# Defining the meaning of TPTP formatted proofs

Roberto Blanco, Tomer Libal and Dale Miller

Inria & LIX/École polytechnique
`{roberto.blanco,tomer.libal,dale.miller}@inria.fr`

**Abstract**

The TPTP library is one of the leading problem libraries in the automated theorem proving community. Along the years, support was added for problems beyond those in first-order clausal form. Another addition was the augmentation of the language to support proofs outputted from theorem provers and the maintenance of a proof library, called TSTP. In this paper we propose another augmentation of the language for the support of the semantics of the inference rules used in these proofs.

## 1 Introduction

A key element in optimizing the performance of systems is the ability to compare them on a common benchmark. This benchmark, for the automated theorem proving community, is depicted via the 'Thousands of Problems for Theorem Provers' (TPTP) library [15]. A part of the library success lies in its accompanying syntax, which is both intuitive to read and strong enough to encode all kinds of problems. Another advantage of the syntax is its predicative form which allows the use of logic programming for parsing and other utilities. As part of the evolution of the library and its syntax, a support for proofs was added. In order to support the proof library, called 'Thousands of Solutions from Theorem Provers' (TSTP), an augmentation to the syntax was required for the support of different types of proofs, in particular directed acyclic graph proofs. This syntax allows to describe these proofs in a structural way by defining the elements involved in each step as well as the inference used and some additional annotations. One shortcoming of this format is its emphasis on syntax only and its inability to describe precisely the semantics of the inferences used.

The increased complexity of noways theorem provers brought with it a stronger need for proof certification. Errors in proofs can result from several sources, from bugs in the code to inconsistencies in the object theory. In order to amend that, several tools for proofs certification were implemented which allows for a better confidence in the theorem provers output. These tools, though, had so far only a small success in the general community. The main reason for that is that a key component in certifying proofs is the ability to understand the semantics of the object calculi, without which, certification itself will rely heavily on sophisticated proof search methods and therefore, prone to the same problems as those of the theorem provers.

The difficulty in understanding the semantics of the object calculi lies in the gap between the implementors of theorem provers and those of the proof certifiers. The normal process currently for certifying the output of a certain theorem prover is for a dedicated team on the certifier side to try to understand the semantics of each inference rule of the object calculus. This approach suffers many times from missing documentation, different naming and versioning of software, and insufficient information in the proofs themselves [3].

One way to overcome this gap is to supply the implementors of theorem provers with an easy to use and well-known format in which to describe the semantics of their inference rules. This format should be general enough to allow both precise (functional) definitions - translating a proof in the object calculus into a proof in another, trusted and well known, calculus - and somewhat informal definitions, which hint on the right way to understand the object calculus, without specifying how to actually reconstruct a proof.

In this paper we propose to use a format which is well known for the implementors of theorem provers - the TPTP format itself - for the purpose of describing not only problems and proofs, but the semantics of proofs as well. This will make a TPTP file an independent unit of information which could be used for certification as well.

An additional advantage of using the TPTP format to specify semantics is the same one mentioned above for building tools for the TPTP library. The predicate logic form of the problems, their solutions, and now, also their semantics, will allow proof certifiers which are based on logical programming to easily be able to access the semantics and will diminish the gap between the theorem provers and their certifiers further. Such a logic program, such the `checkers` proof certifier [3], will only require minor computations to be applied to the input files, if any. Those, making the trust of the certification process stronger.

The paper is organized as follows. In the next section we present and describe both the TPTP syntax and the notion of using predicates in order to define the semantics of logics. In section 3 we present and discuss the minor augmentations needed in the TPTP format in order to support the ability to use this format to denote semantics. Section 4 is devoted to the full description of three examples from three different theorem provers. The last section suggests some advantages of using this approach and conclude the work.

## 2 Preliminaries

### 2.1 Syntax in TPTP

A beneficial side effect of the TPTP library as gold standard test for automated theorem provers is the development of a language to express logic problems —and their solutions— and its adoption among the theorem provers whose input it aspires to be. It is no coincidence that this language is credited as one of the keys to the lasting success of the project [15].

The TPTP language is based on a core language that builds upon Church's simple theory of types, called THF0 [1]. It supports three different languages: first-order form (FOF) and clause normal form (CNF) for untyped first-order logic, typed first-order form (TFF), and typed higher-order (THF). All three are defined on top of the core language THF0, itself a "syntactically conservative extension" of FOF as one of the primeval components of the language, and a subset of the full-fledged THF. The syntax is based on the concept of annotated formulae and is expressive enough to structure proofs and embed arbitrarily complex information as annotations.

Among the stated design goals of the format extensibility and readability (both by machines and logicians) figure prominently. In addition, care has been taken to ensure the grammar remains compatible with the logic programming paradigm, and TPTP documents are, in fact, valid Prolog programs.

For defining a formula using the TPTP format, one uses the following template:

```
Lang(Name, Role, Formula, Annotations)
```

where *Lang* $\in \{$`cnf,fof,tff,thf,tpi`$\}$, *Role* describes the role of the formula - i.e. 'axiom' or 'type', $Formula$ is the encoding of the formula in the respective language and *Annotations* is an optional additional information.

The information in the first three arguments, *Lang*, *Role* and $Formula$, is enough to describe most kinds of problems. Since further structure is required in order to describe proofs, an extensive use of the *Annotations* field is being made.

A template for defining structural derivations is the following:

```
Lang(Name, Role, Formula, inference(Rule,AdditionalInfo))
```

Where *Rule* denotes the name of the inference rules used and *AdditionalInfo* normally refers to the names of the formulae which were used in order to apply this rule. *Formula*, as before, is an encoding of the derived formula in the respective language.

Another useful feature of TPTP is the `include` directive, which performs a syntactical inclusion of one file into another. This directive may help reducing redundancies.

## 2.2 Denoting semantics as logic programs

As already mentioned, one way to formally describe the semantics of an inference rule is by a functional translation of an instance of this rule into a derivation in another, well-known, calculus. Logic programming generalizes this idea by allowing also some notion of proof search in this calculus. Using logic programming, one does not have to give a precise functional translation but a set of predicates which ensures that the search adheres to some requirements, such as bounded space.

A precise definition of this approach and the infrastructure required to support it were given by Miller [8] and were implemented, among others, in the `checkers` proof certifier [3]. The basic idea was to program a set of predicates which will guide the search in the target calculus. By guiding the search for a derivation of an instance of an inference rule in a well-known calculus, these set of predicates can be considered as denoting the semantics of this inference.

The definition of these predicates using the TPTP syntax is the ultimate goal of our proposal, and before illustrating it we need to present the underlying principles behind the idea.

First, and critically, semantic descriptions are not "one size fits all": there is an underlying trade-off between space and time. More detailed semantic translations, insofar as the information provided is useful, produce more efficient verifications; conversely, high-level, conceptual descriptions may serve as guidance but cannot be used to generate a constructive decision procedure without additional information, or search. On one extreme are fully functional translations of inference rules, on the other minimal but sufficient hints to allow a possible reconstruction of a proof in an independent calculus. The later is a bit more general than that defined by Miller, which requires these hints to be sufficiently strong as to be able to guide the proof search. In contrast, here we consider the full spectrum of implicit vs. explicit reconstruction.

For example, suppose we wish to obtain a proof of a formula $A \wedge B \wedge C$. It may simply be stated that to do this separate proofs for $A$, $B$, and $C$ are needed:

$$\frac{A \quad B \quad C}{A \wedge B \wedge C}$$

To understand the meaning of this inference rule, one can try to infer the conclusion from the hypotheses using a well-known calculus, the sequent calculus for example, which tells us that in order to derive the original goal, two proofs are needed, one for $A$ and a second one for $B \wedge C$, and then divide in turn this second composite proof into sub-proofs of $B$ and $C$:

$$\frac{A \quad \dfrac{B \quad C}{B \wedge C}}{A \wedge B \wedge C}$$

The question of whether we can trust the first inference relies on the fact that its semantics is defined by the second, in the sense that it constitutes a formal derivation of the intended meaning, namely, that one can obtain a proof of the conjunction of three goals from proofs of each of those goals. Trust in the calculus of choice extends to trust in the inference rules that it can justify. Contrariwise, consider an alternative candidate for an inference rule.

$$\frac{A \quad B \quad C}{A \vee B \vee C}$$

It can be proved that a reasonable calculus will be unable to derive an inference of this shape. In the absence of a trustworthy proof reconstruction of the postulated inference rule, its validity cannot be accepted.

Consider now the more realistic case of paramodulation [11], a concrete instance of which we study in section 4.1. In this case, an explicit functional translation of the formal definition is far from being trivial. Conversely, it may be stated, more informally, that paramodulation handles equality modulo reflexivity: that is to say, the transitivity and symmetry axioms can be used to simulate this rule in a logic without explicit handling of equality (note that reflexivity axioms must be given externally for the equality procedure to be complete).

By applying some additional effort, this approach was implemented successfully in `checkers` and is capable of guiding the proof search for arbitrary instances of the paramodulation rule. This implies, therefore, that supplying the two axioms consists of enough information to help the automatic certification of this inference rule.

## 3 Thousands of Semantically Annotated Solutions for Theorem Provers (TATP)

In order to define the semantics of logical formulae of order $n$, one needs to use the syntax of the meta-level, i.e. formulae of order $n+1$. As seen in section 2, TPTP is equipped with the necessary syntax in order to define formulate of arbitrary finite order. For example, in order to define the provability of a classical first-order quantifier, one can use the following formula in $\lambda$Prolog notation: $pr(\forall x.Bx)$ `:-` $\Pi x.pr(Bx)$ [4] where $\Pi$ is the meta-level universal quantifier.

TPTP proofs are already annotated by the inference rules that are used in order to derive the formula. This annotation, though, lacks the semantics and normally cannot be understood by a person not familiar with the calculus in which the proof is written in.

We will harness therefore the power of the TPTP `thf` syntax in order to allow the implementor of a theorem prover to include semantics information about these annotations. Note that by using the TPTP `include` directive, one doesn't need to include these definitions in every proof generated but just define them once.

In order to allow such a use, we can first define a new `role` for formula definitions. We therefore add the following directive to the TPTP syntax:
`<formula_role> :== semantics`.

A TPTP semantics definition will have the following form:

```
thf(Name, semantics, Formula, Annotations)
```

where *Name*, by convention, should consist of the prover name, underscore and then the inference name which was used in the proof. Next, we allow arbitrary higher-order typed formula in the semantics definitions. In the rest of the paper, though, we ignore typing information in order to focus on clarity. Note that the logical programming language $\lambda$Prolog requires formula to be in higher-order hereditary-Harrop form [9]. Therefore, it would be advantageous to define the semantics in this form as then one could use, with minimal intervention, proof checking software like `checkers` to verify proofs and the proofs will be, therefore, self-contained.

A question we still need to answer is how one can define the semantics of an inference rule and how can we make the task as simple as possible. The decision taken here is to allow the implementor of a theorem prover to use, in order to define the semantics of his own rules, the inference rules and the theory of any other calculus. In general, the implementor of a resolution theorem prover can decide to specify the semantics using the inference rules of another theorem prover. It would be better though, to specify the semantics using the inference rules of a well known calculus, like the original resolution calculus by Robinson [12], for example.

When defining proofs in the TPTP format, the information of which inference rule to used is supplied using the annotation directive. We will do the same and use this directive in order to supply the information of what calculus is being used in order to define the semantics.

In order to achieve that, we will add the following directives to the TPTP syntax:

```
<source> ::= <calculus_info>
<calculus_info> ::= calculus(<calculus_name><optional_info>)
<calculus_name> ::= <atomic_word>
```

Using these directives, the user can specify the name of the calculus used and supply additional information, such as the name of a paper where this calculus is defined.

The last remaining task is to be able to bind the instances of the inferences rules in the proofs to their semantics definitions. We suggest the following convention - in order to specify an inference call of dag form, which are the ones used in proofs, the user will use the following predicate: `<inference_rule>(`$f, f_1, \ldots, f_n$`)` where $f$ is the derived formula and the remaining formula are the ones used in the derivation. Examples of this convention are given next.

## 4 Examples

We demonstrate the use of the annotations on three examples taken from inference rules used by three different theorem provers.

### 4.1 Paramodulation in the E prover

This inference rule has the following form
`cnf(`$c_3$`,role, clause,inference(pm,[status(thm)],[`$c_1, c_2$`,theory(equality)])).`
where `role` and `clause` defines the role and the content of the clause and $c_1$, $c_2$ and $c_3$ are the two clauses on which the rule is applied and the result, respectively. The semantics of this rule are similar to the semantics of the paramodulation rule [11] but are symmetric for both $c_1$ and $c_2$.

To define the semantics of this rule, we will use as foundation the following definition of paramodulation.

**Definition 1** (Paramodulation [11]). *Given clauses $A$ and $\alpha' = \beta' \vee B$ (or $\beta' = \alpha' \vee B$) having no variable in common and such that $A$ contains a term $\delta$, with $\delta$ and $\alpha'$ having a most general common instance $\alpha$ identical to $\alpha'[s_i/u_i]$ and to $\delta[t_j/w_j]$, form $A'$ by replacing in $A[t_j/w_j]$ some single occurrence of $\alpha$ (resulting from an occurrence of $\delta$) by $\beta'[s_i/u_i]$, and infer $A' \vee B[s_i/u_i]$.*

The E prover implements a variation of paramodulation in its `pm` rule, expressed in TPTP syntax with the following form:

```
cnf(ClauseId, Role, Formula, inference(pm, [status(thm)],
    [SourceId1, SourceId2, theory(equality)]))
```

Here, `SourceId1` and `SourceId2` are the two clauses to which the paramodulation rule is applied to obtain `ClauseId`, corresponding to the formula given by `Formula` and with role `Role`. The semantics of this rule are similar to the semantics of the paramodulation rule (from [11] and our definition), with the peculiarity that the tactic presents symmetry for both `ClauseId1` and `ClauseId2`.

To produce the full definition we proceed in two steps. First we present a TPTP formula that denotes the semantics of the `pm` rule.

```
thf(eprover_pm, semantics, Formula, calculus(paramodulation,
    [p.5 in [11]]))
```

Second, we define `Formula` as the mapping between the specific variation of paramodulation defined by the E prover (namely, the `pm` tactic) and the canonical semantics derived from the definition:

```
∀ SourceId1, SourceId2, ClauseId:
pm(SourceId1, SourceId2, ClauseId)
    ⇐ paramodulation(ClauseId, SourceId1, SourceId2)
    ∨ paramodulation(ClauseId, SourceId2, SourceId1)
```

Where, as usual, free variables are universally quantified; we have made this quantification explicit in the present formulation.

## 4.2 Binary resolution in Vampire

Vampire [10] is a theorem prover that implements the superposition calculus. Proofs proceed by saturation and rely on redundancy elimination and a wide range of advanced techniques to maximize performance, one of its original design goals. It features a rich collection of inference rules and supports the TPTP syntax, including various extensions. After Vampire 3.0, there have been no public releases available for download, extending to version 4.0 as available in TSTP and System on TPTP. In the absence of a technical manual or an in-depth systems description, it becomes necessary to inspect TSTP to approximate the actual semantics of the program.

**Definition 2** (Binary resolution [12]). *Given two clauses $A = a_1 \vee \ldots \vee a_m$ and $B = b_1 \vee \ldots \vee b_n$ and a pair of complementary literals, one from each clause, i.e., $a_i = \neg b_j$ or $\neg a_i = b_j$, the resolution rule derives a new clause with all the literals except the complementary pair: $C = a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_m \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_n$.*

The binary resolution rule includes the possibility of applying a unification procedure to a pair of unifiable literals, and substituting the most general unifier in the resolvent $C$. Some categories of binary resolution can be defined. These are not necessarily mutually exclusive:

- Positive resolution, if one of the parent clauses is a positive clause, i.e., all its literals are positive.
- Negative resolution, if one of the parent clauses is a negative clause, i.e., all its literals are negative.

- Unit resolution, if one of the parent clauses is a unit clause, i.e., formed by exactly one literal.

Vampire outputs natively to TPTP in addition to its own internal format, closer to that of Prover9 that we treat in the next subsection. Now, we consider the TPTP output of the basic resolution rule.

```
fof(ClauseId, plain, Formula,
    inference(resolution, [], [SourceId1, SourceId2]))
```

The translation takes this to the higher-order formula and adjusts the annotation information in the inference name to point to the name of the logic program that implements the procedure.

```
thf(vampire_resolution, semantics, Formula, calculus(hol))
```

Where `Formula` is defined to be

```
 ∀ S1, S2, R1, R2: resolution(S1, S2, R1 ∨ R2)
   ⇐ ∃ L: memb_and_rest(S1, L, R1) ∧ memb_and_rest(S2, ¬L, R2)
   ∨ memb_and_rest(S1, ¬L, R1) ∧ memb_and_rest(S2, L, R2).
```

Here we use a list-like predicate representing a "member and rest" operation to select a literal from a clause and yield a copy of the clause without the chosen literal, but still use disjunction as clause concatenation.

A clause produced by binary resolution is specified by the two premise clauses and, considering each of these in CNF form, and in turn a CNF form as an indexed list of disjuncts, by the disjunct from each clause that is involved. We also assume a predicate specifying, and therefore declaratively implementing, binary resolution, that acts on formulae and can check whether the specified application of resolution yields the target formula.

## 4.3 Hyperresolution in Prover9

Prover9 [7] is a theorem prover based around the techniques of resolution and paramodulation, and the successor of the Otter theorem prover. The last available version is 2009-11A, dated November 2009. While development has since ceased, the tool remains in use. Prover9 does not produce output in TPTP format, and therefore TSTP contains unparsed execution traces. However, the input and output formats of the prover are simple and well documented, and their semantics can be easily formalized. Interestingly, such a translation procedure offers the possibility of generating the native TSTP output that is missing from the problem library, together with its semantics.

In this subsection we consider hyperresolution [5], one of the primary tactics used by Prover9. An informal definition of the inference rule follows.

**Definition 3** (Hyperresolution [5]). *Assume a nucleus clause $A$, nonpositive, with a number $k$ of negative literals $\neg a_{i_1}, \ldots, \neg a_{i_k}$, and as many satellite clauses $B_1, \ldots, B_k$, each of which resolves on of those negative literals, i.e., $B_j = \ldots \vee a_{i_j} \vee \ldots$. The hyperresolution rule resolves all the negative literals in the nucleus, each with its satellite, producing a positive clause $C$.*

Hyperresolution can be seen as a sequence of applications of binary resolution. It is likewise possible to reverse polarities and speak of negative hyperresolution. A related concept is that of unit-resulting resolution, where the satellites are unit clauses and the nucleus is reduced down to a single literal, i.e., another unit class.

Prover9 implements this as the `hyper` tactic. The output language divides files in several sections, one of which contains proofs presented as justifications: a sequence of clauses, each derived from the starting clauses or by previous derivations in the chain. Inferences in each step of the justification are themselves lists of tactics: exactly one primary tactic, possibly followed by a number of secondary tactics. Hyperresolution is one of the primary tactics, and for simplicity we will consider its treatment in isolation. It will become clear that sequences of secondary steps follow an analogous compositional pattern.

An example of hyperresolution step is this: `hyper(59, b,47,a, c,38,a)`. Here, clauses are referenced by Arabic numerals and literals within a clause by letters: `a`, `b`, `c`... Though represented by a plain list, it is to be interpreted as the nucleus clause followed by a sequence of triples, each specifying a satellite clause and the literals that are involved to produce the next clause in the hyperresolution chain. Thus, in the example, 59 is the nucleus; applying binary resolution to its second literal and the first literal of clause 47 produces a new clause; and applying binary resolution again, this time between the third literal of the new clause and the first literal of 38, produces the final result.

Ignoring labels and secondary steps in Prover9 syntax, an instance of the hyperresolution rule is expressed as follows.

```
Clause Formula. [hyper(Nucleus,
                       First1,Satellite1,Second1,
                       ...,
                       FirstN,SatelliteN,SecondN)]
```

For the translation to our extension of TPTP, we provide the logic program that implements the procedure and define the mapping.

```
thf(prover9_hyperresolution, semantics, Formula, calculus(hol))
```

And here `Formula` defines the logical semantics of hyperresolution recursively, in terms of the same generic (binary) resolution procedure that was used to model the tactic in VAMPIRE.

```
  ∀ S1, S2, R: hyperresolution([S1, S2], R)
    ⇐ resolution(S1, S2, R).
  ∀ S1, S2, Ss, R: hyperresolution([S1, S2 | Ss], R)
    ⇐ ∃ R': resolution(S1, S2, R')
        ∧ hyperresolution([R' | Ss], R).
```

Insofar as the sequence of clauses and the expected final formula are known, we can ignore the triples passed as additional info and entrust the backtracking search mechanism to find an appropriate application of hyperresolution (assuming one exists). Consequently, the encoding drops the conjunct selection guidance given by Prover9 and represents a more general problem, solvable directly by the definition given here.

## 4.4 Object- to meta-level lifting of disjunction in LEO-II

As a final example, we consider a two-level logic tactic in the theorem prover LEO-II. In particular, we consider the `extcnf_or_pos` tactic, which is responsible for lifting a disjunction from the object level to the meta level of the logic [14]. The rule has the following definition:

$$\frac{C \vee [A \vee B]^{tt}}{C \vee [A]^{tt} \vee [B]^{tt}}$$

The tool expresses the application of this rule natively in TPTP syntax as follows.

```
thf(ClauseId, plain, Formula,
    inference(extcnf_or_pos, [status(thm)], [SourceId]))
```

It should be noted that atoms in LEO-II are labeled with either true or false using the TPTP notation `F = $true`. Once a substitution is applied, atoms can become more complex formulae. Concretely, this inference rule is used to translate the object-level disjunction into the clause-level one.

To provide the semantics of this rule, we use a higher-order logic formulation:

```
thf(leo2_extcnf_or_pos, semantics, Formula, calculus(hol))
```

And here `Formula` supplies the following definition for the underlying semantics.

```
∀ ClauseId, SourceId:
extcnf_or_pos(ClauseId, SourceId)
    ⇐ (((∀ C: C ⇔ C = ⊤) ∧ SourceId) ⇒ ClauseId)
```

Again, with explicit quantification. It is easily seen that using the additional axiom one can easily use any calculus for higher-order logic to prove this normalization rule.

## 5 Discussion and Conclusion

Even when we restrict our attention to the resolution theorem provers community, there are several different approaches for proof certification. Sutcliffe [16] proposed to use the proof derivations in the TSTP library as a skeleton, which one can use, with the help of theorem provers, to reconstruct a proof. The Dedukti proof certifier [2] is a universal proof certifier which was successfully used in order to certify proofs of the iProver resolution theorem prover [6]. The closest to the approach presented in this paper is that of the system `checkers` [3], which uses logic programming in order to encode semantics and reconstruct proofs and was used in order to partially certify Eprover's [13] proofs. While the first method is based on using theorem provers for filling in the missing semantics in TPTP proofs, the later two systems are based on a concrete effort to denote the semantics of different theorem provers using functional and relational approaches, respectively. This effort is normally done by a different team than that which implemented the theorem prover and which has the deepest knowledge about the actual semantics of its calculus.

The approach which was taken in this paper tries to make this effort easier and more accessible to the implementors of theorem provers. First, the language used to denote the semantics is well known to the implementors as it is already used for both the input problems and to output proofs. Second, unlike the last two systems mentioned, the implementors have a high degree of flexibility in which to define the semantics and are not restricted by external notions such as efficient or effective translations. This indeed put at risk the ability to mechanize these definitions into an actual certifier for the system but as mentioned in the paper, the parts which cannot be mechanized as given can, at least, be used to bring mechanization closer with some further help, for example by the certification team.

The aims of this approach is to convince the implementors of theorem provers that even semi formal semantics, which can easily be defined using the approach presented, are useful for the purpose of full certification of their provers. The implementors control thus fully the effort required by them in order to generate the semantics. The examples given in this paper ranges from a minimal effort of just specifying axioms to a greater effort of defining a full translation. While the second can be used efficiently by any of the two systems described at the beginning of this section, the first method requires only minimal additional approach in order to be used for proof reconstruction by a system like `checkers`.

To conclude, TPTP can serve as a format for specifying the semantics of proofs for various degrees of concreteness. By using the same format for both problems, proofs and semantics, implementors are encouraged to consider the semantics as part of the implementation effort. This effort can both serve as a documentation of the internal calculus and as an implementation of the semantics which can be later used for proof checking.

**Acknowledgments.** This work has been funded by the ERC Advanced Grant ProofCert.

## References

[1] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. Thf0–the core of the tptp language for higher-order logic. In *Automated Reasoning*, pages 491–506. Springer, 2008.

[2] Guillaume Burel. A shallow embedding of resolution and superposition proofs into the $\lambda\Pi$-calculus modulo. In J. C. Blanchette and J. Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)*, volume 14 of *EPiC Series*, pages 43–57. EasyChair, 2013.

[3] Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier checkers. In Hans De Nivelle, editor, *Proceedings of the 24th Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, number 9323 in LNCS, pages 201–210. Springer, 2015.

[4] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in LNCS, pages 61–80, Argonne, IL, May 1988. Springer.

[5] Christian G. Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In *Handbook of automated reasoning*, pages 1791–1849. Elsevier Science Publishers BV, 2001.

[6] Konstantin Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.

[7] William McCune. Prover9 and Mace4. `https://www.cs.unm.edu/~mccune/prover9/`

[8] Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 54–69, 2011.

[9] D. Miller and G. Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, June 2012.

[10] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[11] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 298–313. Springer Berlin Heidelberg, 1983.

[12] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, January 1965.

[13] Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.

[14] Nik Sultana and Christoph Benzmüller. Understanding leo-ii's proofs. In *IWIL@ LPAR*, pages 33–52, 2012.

[15] 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.

[16] Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.