language

The design of the TPTP's TFF language was based on consideration of various features of type systems. These fall into four broad categories – the *sorts* (atomic types) that are available, the possibilities for *the types of terms*, the *syntax* of expressions, and the *semantics* of the logic. The decisions made for the first two categories, and the effects of the decisions on ATP systems, are discussed in Section 2.1. The syntax is described in Section 2.2, and semantic issues are discussed in Section 2.3. There were many possibilities for each issue, and the decisions aimed to impose a low initial entry barrier for ATP system developers and users, and to allow for future additions to the language. The initial language is thus known as "TFF0" (much like THF0 [8]).

2.1 Decisions about Types and Terms

The decisions made for TFF0 provide a useful and simple extension of the existing untyped FOF logic. The decisions are:

– TFF0 is a simple many-sorted logic. Sorts are interpreted by non-empty, pairwise disjoint, domains.

[2] `http://www.smtcomp.org/`

– All uninterpreted functions and predicates are monomorphic. Although ad-hoc polymorphism over sorts is conceptually simple, allowing it would require extending the TPTP syntax to provide sort annotations on symbols (as supported in the SMT language version 2).
– Equality is ad-hoc polymorphic over the sorts. An equation between terms that have different sorts is ill-typed.
– Subtyping is not employed. (Subtyping may be added in the future.)

This simple many-sorted type system is an extension of untyped first-order logic. As all symbols are monomorphic, all terms except for variables are automatically typed. For problems without equality, satisfiability of well-typed formulae is not affected by ignoring types. For problems with equality, variables need to carry explicit type information because context does not provide the type information, e.g., the variables in an equation $x = y$.

Many proof calculi generalize directly to the typed case, so that existing techniques and implementations can be carried over without prohibitive effort. In particular, unique most general unifiers and matches still exist, and can be computed by straightforward generalizations of existing algorithms. Standard inferences and simplifications remain correct as long as variable instantiations are type-preserving. Systems based on direct instantiation need to check every variable instantiation. Systems based on unification need to check variable instantiations in equational inferences, e.g., paramodulation and superposition, when paramodulating from (or into, but that is not required in most calculi) a variable term. These checks reject inferences that are allowed when using sort predicates (see Section 2.3), and proof search can be simpler. For finite model finders type information is valuable because it restricts the space of possible models that need to be explored. The domains of some of the sorts can be smaller than the domains of other sorts, leading to possibly more efficient algorithms. While some systems, e.g., Paradox [11], try to derive type information to exploit this, user specified type information can be more precise.

For ATP users, typing leads to much simpler encodings than using type predicates, and the requirement of well-typedness helps to correctly encode problems. A typed language is also necessary for correct encoding of problems with arithmetic.

2.2 Syntax

The TFF0 syntax implements the decisions described in Section 2.1. A new TPTP language variant has been introduced, using `tff` as the language symbol of annotated formulae.[3]

– The sorts `$i` (individuals) and `$o` (booleans) are defined. (Note, as is explained below, `$o` is used only as the result sort in predicate type declarations, i.e., there is no built-in theory of boolean terms.) Other defined sorts are associated with specific theories. In particular, `$int`, `$rat`, and `$real` are defined for interpreted arithmetic – see Section 3.

[3] The BNF is available at `http://www.tptp.org/TPTP/SyntaxBNF.html`

– `$tType` is used to introduce users' sorts, by declaring them to be of the psuedo-sort `$tType`. For example

```
tff(fruit_type,type,                tff(list_type,type,
    fruit: $tType ).                    list: $tType ).
```

This is the only use of `$tType`. Declaration of users' sorts is not required, i.e., sorts can be introduced on the fly. TFF problems in the TPTP problem library have all sorts declared, so as to provide a typo check (pun intended).

– Every function and predicate symbol has at most one declared *type* that specifies the argument and result sorts. For example

```
tff(cons_type,type,                 tff(is_empty_type,type,
    cons: (fruit * list) > list ).      isEmpty: list > $o ).
```

The argument sorts cannot be `$o`. The result sort of a function cannot be `$o`, and the result sort of a predicate must be `$o`. Note that symbols of arity greater than one use the `*` for a cross-product type – currying is not possible. If a symbol's type is declared more than once, and the types are not the same, that is an error. Multiple identical type declarations for a symbol are allowed (to support, e.g., the merging of specifications from multiple different input files).

– Defined functions and predicates have preassigned types.
  • `$true` is of type `$o`
  • `$false` is of type `$o`
  • `=` is ad-hoc polymorphic over the sorts except `$o`. The two arguments must be of the same sort, and the result sort is `$o`. The equality symbol thus represents distinct predicate symbols for each sort.
  • The types of numbers and the arithmetic functions and predicates are defined in Section 3.

– Every variable can be given a sort at quantification time. For example

```
tff(list_not_empty,axiom,
    ! [X: fruit,Xs: list] : ~isEmpty(cons(X,Xs)) ).
```

– If a symbol is used and its type has not been declared, then default types are assumed:
  • All untyped predicates get the type `($i * ... * $i) > $o`.
  • All untyped functions get the type `($i * ... * $i) > $i`.
  • All untyped variables are of the sort `$i`.

  If a symbol's type is declared later to be different from an assumed type, that is an error.

TPTP file names for TFF problems use a _ separator (in the way that `^` is used for THF, `+` is used for FOF, and `-` is used for CNF). Use of the TFF0 language is demonstrated in the following example. The formulae are given in Figure 1.

Every student is enrolled in at least one course. Every professor teaches at least one course. Every course has at least one student enrolled. Every course has at least one professor teaching. The coordinator of a course teaches the course. If a student is enrolled in a course then the student is taught by every professor who teaches the course. Michael is enrolled in CSC410. Victor is the coordinator of CSC410. Therefore, Michael is taught by Victor.

```
%------------------------------------------------------------------------
tff(student_type,type,   student:    $tType ).
tff(professor_type,type, professor:  $tType ).
tff(course_type,type,    course:     $tType ).
tff(michael_type,type,   michael:    student ).
tff(victor_type,type,    victor:     professor ).
tff(csc410_type,type,    csc410:     course ).
tff(enrolled_type,type,  enrolled: ( student * course ) > $o ).
tff(teaches_type,type,   teaches:  ( professor * course ) > $o ).
tff(taught_by_type,type, taughtby: ( student * professor ) > $o ).
tff(coordinator_of_type,type, coordinatorof: course > professor ).

tff(student_enrolled_axiom,axiom,
    ! [X: student] : ? [Y: course] : enrolled(X,Y) ).
tff(professor_teaches,axiom,
    ! [X: professor] : ? [Y: course] : teaches(X,Y) ).
tff(course_enrolled,axiom,
    ! [X: course] : ? [Y: student] : enrolled(Y,X) ).
tff(course_teaches,axiom,
    ! [X: course] : ? [Y: professor] : teaches(Y,X) ).
tff(coordinator_teaches,axiom,
    ! [X: course] : teaches(coordinatorof(X),X) ).
tff(student_enrolled_taught,axiom,
    ! [X: student,Y: course] :
      ( enrolled(X,Y)
     => ! [Z: professor] : ( teaches(Z,Y) => taughtby(X,Z) ) ) ).
tff(michael_enrolled_csc410_axiom,axiom,
    enrolled(michael,csc410) ).
tff(victor_coordinator_csc410_axiom,axiom,
    coordinatorof(csc410) = victor ).

tff(teaching_conjecture,conjecture,
    taughtby(michael,victor) ).
%------------------------------------------------------------------------
```

Fig. 1. Example TFF problem

2.3 Type Checking and Semantics

A formula is well-typed iff all the atoms in the formula are well-typed. A non-equality atom is well-typed iff all the terms in the atom are well-typed, and the sorts of the arguments of the atom conform to the predicate symbol's type. An equality atom is well-typed iff both the terms of the equation are well-typed and are of the same sort. A term is well-typed iff all the subterms in the term are well-typed, and the sorts of the arguments of the term conform to the function symbol's type.

The semantics of TFF0 (without arithmetic) is a standard and straightforward generalization of the standard semantics of untyped first-order logic. A semantics consistent with the one below has been given, e.g., in [15].

Assume a TFF0 language with sorts $s_1, \ldots, s_n$ and variables $V = V_{s_1} \uplus \ldots \uplus V_{s_n}$, where variables from V_{s_i} have the sort s_i. Further, assume a formula (or set of formulae) build over V, function symbols from F and predicate symbols from P. An *interpretation* I consists of a *domain* $D = D_{s_1} \uplus \ldots \uplus D_{s_n}$ with disjoint, non-empty sub-domains for each sort, and a sort and arity-respecting mapping of function symbols to functions and predicate symbols to relations (representing the tuples of which the predicate holds true). In other words, if the function symbol $\mathtt{f} \in F$ is declared as $f : (s_1 * \ldots * s_n) > s$, then its interpretation is a function $I(\mathtt{f}) : D_{s_1} \times \ldots \times D_{s_n} \to D_s$. If the predicate symbol $\mathtt{p} \in P$ is declared as $p : (s_1 * \ldots * s_n) > \$\mathtt{o}$, then its interpretation is a relation $I(\mathtt{p}) \subseteq (D_{s_1} \times \ldots \times D_{s_n})$. A *typed valuation* is a function $\phi : V \to D$ with the property that $\phi(V_s) \subseteq D_s$ for all sorts s. $\phi_{x \leftarrow d}$ denotes a valuation that is equal to ϕ for all variables but X, and maps X to d.

The value of a term under an interpretation I and valuation ϕ is: $eval_{I,\phi}(X) = \phi(X)$ for $X \in V$, and $eval_{I,\phi}(\mathtt{f}(t_1, \ldots, t_n) = I(\mathtt{f})(eval_{I,\phi}(t_1), \ldots, eval_{I,\phi}(t_n))$ for $\mathtt{f} \in F$. Let $\{T, F\}$ be the truth values. For atoms $eval_{I,\phi}(\mathtt{p}(t_1, \ldots, t_n)) = T$ iff $(eval_{I,\phi}(t_1), \ldots, eval_{I,\phi}(t_n)) \in I(\mathtt{p})$. Formulae with connectives are interpreted as usual. Quantifiers, on the other hand, respect the type of the bound variable: $eval_{I,\phi}(\forall X : s \ . \ G) = T$ iff $eval_{I,\phi_{X \leftarrow d}}(G) = T$ for all $d \in D_t$ and $eval_{I,\phi}(\exists X : s \ . \ G) = T$ iff $eval_{I,\phi_{X \leftarrow d}}(G) = T$ for at least one $d \in D_t$. Note that for closed formulae the valuation of the variables is determined by the quantifiers, and the value of a closed formula depends only on the interpretation.

Recall that the equality symbol represents distinct predicate symbols for each sort, each written here as as $\mathtt{=}_s$ for a sort s. An interpretation I is an *E-interpretation*, if $I(\mathtt{=}_{s_i})$ is the equality relation on D_{s_i}, i.e., $I(\mathtt{=}_{s_i}) = \{(d, d) \mid d \in D_{s_i}\}$, for $i = 1, \ldots, n$.[4] An E-interpretation I is a *TFF model* of a formula F if $eval_I(F) = T$. As usual, a formula is *TFF satisfiable* if it has at least one TFF model, *TFF unsatisfiable* otherwise. A formula is a *TFF tautology* if every E-interpretation is a TFF model.

The semantics is alternatively given by the following standard (e.g., [36]) translation into untyped first-order logic with equality. A well-typed formula F has a typed model iff its translated untyped counterpart F' has an (untyped) model. Each sort becomes a new unary predicate in the untyped world. Then

– A TFF sort declaration $a\_sort : \$\mathtt{tType}$ produces a FOF axiom $\exists X \ . \ a\_sort(X)$. This ensures that sorts are inhabited.

[4] In refutational theorem proving it is customary to work with Herbrand interpretations and congruence relations on them to provide semantics for the equality symbol. This approach can still be used in the context of (in particular) clause logic and theory reasoning, see e.g., [19]. However, it cannot be used when arbitrary quantification is allowed, as Herbrand's theorem no longer holds, even without theories.

- Pairs of TFF sort declarations *one_sort* : `$tType` and *two_sort* : `$tType` produce a FOF axiom $\forall X, Y \ . \ \mathit{one\_sort}(X) \wedge \mathit{two\_sort}(Y) \Rightarrow X \neq Y$. This ensures that sorts are pairwise disjoint. These axioms are not logically necessary, because a model of the FOF formulae without these axioms can be used to construct a model of the TFF formulae [12], i.e., a formula has a model with disjoint domains iff it has a model with one domain. However, for model generation these axioms are useful because they force terms with different types to be interpreted as different domain elements, i.e., the domain of the FOF model can be divided into subdomains for the different sorts.
- A TFF function type declaration $f : (s_1 {*} \ldots {*} s_n) > s_f$ produces a FOF axiom $\forall X_1, \ldots, X_n \ . \ s_f(X_1, \ldots, X_n)$. It is unnecessary to have an implication with the antecedent checking the sorts of the arguments `X1,...,Xn`, because it is impossible to use incorrectly sorted arguments in a well-typed formula.
- Predicate type declarations are ignored.
- A TFF universally quantified formula $\forall X_1 : s_1, \ldots, X_n : s_n \ . \ p(X_1, \ldots, X_n)$ produces a FOF formula $\forall X_1, \ldots, X_n \ . \ s_1(X_1) \wedge \ldots \wedge s_n(X_n) \Rightarrow p(X_1, \ldots, X_n)$.
- A TFF existentially quantified formula $\exists X_1 : s_1, \ldots, X_n : s_n \ . \ p(X_1, \ldots, X_n)$ produces a FOF formula $\exists X_1, \ldots, X_n \ . \ s_1(X_1) \wedge \ldots \wedge s_n(X_n) \wedge p(X_1, \ldots, X_n)$.

3 TPTP Arithmetic

The TFF0 language has features that facilitate the addition of interpreted functions and predicates for integer, rational, and real arithmetic. Arithmetic requires a separate name space for numeric constants (i.e., numbers) and operators. Separate structures are assumed for integer, rational, and real arithmetic, each comprised of denumerably many numeric constants, and certain defined function and predicate symbols.

3.1 Syntax

The TPTP syntax for numeric constants[5] and the defined function and predicate symbols are given in Table 1. Each function and predicate symbol is adhoc polymorphic over the numeric sorts (with one exception – `$quotient` is not defined for `$int`). All arguments must have the same numeric sort. All the functions, except for the coercion functions `$to_int` and `$to_rat`, have the same result sort as their arguments. For example, `$sum` can be used with the types `($int * $int) > $int`, `($rat * $rat) > $rat`, and `($real * $real) > $real`. The coercion functions `$to_???` always have a `$???` result. All the predicates have a `$o` result. For example, `$less` can be used with the types `($int * $int) > $o`, `($rat * $rat) > $o`, and `($real * $real) > $o`.

TPTP file names for TFF problems with arithmetic use a `=` separator. Use of the TFF0 language with integer arithmetic is demonstrated in the following example. The formulae are given in Figure 2.

[5] See `http://www.tptp.org/TPTP/SyntaxBNF.html` for the precise syntax in BNF.

Symbol	Usage, comments, examples
`$int`	The type of integers. Examples: 123, -123
`$rat`	The type of rationals. Examples: 123/456, -123/456, +123/456 The denominator must be unsigned and positive.
`$real`	The type of reals. Examples: 123.456, -123.456, 123.456E789.
`=` (infix)	See Section 2.2
`$less`/2	Less-than comparison of two numbers.
`$lesseq`/2	Less-than-or-equal-to comparison of two numbers.
`$greater`/2	Greater-than comparison of two numbers.
`$greatereq`/2	Greater-than-or-equal-to comparison of two numbers.
`$uminus`/1	Unary minus of a number.
`$sum`/2	Sum of two numbers.
`$difference`/2	Difference between two numbers.
`$product`/2	Product of two numbers.
`$quotient`/2	Exact quotient of two `$rat` or `$real` numbers. For zero divisors the result is not specified.
`$quotient_?`/2	Integral quotient of two numbers, `?` is one of `e`, `t`, or `f`. `$quotient_e` is the Euclidean quotient. `$quotient_t` and `$quotient_f` are respectively the truncation and floor of the real division of the arguments. For zero divisors the result is not specified.
`$remainder_?`/2	Remainder after integral division of two numbers using `$quotient_?`. For zero divisors the result is not specified.
`$floor`/1	Floor of a number.
`$ceiling`/1	Ceiling of a number.
`$truncate`/1	Truncation of a number.
`$round`/1	Rounding of a number.
`$is_int`/1	Test for coincidence with an integer.
`$is_rat`/1	Test for coincidence with a rational.
`$to_int`/1	Coercion of a number to `$int`, using `$floor`.
`$to_rat`/1	Coercion of a number to `$rat`. For reals that are not (known to be) rational the result is not specified.
`$to_real`/1	Coercion of a number to `$real`.

Table 1. The TPTP arithmetic syntax

Lists of integers are constructed from a head element and a tail list, with the empty tail being represented by `nil`. A list is *Fibonacci sorted* if it is sorted, and every element is greater or equal to the sum of its two predecessors (from the third element onwards). Therefore the list $[1, 2, 4]$ is Fibonacci sorted.

The TFF arithmetic language aims to provide a comprehensive basis for automated reasoning with arithmetic. There are some minor differences between the TFF arithmetic and SMT-LIB's `Ints`, `Reals`, and `Reals_Ints` theories, e.g., rationals are not explicitly available in SMT, negative numbers are available in TFF, and the available defined predicates and functions are different. Some of the decisions regarding the TFF defined predicates and functions warrant

```
%------------------------------------------------------------------
tff(list_type,type,   list: $tType ).
tff(nil_type,type,    nil: list ).
tff(mycons_type,type, mycons: ( $int * list ) > list ).
tff(sorted_type,type, fib_sorted: list > $o ).

tff(empty_fib_sorted,axiom,
    fib_sorted(nil) ).
tff(single_is_fib_sorted,axiom,
    ! [X: $int] : fib_sorted(mycons(X,nil)) ).
tff(double_is_fib_sorted_if_ordered,axiom,
    ! [X: $int,Y: $int] :
      ( $less(X,Y)
     => fib_sorted(mycons(X,mycons(Y,nil))) ) ).
tff(recursive_fib_sort,axiom,
    ! [X: $int,Y: $int,Z: $int,R: list] :
      ( ( $less(X,Y)
        & $greatereq(Z,$sum(X,Y))
        & fib_sorted(mycons(Y,mycons(Z,R))) )
     => fib_sorted(mycons(X,mycons(Y,mycons(Z,R)))) ) ).

tff(check_list,conjecture,
    fib_sorted(mycons(1,mycons(2,mycons(4,nil)))) ).
%------------------------------------------------------------------
```

Fig. 2. Example TFF problem with arithmetic

justification: The decision to support the three integral quotients (and hence the corresponding remainder functions) came from John Harrison's observations [17] that most programming languages and hardware uses the "t" definition, most interactive theorem provers use the "f" definition, Boute's [9] arguments for the "e" definition are quite sound, and the "e" definition fits better with the generalization to other Euclidean rings. The decision to separate the floor, ceiling, and truncation functions from the `$to_???` type coercion functions allows the production of integral numbers from non-integral numbers without changing their type. The type coercion functions can be used separately to change the type of a number. The decision to overload the type coercion functions for all three numeric types provides the flexibility to change the types of variables in formulae without having to change the formula structure, e.g., `! [X:$real] : p($to_int(X))` can be changed to `! [X:$int] : p($to_int(X))`. Feedback on the TFF arithmetic language is welcome.

3.2 Semantics

The semantics of TFF formulae with arithmetic is defined as a refinement of the semantics of TFF0 formulae in Section 2.3. An E-interpretation I *extends arithmetic* iff (i) the domains of the numeric sorts `$int`, `$rat` and `$real` are $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$, respectively, and, (ii), the numeric constants and operators are interpreted as described in Table 1. Note that in the case of `$quotient`, `$quotient_?` and `$remainder_?` the result is not specified for zero divisors. An interpretation may assign any value to a quotient term whose divisor evaluates to zero. This way, for instance, `$quotient(5,0) = 4` is true in some interpretations and false in others. With these provisions, the semantics of TFF in Section 2.3 carries over to TFA in the expected way. A *TFA model* of a formula F is a TFF model of F that extends arithmetics. A formula is *TFA satisfiable* if it has at least one TFA model, *TFA unsatisfiable* otherwise. A formula is a *TFA tautology* if every E-interpretation that extends arithmetic is a TFA model.

The above definitions are intended to encompass existing theorem proving approaches, such as [3, 19, 6, 25, 2], in the sense that the TFA tautologies are the same on the common logical languages and theories. For example, the approaches in [19, 6, 25, 2] all assume a single arithmetic background theory and linear arithmetic expressions. Restricting TFA correspondingly then is intended to provide a reference semantics for these fragments.

The translation from typed to untyped logic (Section 2.3) can still be used in presence of arithmetic, to "translate away" uninterpreted sorts. Variables of a numeric sort lead to new sort predicates that recognize numeric constants. Additionally, for the overloaded arithmetic functions, e.g., `$sum`, `$difference`, etc., the translations need to have an implication with the antecedent checking the sorts of the arguments. ATP systems must build in these numeric sort predicates in order to completely and correctly process translated problems with arithmetic.

3.3 Solvability and Decidability

The extent to which ATP systems are able to work with the arithmetic predicates and functions is expected to vary, from a simple ability to do arithmetic by evaluating ground numerical terms, e.g., `$sum(2,3)` might be evaluated to `5`, through an ability to instantiate variables in equations involving such functions, e.g., `? [X:$int] : $product(2,$uminus(X)) = $uminus($sum(X,2))` might instantiate `X` to `2`, to extensive algebraic manipulation capability and ability to prove general arithmetic statements, e.g., `! [X: $int] : ? [Y: $int] : $greater(Y,X)`.

The TFA language is rich enough to accommodate virtually any interesting formula class, and asking whether a formula is TFA valid just requires stating that formula as a `conjecture`. Unfortunately, decision procedures or even semi-decision procedures for that validity problem can exist for only rather restricted fragments of TFA. For example, it is well-known that linear arithmetic (over all three numeric domains) is decidable.[6] However, as soon as free predicate symbols are allowed, semi-decidability is lost. Just adding one unary predicate symbol to linear integer arithmetic gives a validity problem that is Π^1_1-hard [16], and hence no complete calculus can exist. Whether function symbols with result sort `$int` are allowed or not does not make a difference, as they can be encoded using predicate symbols (recall that full quantification is available).

[6] See [21] for a recent study of decision methods based on quantifier elimination, for linear integer and for linear real arithmetic.

Most theorem proving calculi are based on clause logic. Without full quantification, it makes a significant difference whether free function symbols with result sort `$int` are allowed or not.[7] Without free function symbols, but with free predicate symbols, (refutationally) complete calculi still exist (e.g., [3, 6, 25]). Allowing free function symbols with result sort `$int` leads again to a Π^1_1-hard unsatisfiability problem, even for formulas without `$int`-sorted variables [19]. This applies to all three numeric domains, as the integers can be encoded in the rationals (and the real numbers) [19, 18]. However, completeness can be achieved under certain assumptions – see [3, 19, 6] for (different) approaches.

4 TFF Problems, ATP Systems, TPTP Software

Prior to the development of the TFF part of the TPTP World, ATP users and developers had long expressed support for extending the TPTP language to include the typed first-order form and arithmetic. However, there had not been a corresponding production of TPTP problems that use typing or arithmetic, or the development of ATP systems that could solve TPTP problems that use typing or arithmetic. This was a chicken-and-egg situation – without such problems in the TPTP problem library there was little infrastructure support for developing the systems, and without the systems there was little motivation for ATP users to produce such problems. It is hoped that the TFF0 developments have broken the cycle: TFF0 problems have been added to the TPTP problem library, systems that can solve TFF0 problems have been developed (with great potential for further work!), and the TPTP World infrastructure has been extended to process TFF0 problems and solutions.

TFF0 problems without and with arithmetic were added to the TPTP in release v5.0.0. The problems came from various sources. Firstly, problems were found in the many papers that describe type systems, e.g., [36, 13]. Not all the problems were suitable, mainly because they employ subtyping, but others were translated to the TFF0 syntax. Secondly, existing TPTP CNF problems were analyzed for implicit type information. The CNF problems were converted in an obvious way to FOF, and then combined with the type information to produce TFF0 problems. Thirdly, TPTP users were asked for such problems, and several replied. Finally, a suite of purely arithmetic conjectures was produced, aimed at testing the basic arithmetic capabilities of ATP systems (these are in the ARI domain of the TPTP problem library). Since then some users have contributed TFF0 problems with and without arithmetic, and TPTP v5.3.0 contains 970 TFF0 problems, of which 846 include arithmetic.

Twelve ATP systems have been written for or adapted to problems written in TFF0, eleven of which have some arithmetic capability. They are CVC3 [5] H2WO4 [33], leanCoP-Ω [31], Otter [20], MELIA [7], MetiTarski [1], SNARK [28], SPASS+T [22], SPASS-XDB [35], ToFoF, Vampire, Z3 [14]. ToFoF is the system that has no arithmetic capability – it is simply the TPTP2X implementation of the translation described in Section 2.3, combined with either the E prover

[7] Free function symbols with the result type `$i` are less problematic.

[27] for theorem proving or Paradox [11] for model finding. For input, H2WO4, leanCoP-Ω, MELIA, SNARK, Vampire, and Z3 read TFF0 natively. For CVC3, TPTP2X is used to translate the formulae to SMT2 syntax [4]. For the other five systems, TPTP2X is used to translate the formulae to FOF. SPASS-XDB and the ToFoF backends read FOF natively. For leanCoP-Ω, Otter, and MetiTarski, TPTP2X is further used to export the formulae in their input syntaxes. Six of the systems rely, to a greater or lesser extent, on external procedures for dealing with the arithmetic aspects of problems. H2WO4 and SPASS-XDB use Mathematica, leanCoP-Ω uses the Omega test system [23], SPASS+T uses the Yices or CVC3 SMT solver, and MetiTarski uses the QEPCAD-B decision procedure for the theory of real closed fields [10]. All the systems are available in the SystemOnTPTP interface.[8] Seven of the systems entered the TFA division of CASC-23, which was won by SPASS+T [34].

The TPTP World infrastructure includes various tools to support ATP users and developers. This infrastructure has been extended to process TFF0 formulae. The Prolog, Java, lex/yacc, and C parsers, which are available as part of the TPTP World, have been updated to support TFF0. These developments make it possible to extend other TPTP World tools, e.g., the GDV derivation verifier and the IDV derivation viewer, to TFF0 data. A utility for checking that all symbols have declared types has been implemented, and a full type checker is being developed. This is ongoing work.

5 Conclusion

This paper has described the TPTP World infrastructure for typed first-order form logic, and its use for expressing arithmetic. The aim of developing the infrastructure is to support research, development, and deployment of ATP for the TFF logic, as a step towards satisfying a long-standing demand from ATP users. Propagation of the TFF language is partially reliant on contributions of TFF problems to the TPTP, and the automated reasoning community is encouraged to make contributions.

Current work includes the addition of conditional terms and formulae, let-binders, and a `$distinct` predicate to implement unique names. Other TPTP users are extending TFF0 with polymorphic types.[9] Future work includes developing a general framework for specifying further theories, e.g., booleans, arrays, bit-vectors, in a machine readable way, along the lines of the SMT-LIB theory specifications.

Acknowledgments: Alexandre Riazanov did the analysis of TPTP CNF problems for implicit type information. Michael Schick and Peter Watson produced many of the TFF and arithmetic problems. Mark Stickel provided a lot of useful feedback on the arithmetic syntax, and the selection of defined arithmetic functions and predicates. Uwe Waldmann provided valuable feedback on precise

[8] `http://www.tptp.org/cgi-bin/SystemOnTPTP`

[9] `https://sites.google.com/site/polymorphictptptff/home`

formulation of parts of the specification. John Harrison helped with insights on computability issues. Andrei Voronkov made some helpful suggestions.

References

1. B. Akbarpour and L. Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.
2. E. Althaus, E. Kruglov, and C. Weidenbach. Superposition Modulo Linear Arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the 7th International Symposium on Frontiers of Combining Systems*, number 5749 in Lecture Notes in Artificial Intelligence, pages 84–99. Springer-Verlag, 2009.
3. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational Theorem Proving for Hierachic First-Order Theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3/4):193–212, 1994.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
5. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification*, number 4590 in Lecture Notes in Computer Science, pages 298–302. Springer-Verlag, 2007.
6. P. Baumgartner, A. Fuchs, and C. Tinelli. ME(LIA) - Model Evolution with Linear Integer Arithmetic Constraints. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 5330 in Lecture Notes in Artificial Intelligence, pages 258–273. Springer-Verlag, 2008.
7. P. Baumgartner, B. Pelzer, and C. Tinelli. Model Evolution with Equality - Revised and Implemented. *Journal of Symbolic Computation*, page To appear, 2011.
8. 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.
9. R. Boute. The Euclidean Definition of the Functions div and mod. *The Euclidean definition of the functions div and mod*, 14(2):127–144, 1992.
10. C.E. Brown. QEPCAD B - A Program for Computing with Semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97–108, 2003.
11. K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
12. A.G. Cohn. Many Sorted Logic = Unsorted Logic + Control? In Bramer M., editor, *Proceedings of Expert Systems '86, The 6th Annual Technical Conference on Research and Development in Expert Systems*, pages 184–194. Cambridge University Press, 1986.
13. A.G. Cohn. A More Expressive Formulation of Many Sorted Logic. *Journal of Automated Reasoning*, 3(2):113–200, 1987.
14. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems, number 4963 in Lecture Notes in Artificial Intelligence, pages 337–340. Springer-Verlag, 2008.

15. J. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Computer Science and Technology Series. Wiley, 1986.
16. J. Halpern. Presburger Arithmetic With Unary Predicates is Π^1_1-Complete. *Journal of Symbolic Logic*, 56(2):637–642, 1991.
17. J. Harrison. Email to Cesare Tinelli.
18. J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.
19. K. Korovin and A. Voronkov. Integrating Linear Arithmetic into Superposition Calculus. In J. Duparc and T. Henzinger, editors, *Proceedings of the 16th EACSL Annual Conference on Computer Science and Logic*, number 4646 in Lecture Notes in Artificial Intelligence, pages 223–237. Springer-Verlag, 2007.
20. W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.
21. T. Nipkow. Linear Quantifier Elimination. *Journal of Automated Reasoning*, 45(2):189–212, 2010.
22. V. Prevosto and U. Waldmann. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, number 192 in CEUR Workshop Proceedings, pages 19–33, 2006.
23. W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 31(8):4–13, 1992.
24. S. Ranise and C. Tinelli. The SMT-LIB Format: An Initial Proposal. In B. Nebel and W. Swartout, editors, *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.
25. P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 5330 in Lecture Notes in Artificial Intelligence, pages 274–289. Springer-Verlag, 2008.
26. M. Schmidt-Schauss. A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation. In Joshi A., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1162–1168, 1985.
27. S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
28. M.E. Stickel. SNARK - SRI's New Automated Reasoning Kit. http://www.ai.sri.com/ stickel/snark.html.
29. G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.
30. 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.
31. G. Sutcliffe. Proceedings of the 5th IJCAR ATP System Competition. Edinburgh, United Kingdom, 2010.
32. 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.

33. G. Sutcliffe. Proceedings of the CADE-23 ATP System Competition. Wroclaw, Poland, 2011.
34. G. Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Communications*, page To appear, 2012.
35. G. Sutcliffe, M. Suda, A. Teyssandier, N. Dellis, and G. de Melo. Progress Towards Effective Automated Reasoning with World Knowledge. In C. Murray and H. Guesgen, editors, *Proceedings of the 23rd International FLAIRS Conference*, pages 110–115. AAAI Press, 2010.
36. C. Walther. A Many-Sorted Calculus Based on Resolution and Paramodulation. In Bundy A., editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 882–891, 1983.