EPiC Series in Computing

Volume 40, 2016, Pages 78–90

IWIL-2015. 11th International Workshop on the Implementation of Logics

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. Over time, support was added for problems beyond those in first-order clausal form. TPTP has also been augmented with support for various proof formats output by theorem provers. Such proofs can also be maintained in the TSTP proof library. In this paper we propose an extension of this framework to support the semantic specification of the inference rules used in proofs.

1 Introduction

A key element in optimizing the performance of systems is the ability to compare them on common benchmarks. In the automated theorem proving community, such benchmarks are available via the “Thousands of Problems for Theorem Provers” (TPTP) library [31]. A part of the library’s success lies in its syntactic conventions, which are both intuitive to read and rich enough to encode many kinds of problems. Another advantage of its syntax is its simple structure that allows one to easily write parsers and other utilities for it. 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), the syntax needed to be extended to support different types of proofs, in particular, directed acyclic graph proofs. This syntax allows for the description of proofs as a series of steps which are themselves encoded collections of inference rules and some additional annotations. One shortcoming of this format is its emphasis on syntax and its inability to describe precisely the semantics of the inferences used.

The increased complexity of today’s automated theorem provers has brought with it a need for proof certification. Errors in proofs can result from several sources ranging from bugs in the code to inconsistencies in the object theory. In order to improve this situation, several tools for proof certification have been implemented that can improve our confidence in the proofs output from theorem provers. These tools can be classified into two groups. First it is possible to actually prove that a theorem prover is formally correct (see, for example, Ridge and Margetson [24]). The second group consists of tools for verifying, not the theorem provers themselves, but their output. This group can be further divided into two groups. The first group consist of systems for replaying proofs using external theorem provers for verifying specific steps. Among these, one can count the general tools Sledgehammer [22, 3], PRocH [13] and GDV [29] as well as more specific efforts such as the verification of E prover [27] proofs using Metis [22]. The second group contains tools having an encoding or a translation of the semantics of theorem provers, which is then used in the replaying process. This last group can be divided again into specific tools, such as Ivy [17] and the encodings of MESON [15] and Metis [12] in HOL Light and Isabelle, respectively, and general tools such as Dedukti [2] and ProofCert [19].

These various classes of tools represent different approaches to proof certification. While we can have a high level of trust in the correctness of the provers in the first group, their performance cannot be compared to that of the leading theorem provers like E [27] and Vampire [23]. The remaining groups do not pose restrictions on the provers themselves but the generality and automation of those in the second group come with the cost of using an external theorem prover and translations, which might result in reduced confidence. The last two groups require an understanding of a theorem prover's semantics so that one can guarantee the soundness of proofs by their reconstruction in a low level formal logic. Working with an actual proof has several advantages as one can apply proof transformations and other procedures. The last group has additional advantages over the previous one: a single certifier can be written that should be able to check proofs from a range of different systems and the existence of a common language for proofs allows for the creation of proof libraries and marketplaces [19].

Those tools in the last group have, so far, only limited success in the general community. One reason for this is that understanding and specifying the semantics of proofs requires sophistication in the interplay between deduction and computation (whether via function-style rewriting or proof-search).

The difficulty in understanding the semantics of the object calculi lies in the gap between the implementers of theorem provers and the implementers of the proof certifiers. Currently, the normal process 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 names and versions of actual software, and insufficient information in the proofs themselves [5]. This gap is enlarged by the fact that teams of implementers and certifiers can reside in different locations or even work in different periods, thus making the communication between them difficult or even impossible.

One way to overcome this gap is to supply the implementers 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 specifications to range from precise (functional) definitions—translating a proof in the object calculus into a proof in another, trusted and well-known calculus—and informal definitions, with hints on the right way to understand the object calculus without needing to specify how to actually reconstruct a formal proof.

In this paper we aim at helping to reduce the gap mentioned above between those who produce proofs and those who must certify them. We propose to use a format which is well-known to the implementers of theorem provers—the TPTP format itself—for the purpose of describing not only problems and proofs but also the semantics of proofs. This will make a TPTP file an independent unit of information which can 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 to easily access the semantics and will further diminish the gap between the theorem provers and their certifiers. For example, the `checkers` proof certifier [5] (written using logic programming), will only require minor computations to be applied to the input files, if any. Such simplicity helps to improve the trust of the certification process.

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 four examples from four different theorem provers. The concluding section suggests some additional advantages of using this approach.

2 Preliminaries

2.1 Syntax in TPTP

A beneficial side effect of the TPTP library as a standard test suite for automated theorem provers is the standardization of a language to express logic problems and their solutions. It is no coincidence that this standardization on the language is credited as one of the keys to the success of the TPTP project [31].

The TPTP syntax is built upon a core language called THF0 [1]. This core can be restricted to support a number of interface languages: untyped first-order logic as first-order form (`fof`) and clause normal form (`cnf`), typed first-order form (`tff`), and typed higher-order (`thf`). Furthermore, all of these can be used in combination with a process instruction language (`tpi`) for the manipulation of formulas. The concrete syntax revolves around the concept of annotated formulas and is expressive enough to structure proofs and embed arbitrarily complex information as annotations.

Among the stated design goals of the format, both extensibility and readability (by machines and logicians) figure prominently. In addition, care has been taken to ensure that 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 templates:

```
Language(Name, Role, Formula).
Language(Name, Role, Formula, Annotations).
```

where `Language` ∈ {`cnf,fof,tff,thf,tpi`} (see above for the list of interface languages), `Role` describes the role of the formula —i.e. ‘axiom’ or ‘type’—, `Formula` is the encoding of the formula in the specified language and `Annotations` contains optional additional information.

A template for defining structural derivations is the following:

```
Language(Name, Role, Formula, inference(Rule, Info, Parents)).
```

Here `Rule` denotes the name of the inference rules used, `Info` optionally specifies additional information, like the SZS output value of the inference and `Parents` also optionally refers to the names of the formulas which were used in the application of this rule. `Formula`, as before, is an encoding of the derived formula in the respective language. The SZS ontology [30] referred to above supplies a set of inference properties, such as theoremhood, satisfiability, etc., which give some semantical information. It should be noted that this information might suffice for proof replaying using an external prover [22, 13, 29], but does not fully help in understanding the semantics of the inference rules themselves.

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

2.2 Denoting semantics as logic programs

As already mentioned, one way to formally describe the semantics of an inference rule is by translating an instance of this rule into a derivation in another, well-known, calculus. This translation can be *determinate* or *nondeterminate*: in the latter case, a logic programming implementation of the translation could allow for that nondeterminism to be explored using backtracking search. Nondeterminism in the specification of proof semantics has been considered in other systems as well. In particular, nondeterminism is allowed in the *Foundational Proof Certificates* (FPC) framework [7, 19] where client-side inference rules (i.e., rules implemented in theorem provers) are translated into low-level rules of sequent calculus. The `checkers` proof certifier [5], based on the FPC framework, used the λProlog logic programming language [20] to provide for a backtracking search approach to exploring any nondeterminism in such translations. The basic idea is 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, this set of predicates can be considered as denoting the semantics of this inference.

Before describing how we plan to use the TPTP framework to specify the translation of inference rules, 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 (for storing a proof) and time (for checking a proof). 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. At one extreme are fully determinate translations of inference rules and at the other extreme are minimal but sufficient hints to allow a possible reconstruction of a proof in an independent checker. 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 [25], 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 provides enough information to assist the automatic certification of this inference rule.

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

In order to have the cleanest and most declarative treatment of one logic (i.e, the logic of the client prover) within a second logic encoding inference rules and their associated proof search, we shall make use of the notion of *order* of THF0. In this setting, defining the semantics of logical formulas of order n employs a meta-level logic of order $n + 1$. For example, if our client proofs are only propositional formulas (order 0) then the first-order fragment of THF0 suffices. However, if our client proofs are first-order formulas, then we employ directly the second-order subset of THF0. As seen in section 2, TPTP is equipped with the necessary syntax necessary to define formulas of an arbitrary finite order. It is largely for this reason that we will employ λProlog [20] to automate[1] the translation of inference rules, since the logic underlying λProlog is close to that underlying THF0 (both are closely related to Church's Simple Theory of Types [8]). For example, in order to define the provability (via the predicate *pr*) of a classical first-order quantifier, one can use the following λProlog clause:

$$pr(\forall x.Bx) \ \texttt{:-}\ \Pi x.\ pr(Bx).$$

where $\forall$ is the object-logic universal quantifier and Π is the meta-level universal quantifier. The implementation of λProlog deals directly with the many issues related to binding, substitutions, eigenvariables, and unification [10].

TPTP proofs are already annotated by the inference rules that are used in order to derive the formula. These annotations, however, lack a formal semantics and they cannot normally be understood by a person not familiar with the details of the system that outputs those annotations. We propose to use the TPTP `thf` syntax in order to allow the implementer of a theorem prover to include semantical information about their inference rules, thereby replacing imprecise and specialized annotations with more formal annotations. Note that by using the TPTP `include` directive, one does not 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).
```

[1]Both Teyjus [21] and ELPI [9] are implementations of λProlog.

where `Name`, by convention, should consist of the prover name, underscore and then the inference name which was used in the proof, for example "`e_pm`". `Formula` can contain an arbitrary higher-order typed formula denoting 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 λProlog requires formulas to be in fragment of higher-order logic called *hereditary-Harrop formulas* [20]. As has been shown elsewhere (for example, [10] and [20, Chapter 9]), this fragment of logic is able to elegantly specify a variety of inference rules. Thus, if one can define the semantics of inference rules using these formulas, one could use, with minimal intervention, proof checking software like `checkers` to verify proofs. Lastly, `Annotations` can contain additional (informal) information which can help understanding the semantics. These annotations may include the name of a target logic which can be used for proof reconstruction or a reference to a paper which defines the target or object logics.

A question we still need to answer is how one can define the semantics of an inference rule and how we can make that task as simple as possible. The decision taken here is to allow the implementer of a theorem prover to use, in order to define the semantics of their own rules, the inference rules and the theory of any other calculus. In general, they can decide to specify the semantics using the inference rules of another theorem prover. However, it would be preferable to specify the semantics using the inference rules of a well known calculus, like the original resolution calculus by Robinson [26], for example. The approach we are discussing here (with examples in the next section) stands in contrast to what is being done in the ProofCert project [18]. In that project, the meanings of all inference rules are "compiled" into a low-level proof system (representing an "assembly language" for inference). We do not insist on employing that framework, opting instead for a less tedious and more high-level approach to providing some useful information about the inference rules used in a specific theorem prover.

It should be noted that according to the above conventions, one has full control over the amount of detail and choice of the target calculus. A large amount of detail might enable a precise proof reconstruction in a fine grained calculus, for example, in the sequent calculus (the ProofCert project does exactl1y this, for example). We want to stress here that since the aim of these definitions is to communicate information about the semantics, such detailed information is not necessary. There can be benefits for both the certification team and the implementers in specifying information about the semantics of their own rules using the highest level calculus known to the community. This will make the definition simpler and will also contribute to the modularity of a certification tool since the certifiers will only need to implement the semantics of the high level calculus, the semantics of which being widely known. For example, all superposition provers are using variants of the paramodulation inference rule [25]. Defining the semantics of these variants can be done by a number of individualized, detailed descriptions or be based on the known notion of paramodulation. It seems more intuitive and simple for implementers to choose the second option and let the certification team implement the general semantics of paramodulation. This is the approach taken in `checkers` and described below in the examples. A fine grained description of the semantics of paramodulation can be found, for example, in [6].

When defining proofs in the TPTP format, the information of which inference rule to use 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 to define the semantics. To this end, 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 inference rules in the proofs to their semantics definitions. We suggest the following convention: in order to specify an inference call in DAG form, i.e. the ones used in proofs, the user will employ the following predicate:

$$\texttt{<inference\_rule>}(f, f_1, \ldots, f_n)$$

where f is the derived formula and the remaining formulas are the inputs used in the derivation. Examples of this convention are given next.

4 Examples

We demonstrate the use of THF0-style annotations on four examples taken from inference rules used by four different theorem provers.

4.1 Paramodulation in E

The E prover [27] was among the first provers to output proofs using the TPTP format. A staple on the podium at the annual CASC competitions [32], E is used by many other first- and higher-order theorem provers. E is a superposition-based, saturating, automated theorem prover based on a purely equational paradigm. As such, it implements several variants of the paramodulation rule.

Definition 1 (Paramodulation [25]). *Given clauses A and $\alpha' = \beta' \vee B$ (or $\beta' = \alpha' \vee B$) having no variables in common and such that A contains a term δ, with δ and α' having a most general common instance α 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 α (resulting from an occurrence of δ) by $\beta'[s_i/u_i]$, and infer $A' \vee B[s_i/u_i]$.*

One concrete variant, for example, is given in the `pm` rule of E. This rule is applied in TPTP syntax using the following form:

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

where `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 is similar to the semantics of the paramodulation rule (from [25] 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  [25] ])).
```

where the semantics is documented by a suitable bibliographic reference. 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 [23] is a theorem prover that implements the superposition calculus and is the regular winner of the first-order division in the CASC competition over the last decade [32]. 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. Here we inspect TSTP entries produced by Vampire 4.0 to infer program semantics.

Definition 2 (Binary resolution [26])**.** *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: select(S1, L, R1) ∧ select(S2, ¬L, R2)
       ∨ select(S1, ¬L, R1) ∧ select(S2, L, R2).
```

Here the standard list selection predicate is used to pick a literal from a list-like clause and yield a copy of the clause without the chosen literal. We are still free to use concatenate clauses by way of a disjunction.

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 formulas and can check whether the specified application of resolution yields the target formula.

4.3 Hyperresolution in Prover9

Prover9 [16] 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 [11], one of the primary tactics used by Prover9. An informal definition of the inference rule follows.

Definition 3 (Hyperresolution [11])**.** *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 clause.

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 `hyper(59, b, 47, a, c, 38, a)` where 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)).
```

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 [28]. 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 formulas. 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)).
```

Here `Formula` supplies the following definition for the underlying semantics (using explicit quantifiers).

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

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 community of resolution theorem provers, there are several different approaches to proof certification. Sutcliffe [29] proposed using the proof derivations in the TSTP library as a skeleton, which one can use to reconstruct a proof (possibly with the help of theorem provers). The Dedukti proof certifier [4] is a universal proof certifier which was successfully used to certify proofs of the iProver resolution theorem prover [14]. The proof certifier closest to the approach presented in this paper is that of the system `checkers` [5], which uses logic programming in order to encode inference rule semantics and to reconstruct proofs. `checkers` has been used to partially certify E's [27] proofs. While the first method is based on using theorem provers for filling in the missing semantics in TPTP proofs, the latter two systems are stem from a concrete effort to denote the semantics of different theorem provers using deterministic and non-deterministic approaches, respectively. This effort is normally made by a different team from 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 implementers of theorem provers. First, the language used to denote the semantics is well known to the implementers as it is already used to input problems and to output proofs. Second, unlike the last two systems mentioned, the implementers have a high degree of flexibility 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 aim of this proposal is to convince the implementers 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 implementers can thus control the effort required of them in order to generate the semantics. The examples given in this paper range from the minimal effort of specifying a simple set of axioms to the 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 effort in order to be used for proof reconstruction by a system like `checkers`.

In conclusion, 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, implementers are encouraged to consider the semantics as part of the implementation effort. This effort can both serve as 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] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

[3] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *Automated Reasoning*, pages 107–121. Springer, 2010.

[4] Guillaume Burel. A shallow embedding of resolution and superposition proofs into the λΠ-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.

[5] 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.

[6] Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. To appear in the Post-proceedings of LSFA 2015: 10th Workshop on Logical and Semantic Frameworks, with Applications. Natal, Brazil. Draft dated 28 October 2015.

[7] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013.

[8] Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[9] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. A fast interpreter for λProlog. In *LPAR-20: Logic Programming and Automated Reasoning, International Conference*, 2015. To appear.

[10] 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.

[11] 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.

[12] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[13] Cezary Kaliszyk and Josef Urban. Proch: Proof reconstruction for hol light. In *Automated Deduction–CADE-24*, pages 267–274. Springer, 2013.

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

[15] Donald W Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM (JACM)*, 15(2):236–251, 1968.

[16] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[17] William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided reasoning*, pages 265–281. Springer, 2000.

[18] Dale Miller. Proofcert: Broad spectrum proof certificates. An ERC Advanced Grant funded for the five years 2012-2016, February 2011.

[19] 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.

[20] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[21] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[22] Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

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

[24] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *TPHOLs*, volume 3603, pages 294–309. Springer, 2005.

[25] 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.

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

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

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

[29] Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.

[30] Geoff Sutcliffe. The szs ontologies for automated reasoning software. In *LPAR Workshops*, volume 418, 2008.

[31] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[32] Geoff Sutcliffe and Christian Suttner. The state of casc. *AI Communications*, 19(1):35–48, 2006.