Proof-Reconstruction in Type Theory for Propositional Logic

Jonathan Prieto-Cubides[1] [2] and Andrés Sicard-Ramírez[1]

[1] Universidad EAFIT, Medellín, Colombia
asr@eafit.edu.co
[2] University of Bergen, Bergen, Norway
jonathan.cubides@uib.no

Abstract. We describe a syntactical proof-reconstruction approach to verifying derivations generated by the automatic theorem prover `Metis` for classical, propositional logic. To verify the derivations, we use `Agda`, a proof assistant for dependent type theory. We formalise in `Agda` each inference rule of `Metis` reasoning, that is, we transform `Metis` derivations into `Agda` proof-terms. In a future full-fledged integration of `Agda` with off-the-shelf automatic theorem provers, proof-reconstruction shall be a necessary component.

Keywords: proof-reconstruction, automatic theorem provers, proof-assistants, Agda, Metis

1 Introduction

An automatic theorem prover (henceforth ATP) is a program that intends to prove conjectures from axioms and inference rules of some logical system. In the last decades, ATPs are quickly becoming a key instrument in many disciplines and real applications (e. g., verifying a railway interlocking system, an operating-system kernel, or a pseudo-random number generator). Since some programming errors have been found in these programs (see, for example, [26, 6, 17]), researchers and users from academia and industry have shown an increased interest in *formally* proving the validity of ATPs' results.

In order to give confidence to the users of the ATP, many of these systems have started to include in their outputs the full derivations associated to the proved theorems. However, existing research recognizes that in many cases these derivations encode non-trivial reasoning hard to reconstruct and therefore hard to verify [32, 26].

Proof-reconstruction addresses this problem. Since many ATPs are poor-documented, this problem becomes a reverse engineering task to verify the prover reasoning. Usually, is the reconstruction is made by another and not by the developers of the ATP. Therefore, the presentation of the derivations generated by the prover plays an important role in proof-reconstruction.

To verify such automatically generated derivations by the prover, it is convenient to have them in a consistent format, that is, a full script describing the derivation step-by-step with exhaustive details and without ambiguities. For example, from a list of at least forty ATPs for classical propositional logic (henceforth CPL) —available from the Web service `SystemOnTPTP` of the TPTP World[3]— just few of them show proofs when they find the conjecture of the problem is a theorem.

One approach to address the proof-reconstruction problem is proving each deduction of the prover, the *source* system, with a formalization of the prover reasoning in a proof-assistant, the *target* system. The target system is the *proof-checker* in charge of verifying the source system

[3] http://www.cs.miami.edu/~tptp/.

reasoning for each derivation. These proof-assistants allow us to formalize the logical system used in the proofs, i. e., logical constants, axioms, inference rules, hypotheses, and theorems. A proof-reconstruction tool provides such an integration, translating the derivation generated by the prover into the formalism of the proof-assistant.

Previous studies have reported proof-reconstruction using proof-assistants based on higher-order logic where the development is at a mature stage [31, 22, 23]. Another approaches has been proposed for proof-reconstructing based on type theory in [4, 25, 26].

We describe a formal reconstruction of proofs generated by the `Metis` prover [20]—our source system—in Martin-Löf type theory [29]. To do so, we only take into account the syntax rather than the semantics of the subset of the `Metis` inference rules for the propositional logic fragment. The `Metis` reasoning was formalized in `Agda` [42]—our target system—in two libraries [34, 36] and we implemented a proof-reconstruction tool named `Athena` [35] written in `Haskell` that is able to generate `Agda` proof-terms for `Metis` derivations. In the course of this work, we uncovered and reported two programming errors in the ATP `Metis`: a bug[4] in the printing of the derivation and a soundness bug[5] in the stripping of the goal.

This paper is organized in the following way: in Section 2, some limitations of type theory are discussed from our proof-reconstruction point of view. In Section 3, we introduce the `Metis` prover. In Section 4, we show our approach to reconstruct `Metis` derivations. Related work is described in Section 5. Conclusions and suggestions for future work are presented in Section 6.

The source code accompanying this paper (programs, libraries, and examples) is available at `GitHub`:

- The `Athena` program that translates proofs generated by `Metis` to `Agda` code: http://github.com/jonaprieto/athena.
- The `agda-prop` library as a formalization in `Agda` for classical propositional logic: http://github.com/jonaprieto/agda-prop.
- The `agda-metis` library as a formalization in `Agda` to justify `Metis` derivations of classical propositional logic: http://github.com/jonaprieto/agda-prop.

«««< HEAD:pubs/paper/sections/02-introduction.tex The proof-reconstruction tool `Athena` was tested with `GHC` v8.6.5. Both libraries, `agda-prop` and `agda-metis` were tested with `Agda` v2.6.0.1 and `Agda` standard library v1.1. `Athena` jointly with `agda-prop` and `agda-metis` are able to reconstruct propositional proofs of `Metis` v2.4.20180810. We successfully reconstructed eighty theorems in classical propositional logic from the `TPTP` collection [33] to test the developments and the formalization presented in this research. ======= The proof-reconstruction tool `Athena` was tested with `GHC` 8.4.3. Both libraries, `agda-prop` and `agda-metis` were tested with `Agda` 2.5.4 and `Agda` standard library 0.16. `Athena` jointly with `agda-prop` and `agda-metis` are able to reconstruct propositional proofs of `Metis` 2.4 (release 20180207). We successfully reconstruct around eighty theorems in classical propositional logic from the `TPTP` collection [33] to test the developments and the formalization presented in this research. »»»> 078fe67b5636e860b63c9f7ccb7fedded0cdfd44:pubs/itp-2018/sections/introduction.tex

[4] Issue No. 2 at https://github.com/gilith/metis/issues/2
[5] Issue No. 4 at https://github.com/gilith/metis/issues/4

2 Type Theory

Type theory is a formalism for the foundation of mathematics. It has become a key instrument to study logic and proof theory, and follows the same basis of constructive mathematics where the witnesses of a statement, the proofs, have a fundamental role as same as their statement.

Following the Curry-Howard correspondence (see, for example, [44]), a formula corresponds to a *type*, and one proof of that formula is a *term* of the correspondent type. Therefore, inhabited types are such formulas with proofs, they are theorems.

Type theory is a formal system with a syntax and a set of derivation rules. These rules enables us to derive conclusions called *judgments* which can be seen as derivation trees [4]. Each term has associated a derivation and we refer to these derivations as *proof-terms*.

Notation. We write the typing judgments as $a : \mathsf{A}$ to denote that the term a is of type A.

Type theory can be seen as typed λ-calculus with dependent function types, evaluation of λ-terms also called *normalization* of the proof-terms, is a process of reductions with the system inference rules. Therefore, the proof verification task becomes in type theory as verifying that the proof-term has the correspondent type of the theorem. We refer to this process as *type-checking*.

This feature of type theory allow us to verify a proof generated by an ATP by reconstructing its proof-term to type-check the proof. For such a purpose, we use a proof-assistant based on type theory, `Agda`, to delegate this task as in [4]. Some limitations of the type theory point-of-view for proof-reconstruction is described in Section 2.1 and Section 2.2.

2.1 Terminating functions

To reconstruct `Metis` inference rules in type theory, we observed that some rules or their inner functions are implemented by *general recursive* functions (see the definition of these functions in [9]).

Functions defined by a general recursion can not be directly translated in type theory since it is not guaranteed that they terminate. For that reason, we follow the technique described in [3] to avoid termination problems by modifying the recursive functions to be *structurally recursive*.

A recursive function is structurally recursive if it calls itself with only structurally smaller arguments [1]. General recursive functions can sometimes be rewritten into structurally recursive functions by using for instance, the *bounded recursion* technique or other methods as in [12, 1, 10].

The bounded recursion technique defines a new structural recursive function based on the general recursive function by adding an argument. The new argument is the *bound*, a natural number given by the function complexity. In other words, the added argument will represent the number of times the function needs to call itself to get the expected outcome.

Notation. We define Prop for the type of propositions. A proposition is an expression of indivisible propositional variables (e. g., symbols $\varphi_0, \varphi_1, \dots$), and the logic constants: $\bot$, $\top$, the binary connectives ($\wedge, \vee, \supset$), and the negation ($\neg$). We use the inductive definition for propositions presented in [13]. The syntactical equality between two propositions φ and ψ is written using the symbol ($\equiv$) as $\varphi \equiv \psi$. This equality between propositions is precisely the Identity type $\mathsf{Id}_{\mathsf{Prop}}(\varphi, \psi)$ in Martin-Löf type theory. Therefore, it holds the reflexivity, symmetry and transitivity property for equality. The type of natural numbers is called Nat, and it is defined as usual, i. e., zero and succ are its data constructors. We use names and symbols for the arithmetic operations as usual (e. g., $+$, $-$, $*$). We use syntax sugar for zero, succ, $\mathsf{succ}\ \mathsf{zero}$, ..., with the decimal representation 0, 1, 2, ... as well.

Hence, one approach to define a structural recursive function based on a general recursive function $f_0 : \mathsf{A} \to \mathsf{B}$ is to formulate a new function $f_1 : \mathsf{A} \to \mathsf{Nat} \to \mathsf{B}$ where all of its recursive calls are done with *smaller* values on its second argument [1].

Notation. In the latter definitions and theorems, we use a common notation in type theory, pattern-matching [12] to define functions by cases on the inductive definition for the type of the arguments.

Example 1. Let us consider the following example to show the bounded recursion technique for defining the `uh` function. This function is used for reconstructing a `Metis` inference rule in Section 4.2.1.

$$\begin{array}{l} \mathsf{uh}_0 : \mathsf{Prop} \to \mathsf{Prop} \\ \mathsf{uh}_0\ (\varphi_1 \supset (\varphi_2 \supset \varphi_3)) = \mathsf{uh}_0\ ((\varphi_1 \wedge \varphi_2) \supset \varphi_3) \\ \mathsf{uh}_0\ (\varphi_1 \supset (\varphi_2 \wedge \varphi_3)) = \mathsf{uh}_0\ (\varphi_1 \supset \varphi_2) \wedge \mathsf{uh}_0\ (\varphi_1 \supset \varphi_3) \\ \mathsf{uh}_0\ \varphi \qquad\qquad\qquad\quad = \varphi. \end{array} \tag{1}$$

In (1), the first two equations in the uh_0 function definition are not structurally recursive. As a consequence, we could not use this function in `Agda` which would report a non-terminating error. This happens because the formula used in the recursive calls are not subformulas of the input formula in the function argument. Therefore, one way to define uh_0 in type theory is to define a new function using a bounded recursion.

$$\begin{array}{l} \mathsf{uh}_1 : \mathsf{Prop} \to \mathsf{Nat} \to \mathsf{Prop} \\ \mathsf{uh}_1\ (\varphi_1 \supset (\varphi_2 \supset \varphi_3))\ (\mathsf{succ}\ n) = \mathsf{uh}_1\ ((\varphi_1 \wedge \varphi_2) \supset \varphi_3)\ n \\ \mathsf{uh}_1\ (\varphi_1 \supset (\varphi_2 \wedge \varphi_3))\ (\mathsf{succ}\ n) = \mathsf{uh}_1\ (\varphi_1 \supset \varphi_2)\ n \wedge \mathsf{uh}_1\ (\varphi_1 \supset \varphi_3)\ n \\ \mathsf{uh}_1\ \varphi \qquad\qquad\qquad\quad n \qquad\quad = \varphi. \end{array}$$

We bounded the recursion calls of the uh_0 function using its *complexity measure*. We refer to the complexity measure of a function as the number of steps the function takes to finish. Since the uh_0 function is recursive, we can define its complexity measure by defining a recursive function instead of a closed formula for such a number. However, this function must to be structurally recursive as well, and it can be defined by following the pattern-matching cases of its definition. Thus, the complexity measure of the uh_0 function can be defined as follows.

$$\begin{array}{l} \mathsf{uh}_{cm} : \mathsf{Prop} \to \mathsf{Nat} \\ \mathsf{uh}_{cm}\ (\varphi_1 \supset (\varphi_2 \supset \varphi_3)) = \mathsf{uh}_{cm}\ \varphi_3 + 2 \\ \mathsf{uh}_{cm}\ (\varphi_1 \supset (\varphi_2 \wedge \varphi_3)) = \mathsf{max}\ (\mathsf{uh}_{cm}\ \varphi_2)\ (\mathsf{uh}_{cm}\ \varphi_3) + 1 \\ \mathsf{uh}_{cm}\ \varphi \qquad\qquad\qquad\quad = 0. \end{array}$$

Finally, we are able to define the `uh` function which is the structural recursive definition of the function uh_0 also suitable to use in `Agda`.

$$\begin{array}{l} \mathsf{uh}_{cm} : \mathsf{Prop} \to \mathsf{Prop} \\ \mathsf{uh}\ \varphi\ = \mathsf{uh}_1\ \varphi\ (\mathsf{uh}_{cm}\ \varphi). \end{array}$$

2.2 Intuitionistic logic

Type theory is foundation for constructive mathematics, and its internal language is the intuitionistic logic. Proving propositions in this theory demands a witness construction for the proof. Traditionally, for example in classical logic, a witness is not always needed as we can prove arguments by contraction, named *reductio ad absurdum*.

To reconstruct proofs generated by `Metis`, we have formalized in type theory the classical propositional logic in [34]. A briefly description of it is presented in [13]. In this formalization, we have to assume the principle of excluded middle (henceforth PEM) as an axiom since `Metis` is a prover for classical logic. Assuming PEM, we can justify refutation proofs by deriving from it the *reductio ad absurdum* rule (henceforth RAA). The RAA rule is the formulation of the principle of proof by contradiction, that is, a derivation of a contradiction, $\bot$, from the hypothesis $\neg\ \varphi$, is a derivation of φ.

Notation. The List type is the usual inductive and parametric type for lists.

We formalize the syntactical consequence relation of CPL by an inductive family _⊢_ with two indexes, a set of propositions (the premises) and a proposition (the conclusion), that is, $\Gamma \vdash \varphi$ represents that there is derivation with conclusion $\varphi : \mathsf{Prop}$ from the set of premises $\Gamma : \mathsf{List\ Prop}$. We implemented in [34] the syntactical consequence relation in a similar way as it was presented in [11]. For that reason, we have included in Fig. 1 the valid inference rules: structural rules like *weaken*, formation and elimination rules of the logical connectives and to reason classically, we add the PEM axiom.

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{assume} \qquad \frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ \text{weaken} \qquad \frac{}{\Gamma \vdash \top}\ \top\text{-intro} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot\text{-elim}$$

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\ \varphi}\ \neg\text{-intro} \qquad \frac{\Gamma \vdash \neg\ \varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ \neg\text{-elim} \qquad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge\text{-intro}$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ \wedge\text{-proj}_1 \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ \wedge\text{-proj}_2 \qquad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \vee\text{-intro}_1$$

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \vee\text{-intro}_2 \qquad \frac{\Gamma, \varphi \vdash \gamma \qquad \Gamma, \psi \vdash \gamma}{\Gamma, \varphi \vee \psi \vdash \gamma}\ \vee\text{-elim} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \supset \psi}\ \supset\text{-intro}$$

$$\frac{\Gamma \vdash \varphi \supset \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ \supset\text{-elim} \qquad \frac{}{\Gamma \vdash \varphi \vee \neg\ \varphi}\ \text{PEM}$$

Fig. 1. Inference rules for classical propositional logic.

3 Metis Prover

`Metis` is an automatic theorem prover for first-order logic with equality developed by Hurd [20]. This prover is suitable for proof-reconstruction since it provides well-documented proofs to justify its deduction steps from the basis of only six inference rules for first-order logic (see, for example, [32, 16]). For the propositional fragment, `Metis` has three inference rules, see Fig. 2.

$$\frac{}{\Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n}\ \text{axiom}\ \varphi_1, \cdots, \varphi_n \qquad \frac{}{\Gamma \vdash \varphi \vee \neg\, \varphi}\ \text{assume}\ \varphi$$

$$\frac{\Gamma \vdash \ell \vee \varphi \qquad \Gamma \vdash \neg\, \ell \vee \psi}{\Gamma \vdash \varphi \vee \psi}\ \text{resolve}\ \ell$$

Fig. 2. `Metis` inference rules for propositional logic.

3.1 Input language

The `TPTP` language is the input language to encode problems used by `Metis`. It includes the first-order form (denoted by `fof`) and clause normal form (denoted by `cnf`) formats [39]. The `TPTP` syntax[6] is a well-defined grammar to handle annotated formulas with the form:

```
language(name, role, formula).
```

where the `language` can be `fof` or `cnf` and the `name` serves to identify the formula within the problem. Each annotated formula also assumes a `role`. Some roles frequently used are `axiom`, `conjecture`, `definition`, `plain` and `hypothesis`. The reader can find the complete list in [39].

In the last field of the annotated formula we found the `formula`, a well-written expression using the the constants `$true` and `$false`, the negation unary operator (`~`), and the binary connectives (`&`, `|`, `=>`) that represent ($\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\supset$) respectively. To refer us to propositional variables, we can use words composing by letters and numbers.

3.2 Output language

The `TSTP` language is an output language for derivations used by `Metis` and other ATPs [41]. A `TSTP` derivation generated by `Metis` encodes a natural deduction proof using as valid deduction steps the inference rules in Fig. 1. These derivations are directed acyclic graphs, trees of refutations. Each node stands for an application of an inference rule to the formulas in the parent nodes. The leaves in the tree represent formulas in the given `TPTP` input. Each node is labeled with a name of its inference rule (e. g., `canonicalize`). Each edge in the tree links a premise with one conclusion.

Since `Metis` attempts to prove conjectures by refutation, and the root is the final derived formula in these proof trees, we have at their roots the conclusion $\bot$. Such derivations are a list of annotated formulas with the form:

```
language(name, role, formula, source [,useful info]).
```

The new field regarding the `TPTP` syntax is the `source` field that mainly intends to record the inference rule of the deduction step. This field has the pattern

```
inference(rule, useful info, parents).
```

where the `rule` is the inference name, the next field includes supporting arguments to apply the reasoning step, and the last field is a list of the parents nodes.

[6] See the complete syntax grammar at http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html.

For increasing the readability of this paper, we are not presenting these TSTP derivations using the above description. Instead of it, we have adopted a customized TSTP syntax to keep as short as possible the Metis derivations. For instance, we shortened some keywords, removed unnecessary arguments, and replaced some symbols. The complete description of these customizations are showed in Appendix A.

Example 2. We show in Fig. 3 the customized version of the Metis derivation of the theorem (2). This derivation will be used in Section 4 to present the proof-reconstruction process.

$$(p \Rightarrow q) \wedge (q \Rightarrow p) \vdash (p \vee q) \Rightarrow (p \wedge q) \tag{2}$$

```
1  fof(premise, axiom, (p ⊃ q) ∧ (q ⊃ p)).
2  fof(goal, conjecture, (p ∨ q) ⊃ (p ∧ q)).
3  fof(s0, (p ∨ q) ⊃ p, inf(strip, goal)).
4  fof(s1, ((p ∨ q) ∧ p) ⊃ q, inf(strip, goal)).
5  fof(neg0, ¬ ((p ∨ q) ⊃ p), inf(negate, s0)).
6  fof(n00, (¬ p ∨ q) ∧ (¬ q ∨ p), inf(canonicalize, premise)).
7  fof(n01, ¬ q ∨ p, inf(conjunct, n00)).
8  fof(n02, ¬ p ∧ (p ∨ q), inf(canonicalize, neg0)).
9  fof(n03, p ∨ q, inf(conjunct, n02)).
10 fof(n04, ¬ p, inf(conjunct, n02)).
11 fof(n05, q, inf(simplify, [n03, n04])).
12 cnf(r00, ¬ q ∨ p, inf(canonicalize, n01)).
13 cnf(r01, q, inf(canonicalize, n05)).
14 cnf(r02, p, inf(resolve, q, [r01, r00])).
15 cnf(r03, ¬ p, inf(canonicalize, n04)).
16 cnf(r04, ⊥, inf(resolve, p, [r02, r03])).
17 fof(neg1, ¬ ((p ∨ q) ∧ p) ⊃ q, inf(negate, s1)).
18 fof(n10, ¬ q ∧ p ∧ (p ∨ q), inf(canonicalize, neg1)).
19 fof(n11, (¬ p ∨ q) ∧ (¬ q ∨ p), inf(canonicalize, premise)).
20 fof(n12, ¬ p ∨ q, inf(conjunct, n11)).
21 fof(n13, ⊥, inf(simplify,[n10, n12])).
22 cnf(r10, ⊥, inf(canonicalize, n13)).
```

Fig. 3. Customized Metis TSTP derivation of the theorem in (2).

3.3 Inference rules

We present the list of inference rules used by Metis in the TSTP derivations for propositional logic in Table 1. We reconstruct these rules in Section 4 following the same order of this table.

In this table, strip is the first inference rule since is the first one in appearing in the TSTP derivations. The other rules are sorted by a mix between their level of complexity of their main purpose and their dependency with the reconstruction of other rules. For instance, the simplify rule and the clausify rule need theorems developed for the reconstruction of the canonicalize rule.

The `canonicalize` rule depends on the `resolve` reconstruction which depends on the reconstruction of the `conjunct` rule.

An alert reader could also notice that the inference rules presented in Fig. 2 diverge from the rules in aforementioned table. The former rules are implemented by the latter rules in the `TSTP` derivations. For instance, as far as we know, in `TSTP` derivations, the *axiom* rule is implemented by the rules `canonicalize`, `clausify`, `conjunct`, and `simplify`.

Table 1. `Metis` inference rules.

Rule	Purpose	Theorem number
`strip`	Strip a goal into subgoals	7
`conjunct`	Take a formula from a conjunction	10
`canonicalize`	Normalize a formula	21
`clausify`	Perform clausification	23
`simplify`	Simplify definitions and theorems	26
`resolve`	Apply one form of the resolution theorem	28

4 Proof-Reconstruction

In this section, we describe our approach to reconstruct proofs from CPL derivations generated by `Metis`. This reconstruction consists of a translation from a source system to the target system. The system of origin is the automatic theorem prover `Metis` and the target system is the proof-assistant `Agda`. We choose `Agda` but another proof-assistant with the same support of type theory and inductive types could be used (e. g. `Coq`).

4.1 Workflow

The overview of the proof-reconstruction is presented as a workflow in Fig. 4 following the same basis of the proposed workflow presented in [38].

The process begins assuming that the user posses a `TPTP` valid file that encodes a CPL conjecture. If this conjecture is a theorem, by means of `Metis`, we obtain a derivation in `TSTP` format of a proof for this conjecture. Now, we use the `Athena` tool to generate the `Agda` file that corresponds to the proof-term for such `TSTP` derivation. `Athena` parses and prunes the proof-tree of the `TSTP` derivation by removing some redundancies and steps that do not contribute to prove the conjecture. When the translation process finishes, we end up with a proof-term in an `Agda` file with the natural deduction proof. In this file, we make calls to theorems and lemmas in the `Agda` libraries, `agda-prop` (classical propositional logic) and `agda-metis` (propositional reasoning for `Metis` derivations).

Using `Agda` as the proof-checker, we type-check the proof-terms of the `Agda` file previous generated. If this type-checking success, the `TSTP` derivation generated by `Metis` will be *correct* module

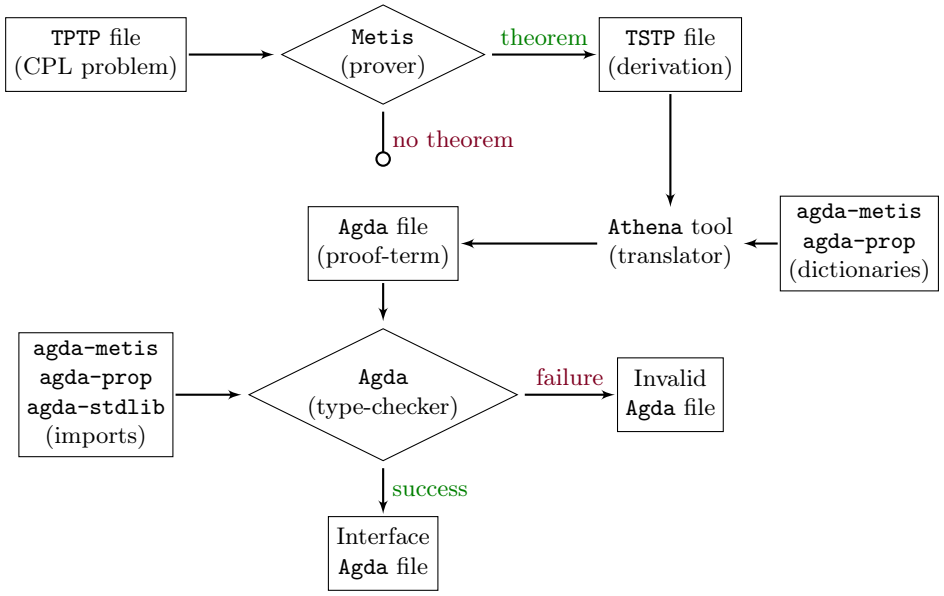

Fig. 4. Proof-reconstruction overview. The rectangles nodes represent text files. The rhombus nodes represent a process with two possible outcomes.

the `Agda` correctness and the proposed formalisation. In such a case, `Agda` will output an interface file. Otherwise, `Agda` will report where the type-checking fails, if so, the failure must be investigated by the user. The possible causes can be due to many factors: *completeness* issues in our `Agda` libraries for the `Metis` reasoning, bugs in the translation by `Athena`, bugs in `Agda`, printing errors of the `TSTP` derivation by `Metis` or soundness bugs in the `Metis` reasoning.

In the remainder part of this section, we present a formal description of definitions and theorems necessary to reconstruct `Metis` inference rules. Through this description, we refer to the lines in the `Metis` derivation presented in Fig. 3 of the theorem in (2) to show examples of each rule.

4.2 Reconstructing `Metis` inference rules

In this section, we reconstruct each `Metis` inference rule from Table 1. We begin describing the reconstruction of the `strip` inference rule since this rule is the only one that closely follows the `Metis` source code. To reconstruct the other rules, we follow a pattern showed in Example 8.

Notation. The function name written in `typewriter` font refers to a `Metis` inference rule in the `TSTP` derivations. The same function name written using sans serif font refers to our formalized version to reconstruct the rule and formally implemented in [36]. To increase the readability for functions and theorems, we use the convention Premise and Conclusion as synonyms of the Prop type. The Premise type in the reconstructed rule definition refers to the argument in the `TSTP` derivation, i. e. the input formula of the inference rule. A formula of Conclusion type refers to the expected result of the rule found in the `TSTP` derivation.

4.2.1 Strip. To prove a goal, `Metis` splits the goal into disjoint cases. This process produces a list of new subgoals, the conjunction of these subgoals implies the goal. Then, a proof of the goal becomes into a set of smaller proofs, one refutation for each subgoal.

Example 3. In Fig. 3, we can see how the subgoals associated to a goal are introduced in the `TSTP` derivation line 3 with the `strip` inference rule. The conjecture in line 2, $(p \vee q) \supset (p \wedge q)$, is stripped into two subgoals (line 4 and 5): $(p \vee q) \supset p$ and $((p \vee q) \wedge p) \supset q$. `Metis` proves each subgoal separately following the same order as they appeared in the `TSTP` derivation.

Remark 1. `Metis` does not make explicit in the `TSTP` derivation the way it uses the subgoals to prove the goal. We show this approach in the correctness of the `strip` inference rule in Theorem 7. To show this theorem, we prove Lemma 4 and Lemma 5. In the former, we introduce the auxiliary function uh_1 to define the uh function used in in the latter in the definition of the strip function.

Notation. Bound is a synonym for Nat type to refer to a natural number that constrains the number of recursive calls done by a function. We use this type mostly in the type of the function.

Lemma 4. Let be $n : \mathsf{Bound}$. If $\Gamma \vdash \mathsf{uh}_1\ \varphi\ n$ then $\Gamma \vdash \varphi$ where

$$\begin{array}{l} \mathsf{uh}_1 : \mathsf{Prop} \to \mathsf{Bound} \to \mathsf{Prop} \\ \begin{array}{lll} \mathsf{uh}_1\ (\varphi_1 \supset (\varphi_2 \supset \varphi_3))\ (\mathsf{succ}\ n) & = & \mathsf{uh}_1\ ((\varphi_1 \wedge \varphi_2) \supset \varphi_3)\ n \\ \mathsf{uh}_1\ (\varphi_1 \supset (\varphi_2 \wedge \varphi_3))\ (\mathsf{succ}\ n) & = & \mathsf{uh}_1\ (\varphi_1 \supset \varphi_2)\ n \wedge \mathsf{uh}_1\ (\varphi_1 \supset \varphi_3)\ n \\ \mathsf{uh}_1\ \varphi\ n & = & \varphi. \end{array} \end{array} \tag{3}$$

To provide such a number given a formula φ, we call the uh_{cm} function defined in (4). For the convenience in the following descriptions, we have the function $\mathsf{uh} : \mathsf{Prop} \to \mathsf{Prop}$ for $\varphi \mapsto \mathsf{uh}_1\ \varphi\ (\mathsf{uh}_{cm}\ \varphi)$.

$$\begin{array}{l} \mathsf{uh}_{cm} : \mathsf{Prop} \to \mathsf{Bound} \\ \begin{array}{ll} \mathsf{uh}_{cm}\ (\varphi_1 \supset (\varphi_2 \supset \varphi_3)) & = \mathsf{uh}_{cm}\ \varphi_3 + 2 \\ \mathsf{uh}_{cm}\ (\varphi_1 \supset (\varphi_2 \wedge \varphi_3)) & = \mathsf{max}\ (\mathsf{uh}_{cm}\ \varphi_2)\ (\mathsf{uh}_{cm}\ \varphi_3) + 1 \\ \mathsf{uh}_{cm}\ \varphi & = 0. \end{array} \end{array} \tag{4}$$

Now, we define the strip function in (5) in a similar way as we did above for the uh function. We have a definition of the strip_1 function in (6) and a function to find its bound for its recursive calls by strip_{cm} defined in [37].

$$\begin{array}{l} \mathsf{strip} : \mathsf{Prop} \to \mathsf{Prop} \\ \mathsf{strip}\ \varphi\ = \mathsf{strip}_1\ \varphi\ (\mathsf{strip}_{cm}\ \varphi). \end{array} \tag{5}$$

The strip function yields the conjunction of subgoals that implies the goal of the problem in the `Metis` `TSTP` derivations. Its definition comes mainly from the reading of the `Metis` source code and some examples from `TSTP` derivations.

$$
\begin{array}{l}
\mathsf{strip}_1 : \mathsf{Prop} \to \mathsf{Bound} \to \mathsf{Prop} \\
\begin{array}{lll}
\mathsf{strip}_1\ (\varphi_1 \wedge \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\mathsf{strip}_1\ \varphi_1\ n) \wedge \mathsf{uh}\ (\varphi_1 \supset \mathsf{strip}_1\ \varphi_2\ n) \\
\mathsf{strip}_1\ (\varphi_1 \vee \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{uh}\ ((\neg\ \varphi_1) \supset \mathsf{strip}_1\ \varphi_2\ n) \\
\mathsf{strip}_1\ (\varphi_1 \supset \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\varphi_1 \supset \mathsf{strip}_1\ \varphi_2\ n) \\
\mathsf{strip}_1\ (\neg\ (\varphi_1 \wedge \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\varphi_1 \supset \mathsf{strip}_1\ (\neg\ \varphi_2)\ n) \\
\mathsf{strip}_1\ (\neg\ (\varphi_1 \vee \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\mathsf{strip}_1\ (\neg\ \varphi_1)\ n) \wedge \mathsf{uh}\ ((\neg\ \varphi_1) \supset \mathsf{strip}_1\ (\neg\ \varphi_2)\ n) \\
\mathsf{strip}_1\ (\neg\ (\varphi_1 \supset \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\mathsf{strip}_1\ \varphi_1\ n) \wedge \mathsf{uh}\ (\varphi_1 \supset \mathsf{strip}_1\ (\neg\ \varphi_2)\ n) \\
\mathsf{strip}_1\ (\neg\ (\neg\ \varphi_1)) & (\mathsf{succ}\ n) & = \mathsf{uh}\ (\mathsf{strip}_1\ \varphi_1\ n) \\
\mathsf{strip}_1\ (\neg\ \bot) & (\mathsf{succ}\ n) & = \top \\
\mathsf{strip}_1\ (\neg\ \top) & (\mathsf{succ}\ n) & = \bot \\
\mathsf{strip}_1\ \varphi & n & = \varphi.
\end{array}
\end{array}
\tag{6}
$$

Lemma 5. Let $n : \mathsf{Bound}$. If $\Gamma\ \vdash\ \mathsf{strip}_1\ \varphi\ n$ then $\Gamma\ \vdash\ \varphi$.

The following lemma is convenient to substitute in a theorem equals by equals in the conclusion of the sequent. Recall the equality ($\equiv$) is symmetric and transitive, and we use these properties without any mention.

Lemma 6 (subst)**.** If $\Gamma \vdash \varphi$ and $\psi \equiv \varphi$ then $\Gamma \vdash \psi$.

We can now formulate the result that justifies the stripping strategy of `Metis` to prove goals. For the sake of brevity, we state the following theorem for the `strip` function when the goal has two subgoals. In other cases, we extend the theorem in the natural way.

Theorem 7. Let s_1 and s_2 be the subgoals of the goal φ, that is, $\mathsf{strip}\ \varphi \equiv\ s_1 \wedge\ s_2$. If $\Gamma \vdash s_1$ and $\Gamma \vdash s_2$ then $\Gamma \vdash \varphi$.

Remark 2. The user of `Metis` will probably note another rule in the derivations related with these subgoals, the `negate` rule. Since `Metis` proves a conjecture by refutation, to prove a subgoal, `Metis` assumes its negation using the `negate` rule. We will find this rule always after the `strip` rule per each subgoal in the `TSTP` derivation.

The remainder of this section will be devoted to present a formal description in type theory of the remaining `Metis` rules presented in Table 1 using the pattern presented in Example 8. We follow the same order to present the rules as the table shows.

Example 8. Let `metisRule` be a `Metis` inference rule in the `TSTP` derivations. To reconstruct this `metisRule`, we define in the type theory a function preserving the name of the original rule. This function should follow the following pattern.

$$
\begin{array}{l}
\mathsf{metisRule} : \mathsf{Premise} \to \mathsf{Conclusion} \to \mathsf{Prop} \\
\mathsf{metisRule}\ \varphi\ \psi = \begin{cases} \psi, & \text{if the conclusion } \psi \text{ can be derived by applying certain inference} \\ & \text{rules to the premise } \varphi; \\ \varphi, & \text{otherwise.} \end{cases}
\end{array}
$$

To justify that all transformations performed by the metisRule rule are correct, we will provide a theorem to show its soundness that says:

$$\text{If } \Gamma \vdash \varphi \text{ then } \Gamma \vdash \mathsf{metisRule}\ \varphi\ \psi.$$

4.2.2 Conjunct. The `conjunct` rule extracts from a conjunction one of its conjuncts. This rule is a generalization of the projection rules for the conjunction connective. See in as the following `TSTP` excerpt shows.

Example 9. In Fig. 3 line 20, the `conjunct` rule acts as a left projection ($\wedge$-proj$_1$) taking $\neg\, p \vee q$ from $(\neg\, p \vee q) \wedge (\neg\, q \vee p)$ in line 19.

Theorem 10. Let ψ : Conclusion. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{conjunct}\ \varphi\ \psi$, where

$$\mathsf{conjunct} : \mathsf{Premise} \to \mathsf{Conclusion} \to \mathsf{Prop}$$

$$\mathsf{conjunct}\ \varphi\ \psi = \begin{cases} \psi, & \text{if } \varphi \equiv \psi; \\ \psi, & \text{if } \varphi \equiv \varphi_1 \wedge \varphi_2 \text{ and } \psi \equiv \mathsf{conjunct}\ \varphi_1\ \psi; \\ \psi, & \text{if } \varphi \equiv \varphi_1 \wedge \varphi_2 \text{ and } \psi \equiv \mathsf{conjunct}\ \varphi_2\ \psi; \\ \varphi, & \text{otherwise.} \end{cases}$$

Before continuing with the remaining rules, let us mention the normal forms we are using for propositions and one comment about their pertinence in the reconstruction process.

Definition 11. A *literal* is a propositional variable (positive literal) or a negation of a propositional variable (negative literal).

Notation. Lit is a synonym of Prop to refer to literals.

Definition 12. The *negative normal form* of a formula is the logical equivalent version of it in which negations appear only in the literals and the formula does not contain any implications.

Definition 13. The *conjunctive normal form* of a formula also called clausal normal form is the logical equivalent version expressed as a conjunction of clauses where a *clause* is the disjunction of zero or more literals.

Remark 3. The proof-reconstruction we are exposing needs a mechanism to test logic equivalence between propositions in some cases. This task is not trivial since we left out the semantics to treat only the syntax aspects of the propositional formulas, and therefore we have to manipulate in somehow the formulas to see if they are logically equivalent.

We address this challenge by converting the formulas to their conjunctive normal. We check if their normal forms are equal and if they are not, we try to reorder the first formula to match with the second one to fulfill the syntactical equality. In [37], we make a description of a set of functions and lemmas for such a purpose. For instance, we define functions of (Premise $\to$ Conclusion $\to$ Prop) type following the pattern in Example 8 like the $\mathsf{reorder}_\vee$ function. This function try to fix the order of a *premise* disjunction given by following the order of the *conclusion* disjunction. Similarly, we define the function $\mathsf{reorder}_\wedge$ for conjunctions and the function $\mathsf{reorder}_{\wedge\vee}$ for conjunctive normal forms.

4.2.3 Canonicalize. The `canonicalize` rule is an inference that transforms a formula to its negative normal form or its conjunctive normal form depending on the role that the formula plays in the problem. This rule jointly with the `clausify` rule perform the so-called *clausification* process mainly described in [40].

$$\text{Disjunction:}\quad \frac{\Gamma \vdash \varphi \vee \bot}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \vee \top}{\Gamma \vdash \top} \qquad \frac{\Gamma \vdash \varphi \vee \neg\, \varphi}{\Gamma \vdash \top} \qquad \frac{\Gamma \vdash \varphi \vee \varphi}{\Gamma \vdash \varphi}$$

$$\text{Conjunction:}\quad \frac{\Gamma \vdash \varphi \wedge \bot}{\Gamma \vdash \bot} \qquad \frac{\Gamma \vdash \varphi \wedge \top}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \wedge \varphi}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \wedge \neg\, \varphi}{\Gamma \vdash \bot}$$

Fig. 5. Theorems to remove inside of a formula by the `canonicalize` rule.

The `canonicalize` rule is mainly used by `Metis` to introduce the negation of a subgoal in its refutation proof. When the input formula is an axiom, definition or hypothesis introduced in the `TPTP` file of the problem, the `canonicalize` rule returns its negative normal form. In the remaining cases, as far as we know, `Metis` uses the `canonicalize` rule to perform a sort of simplifications to remove the tautologies listed in Fig. 5 from the conjunctive normal form of the input formula.

To reconstruct this rule, we guide its prensentation as follows. In Lemma 14 and Lemma 15, we show how to simplify in disjunctions and conjunctions, respectively. The negative normal form is stated in Lemma 16 and the conjunctive normal form in Lemma 18. Finally, the reconstruction of the `canonicalize` rule is stated in Theorem 21.

Notation. In a disjunction, $\varphi \equiv \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$, we say $\psi \in_\vee \varphi$, if there is some $i = 1, \cdots, n$ such that $\psi \equiv \varphi_i$. Note that $\psi \in_\vee \varphi$ is another representation for the equality $\psi \equiv \mathsf{reorder}_\vee\ \varphi\ \psi$. We assume all disjunctions to be right-associative unless otherwise stated.

Lemma 14. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{simplify}_\vee\ \varphi$, where

$$\begin{aligned}
&\mathsf{simplify}_\vee : \mathsf{Prop} \to \mathsf{Prop}\\
&\mathsf{simplify}_\vee\ (\bot \vee \varphi) = \mathsf{simplify}_\vee\ \varphi\\
&\mathsf{simplify}_\vee\ (\varphi \vee \bot) = \mathsf{simplify}_\vee\ \varphi\\
&\mathsf{simplify}_\vee\ (\top \vee \varphi) = \top\\
&\mathsf{simplify}_\vee\ (\varphi \vee \top) = \top\\
&\mathsf{simplify}_\vee\ (\varphi_1 \vee \varphi_2) = \begin{cases} \top, & \text{if } \varphi_1 \equiv \neg\, \psi \text{ for some } \psi : \mathsf{Prop} \text{ and } \psi \in_\vee \varphi_2;\\ \top, & \text{if } (\neg\, \varphi_1) \in_\vee \varphi_2;\\ \mathsf{simplify}_\vee\ \varphi_2, & \text{if } \varphi_1 \in_\vee \varphi_2;\\ \top, & \text{if } \mathsf{simplify}_\vee\ \varphi_2 \equiv \top;\\ \varphi_1, & \text{if } \mathsf{simplify}_\vee\ \varphi_2 \equiv \bot;\\ \varphi_1 \vee \mathsf{simplify}_\vee\ \varphi_2, & \text{otherwise,} \end{cases}\\
&\mathsf{simplify}_\vee\ \varphi = \varphi.
\end{aligned} \tag{7}$$

Now, we have removed tautologies in disjunctions by applying the $\mathsf{simplify}_\vee$ function, we formulate in a similar way, the $\mathsf{simplify}_\wedge$ function to simplify tautologies about conjunctions.

Notation. In a conjunction, $\varphi \equiv \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$, we say $\psi \in_\wedge \varphi$, if there is some $i = 1, \cdots, n$ such that $\psi \equiv \varphi_i$. Note that $\psi \in_\wedge \varphi$ is another representation of the equality $\psi \equiv \mathsf{conjunct}\ \varphi\ \psi$. We assume all conjunctions to be right-associative unless otherwise stated.

Lemma 15. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{simplify}_\wedge\ \varphi$, where

$$
\begin{array}{l}
\mathsf{simplify}_\wedge : \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{simplify}_\wedge\ (\bot \wedge \varphi) = \bot \\
\mathsf{simplify}_\wedge\ (\varphi \wedge \bot) = \bot \\
\mathsf{simplify}_\wedge\ (\top \wedge \varphi) = \mathsf{simplify}_\wedge\ \varphi \\
\mathsf{simplify}_\wedge\ (\varphi \wedge \top) = \mathsf{simplify}_\wedge\ \varphi \\
\mathsf{simplify}_\wedge\ (\varphi_1 \wedge \varphi_2) = \begin{cases}
\bot, & \text{if } \varphi_1 \equiv \neg\,\psi \text{ for some } \psi : \mathsf{Prop} \text{ and } \psi \in_\wedge \varphi_2; \\
\bot, & \text{if } (\neg\,\varphi_1) \in_\wedge \varphi_2; \\
\mathsf{simplify}_\wedge\ \varphi_2, & \text{if } \varphi_1 \in_\wedge \varphi_2; \\
\varphi_1, & \text{if } \mathsf{simplify}_\wedge\ \varphi_2 \equiv \top; \\
\bot, & \text{if } \mathsf{simplify}_\wedge\ \varphi_2 \equiv \bot; \\
\varphi_1 \wedge \mathsf{simplify}_\wedge\ \varphi_2, & \text{otherwise,}
\end{cases} \\
\mathsf{simplify}_\wedge\ \varphi = \varphi.
\end{array}
\tag{8}
$$

Using the functions $\mathsf{simplify}_\vee$ and $\mathsf{simplify}_\wedge$ we formulate the functions that finds the normal form based on the `Metis` source code used to *normalize* formulas. We use bounded recursion to define the function nnf_1 in (9).

$$
\begin{array}{lll}
\mathsf{nnf}_1 : \mathsf{Prop} \to \mathsf{Bound} \to \mathsf{Prop} & & \\
\mathsf{nnf}_1\ (\varphi_1 \wedge \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\wedge\ (\mathsf{assoc}_\wedge\ (\mathsf{nnf}_1\ \varphi_1\ n \wedge \mathsf{nnf}_1\ \varphi_2\ n)) \\
\mathsf{nnf}_1\ (\varphi_1 \vee \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\vee\ (\mathsf{assoc}_\vee\ (\mathsf{nnf}_1\ \varphi_1\ n \vee \mathsf{nnf}_1\ \varphi_2\ n)) \\
\mathsf{nnf}_1\ (\varphi_1 \supset \varphi_2) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\vee\ (\mathsf{assoc}_\vee\ (\mathsf{nnf}_1\ ((\neg\,\varphi_1) \vee \varphi_2)\ n)) \\
\mathsf{nnf}_1\ (\neg\,(\varphi_1 \wedge \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\vee\ (\mathsf{assoc}_\vee\ (\mathsf{nnf}_1\ ((\neg\,\varphi_1) \vee (\neg\,\varphi_2))\ n)) \\
\mathsf{nnf}_1\ (\neg\,(\varphi_1 \vee \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\wedge\ (\mathsf{assoc}_\wedge\ (\mathsf{nnf}_1\ ((\neg\,\varphi_1) \wedge (\neg\,\varphi_2))\ n)) \\
\mathsf{nnf}_1\ (\neg\,(\varphi_1 \supset \varphi_2)) & (\mathsf{succ}\ n) & = \mathsf{simplify}_\wedge\ (\mathsf{assoc}_\wedge\ (\mathsf{nnf}_1\ ((\neg\,\varphi_2) \wedge \varphi_1)\ n)) \\
\mathsf{nnf}_1\ (\neg\,(\neg\,\varphi)) & (\mathsf{succ}\ n) & = \mathsf{nnf}_1\ \varphi_1\ n \\
\mathsf{nnf}_1\ (\neg\,\top) & (\mathsf{succ}\ n) & = \bot \\
\mathsf{nnf}_1\ (\neg\,\bot) & (\mathsf{succ}\ n) & = \top \\
\mathsf{nnf}_1\ \varphi & \mathsf{zero} & = \varphi.
\end{array}
\tag{9}
$$

The function nnf in (10) is defined by using the function nnf_{cm} encharges to find the bound for the recursive calls of the nnf_1 function. The interested reader can find the complete definition from Appendix in [37], we do not include it for the seek of brevity.

$$
\begin{array}{l}
\mathsf{nnf} : \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{nnf}\ \varphi = \mathsf{nnf}_1\ \varphi\ (\mathsf{nnf}_{cm}\varphi).
\end{array}
\tag{10}
$$

Lemma 16. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{nnf}\ \varphi$.

To get the conjunctive normal form, we make sure the formula is a conjunction of disjunctions. For such a purpose, we use distributive laws in Lemma 17 to get that form after applying the nnf function.

Lemma 17. $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{dist}\ \varphi$, where

$$\begin{aligned}&\mathsf{dist} : \mathsf{Prop} \to \mathsf{Prop}\\ &\mathsf{dist}\ (\varphi_1 \wedge \varphi_2) = \mathsf{dist}\ \varphi_1 \wedge \mathsf{dist}\ \varphi_2\\ &\mathsf{dist}\ (\varphi_1 \vee \varphi_2) = \mathsf{dist}_\vee\ (\mathsf{dist}\ \varphi_1 \vee \mathsf{dist}\ \varphi_2)\\ &\mathsf{dist}\ \varphi \qquad\quad = \varphi\end{aligned}$$

and

$$\begin{aligned}&\mathsf{dist}_\vee : \mathsf{Prop} \to \mathsf{Prop}\\ &\mathsf{dist}_\vee\ ((\varphi_1 \wedge \varphi_2) \vee \varphi_3) = \mathsf{dist}_\vee\ (\varphi_1 \vee \varphi_2) \wedge \mathsf{dist}_\vee\ (\varphi_2 \vee \varphi_3)\\ &\mathsf{dist}_\vee\ (\varphi_1 \vee (\varphi_2 \wedge \varphi_3)) = \mathsf{dist}_\vee\ (\varphi_1 \vee \varphi_2) \wedge \mathsf{dist}_\vee\ (\varphi_1 \vee \varphi_3)\\ &\mathsf{dist}_\vee\ \varphi \qquad\qquad\qquad = \varphi.\end{aligned}$$

We get the conjunctive normal form by applying the nnf function follow by the dist function.

Lemma 18. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{cnf}\ \varphi$, where

$$\begin{aligned}&\mathsf{cnf} : \mathsf{Prop} \to \mathsf{Prop}\\ &\mathsf{cnf}\ \varphi = \mathsf{dist}\ (\mathsf{nnf}\ \varphi).\end{aligned}$$

Since all the transformations in Lemma 16 and Lemma 17 came from logical equivalences in propositional logic, we state the following lemmas used in the reconstruction of the `simplify` rule in Lemma 24 and in Theorem 21 for the `canonicalize` rule.

Lemma 19. If $\Gamma \vdash \mathsf{nnf}\ \varphi$ then $\Gamma \vdash \varphi$.

Lemma 20. If $\Gamma \vdash \mathsf{cnf}\ \varphi$ then $\Gamma \vdash \varphi$.

Now, we are ready to reconstruct the canonicalize rule. This inference rule defined in (11) performs normalization for a proposition. That is, depending on the role of the formula in the problem, it converts the formula to its negative normal form or its conjunctive normal form. In both cases, the `canonicalize` rule simplifies the formula by removing tautologies inside of it.

Theorem 21. Let $\psi : \mathsf{Conclusion}$. If $\Gamma \vdash \varphi$ then $\Gamma \ \vdash\ \mathsf{canonicalize}\ \varphi\ \psi$, where

$$\begin{aligned}&\mathsf{canonicalize} : \mathsf{Premise} \to \mathsf{Conclusion} \to \mathsf{Prop}\\ &\mathsf{canonicalize}\ \varphi\ \psi = \begin{cases}\psi, & \text{if } \psi \equiv \varphi;\\ \psi, & \text{if } \psi \equiv \mathsf{nnf}\ \varphi;\\ \psi, & \text{if } \mathsf{cnf}\ \psi \equiv \mathsf{reorder}_{\wedge\vee}\ (\mathsf{cnf}\ \varphi)\ (\mathsf{cnf}\ \psi);\\ \varphi, & \text{otherwise.}\end{cases}\end{aligned} \tag{11}$$

4.2.4 Clausify. The `clausify` rule is a rule that transforms a formula into its clausal normal form but without performing simplifications of tautologies. Recall, this kind of conversion was already addressed by the `canonicalize` rule. It is important to notice that this kind of conversions between one formula to its clausal normal form are not unique, and `Metis` has customized approaches to perform such transformations.

To address the reconstruction of this rule, we perform a reordering of the conjunctive normal form given by the cnf function defined in Lemma 18 with the aforementioned function $\mathsf{reorder}_{\wedge\vee}$ to the input formula of the rule.

Example 22. In the following `TSTP` derivation by `Metis`, the `clausify` rule transforms the $\mathtt{n}_0$ formula to get the $\mathtt{n}_1$ formula, that is just the application of the distribution laws.

```
fof(n0, ¬ p ∨ (q ∧ r) ...
fof(n1, (¬ p ∨ q) ∧ (¬ p ∨ r), inf(clausify, n0)).
```

Theorem 23. Let ψ : Conclusion. If $\Gamma \vdash \varphi$ then $\Gamma \vdash \mathsf{clausify}\ \varphi\ \psi$, where

$$\mathsf{clausify} : \mathsf{Premise} \to \mathsf{Conclusion} \to \mathsf{Prop}$$
$$\mathsf{clausify}\ \varphi\ \psi = \begin{cases} \psi, & \text{if } \varphi \equiv \psi; \\ \mathsf{reorder}_{\wedge\vee}\ (\mathsf{cnf}\ \varphi)\ \psi, & \text{otherwise.} \end{cases}$$

4.2.5 Simplify. The `simplify` rule is an inference that performs simplification of tautologies in a list of derivations. This rule simplifies such a list by transversing the list while applying Lemma 14, Lemma 15, among others theorems presented in the following description. The goal of this rule despites simplifying the list of derivations to a new formula (often smaller than its input formulas), consists of finding a contradiction.

We observe based on the analysis of different cases in the `TSTP` derivations that `simplify` can be modeled by a function with three arguments: two source formulas and the target formula.

Since the main purpose of the `simplify` rule is to simplify formulas to get $\bot$, we have defined the reduce_ℓ function to help removing the negation of a given literal ℓ from a input formula.

$$\begin{aligned} &\mathsf{reduce}_\ell : \mathsf{Prop} \to \mathsf{Lit} \to \mathsf{Prop} \\ &\mathsf{reduce}_\ell\ (\varphi_1\ \wedge\ \varphi_2)\ \ell\ = \mathsf{simplify}_\wedge\ (\mathsf{reduce}_\ell\ \varphi_1\ \ell\ \wedge\ \mathsf{reduce}_\ell\ \varphi_2\ \ell) \\ &\mathsf{reduce}_\ell\ (\varphi_1\ \vee\ \varphi_2)\ \ell\ = \mathsf{simplify}_\vee\ (\mathsf{reduce}_\ell\ \varphi_1\ \ell\ \vee\ \mathsf{reduce}_\ell\ \varphi_2\ \ell) \\ &\mathsf{reduce}_\ell\ \varphi\qquad\qquad \ell\ = \begin{cases} \bot, & \text{if } \varphi \text{ is a literal and } \ell \equiv \mathsf{nnf}(\neg\ \varphi); \\ \varphi, & \text{otherwise.} \end{cases} \end{aligned} \tag{12}$$

Lemma 24. Let ℓ be a literal and φ : Prop. If $\Gamma \vdash \varphi$ and $\Gamma \vdash \ell$ then $\Gamma\ \vdash\ \mathsf{reduce}_\ell\ \varphi\ \ell$.

The simplify function is defined in (13). If some input formula is equal to the target formula, we derive that formula. If any formula is $\bot$, we derive the target formula by using the $\bot$-elim inference rule. Otherwise, we proceed by cases on the structure of the formula. For example, when the second source formula is a literal, we use the reduce_ℓ function and Lemma 24.

$$
\begin{array}{l}
\mathsf{simplify} : \mathsf{Premise} \to \mathsf{Premise} \to \mathsf{Conclusion} \to \mathsf{Prop} \\
\mathsf{simplify}\ \varphi_1\ \varphi_2\ \psi\ = \\
\quad \begin{cases}
\psi, & \text{if } \varphi_i \equiv \bot \text{ for some } i = 1, 2; \\
\psi, & \text{if } \varphi_i \equiv \psi \text{ for some } i = 1, 2; \\
\mathsf{simplify}_{\wedge}\ (\mathsf{simplify}\ \varphi_1\ \varphi_{21}\ \psi)\ \varphi_{22}\ \psi, & \text{if } \varphi_2 \equiv \varphi_{21} \wedge \varphi_{22}; \\
\mathsf{simplify}_{\vee}\ (\mathsf{simplify}\ \varphi_1\ \varphi_{21}\ \psi\ \vee\ \mathsf{simplify}\ \varphi_1\ \varphi_{22}\ \psi) & \text{if } \varphi_2 \equiv \varphi_{21} \vee \varphi_{22}; \\
\mathsf{reduce}_{\ell}\ \varphi_1\ \varphi_2, & \text{if } \varphi_2 \text{ is a literal}; \\
\varphi_1, & \text{otherwise.}
\end{cases}
\end{array}
\tag{13}
$$

Lemma 25. Let ψ : Conclusion. If $\Gamma \vdash \varphi_1$ and $\Gamma \vdash \varphi_2$ then $\Gamma \vdash \mathsf{simplify}\ \varphi_1\ \varphi_2\ \psi$.

Theorem 26. Let ψ : Conclusion. Let φ_i : Premise such that $\Gamma \vdash \varphi_i$ for $i = 1, \cdots, n$ and $n \geq 2$. Then $\Gamma \vdash \mathsf{simplify}\ \gamma_{n-1}\ \varphi_n\ \psi$ where $\gamma_1 = \varphi_1$, and $\gamma_i \equiv \mathsf{simplify}\ \gamma_{i-1}\ \varphi_i\ \psi$.

Remark. Besides the fact that List Prop → Prop is the type that mostly fit with the `simplify` rule, we choose a different option. In the translation from `TSTP` to `Agda`, we take the list of derivations and we apply the rule by using a left folding (the `foldl` function in functional programming) with the simplify function over the list of $\varphi_1, \varphi_2, \cdots, \varphi_n$. This design decision avoids us to define a new theorem type to support List Prop type in the premise side.

Example 27. Let us review the following `TSTP` excerpt where `simplify` was used twice.

```
fof(n0, (¬ p ∨ q) ∧ ¬ r ∧ ¬ q ∧ (p ∨ (¬ s ∨ r)), ...
fof(n1, p ∨ (¬ s ∨ r), inf(conjunct, n0)).
fof(n2, ¬ p ∨ q, inf(conjunct, n0)).
fof(n3, ¬ q, inf(conjunct, n0)).
fof(n4, ¬ p, inf(simplify, [n2, n3])).
fof(n5, ¬ r, inf(conjunct, n0)).
fof(n6, ⊥, inf(simplify, [n1, n4, n5])).
```

1. The `simplify` rule derives $\neg p$ in $\mathtt{n}_4$ from $\mathtt{n}_2$ and $\mathtt{n}_3$ derivations.

$$\mathsf{simplify}\ (\neg p \vee q)\ (\neg q)\ (\neg p) = \neg p.$$

2. We use Theorem 26 to derive $\bot$ in $\mathtt{n}_6$, the following is a proof.

$$\dfrac{\dfrac{\Gamma \vdash p \vee (\neg s \wedge r) \qquad \Gamma \vdash \neg p}{\Gamma \vdash \neg s \wedge r}\ \text{Theorem 26} \qquad \Gamma \vdash \neg r}{\Gamma \vdash \bot}\ \text{Theorem 26}$$

4.2.6 Resolve. The `resolve` inference rule in combination with the `simplify` rule are the responsable rules to perform resolution, in particular, the `resolve` rule is is the version of the resolution theorem showed in Fig. 2.

Example 4. In the `TSTP` derivation in Fig. 2, there are two ocurrences in line 14 and line 16 of the `resolve` rule. This rule has two inputs in the following order: the *resolvent* which is the literal denoted by ℓ in Fig. 2 and a list formed by two formulas to resolve.

In [37], we describe our first attempt to reconstruct this rule that performs reordering of formulas jointly with the application of a customized version of the resolution theorem to reconstruct the rule. Unfortunately, that approach suffers of an exponential computational cost for each searching to reorder the formulas. Therefore, we propose in (14) an alternative formulation to reconstruct this rule by using the simplify rule. This observation was taken from the `TSTP` derivations and we check that improves significantly the timing of the type-checking.

Theorem 28. Let ℓ be a literal, ℓ : Lit, and ψ : Conclusion. If $\Gamma \vdash \varphi_1$ and $\Gamma \vdash \varphi_2$ then $\Gamma \vdash$ resolve $\varphi_1\ \varphi_2\ \ell\ \psi$, where

$$\begin{aligned} &\mathsf{resolve} : \mathsf{Premise} \to \mathsf{Premise} \to \mathsf{Lit} \to \mathsf{Conclusion} \to \mathsf{Prop} \\ &\mathsf{resolve}\ \varphi_1\ \varphi_2\ \ell\ \psi\ =\ \mathsf{simplify}\ (\varphi_1 \wedge \varphi_2)\ (\neg\,\ell \vee \ell)\ \psi. \end{aligned} \tag{14}$$

The above theorem finishes the formalization of the `Metis` inference rules. At this point, we are able to justify step-by-step any proof in propositional logic generated by `Metis`. Actually, we tested successfully the translation by `Athena` jointly with the `Agda` formalizations of this section against more than eighty representative theorems in propositional logic. The interested reader can reproduce the testing of all problems from [33] with the sources of the `Athena` tool repository [35].

5 Related Work

Many approaches have been proposed for proof-reconstruction and some tools have been implemented in the last decades. We first mention some tools in type theory and later we listed some proof-reconstruction tools for classical logic.

Kanso in [24, 25] shows a proof-reconstruction in `Agda` using semantics for logic equivalences for derivations in propositional logic generated by `EProver` and `Z3`. In [18] Foster and Struth describe another proof-reconstruction in `Agda` but for pure equational logic of derivations generated by `Waldmeister` [19]. No other proof-reconstruction has been carried out neither in `Agda` nor with `Metis` prover as far as we know.

In `Coq` [43], other proof-assistant closes to `Agda`, we found the `SMTCoq` [2, 15] tool which provides a certified checker for proof witness coming from the SMT solver `veriT` [8] and adds a new tactic named verit, that calls `veriT` on any `Coq` goal. Bezem, Hendriks, and Nivelle in [4] transform a proof produced by the first-order automatic theorem prover `Bliksem` [28] in a `Coq` proof-term given a fixed but arbitrary first-order signature.

There are other successful attempts to reconstruct proofs from ATPs but using proof-assistants for classical logic instead of type theory. Let us mention some representative tools.

The `Isabelle` proof-assistant has the `Sledgehammer` tool. This program provides a full integration between automatic theorem provers [5, 17, 7] and `Isabelle/HOL` [27], the specialization of `Isabelle` for higher-order logic. A modular proof-reconstruction workflow is presented jointly with the full integration of `Leo-II` and `Satallax` provers with `Isabelle/HOL` in Eén and Sörensson [14].

Hurd [21] integrates the first-order resolution prover `Gandalf` with the high-order theorem prover `HOL` [30]. Its `GANDALF_TAC` tactic is able to reconstruct `Gandalf` proofs by using a LCF model. For

HOL Light, a version of HOL but with a simpler logic core, the SMT solver CVC4 was integrated. Kaliszyk and Urban [23] reconstruct proofs from different ATPs with the PRocH tool by replaying detailed inference steps from the ATPs with internal inference methods implemented in HOL Light.

6 Conclusions

We presented a proof-reconstruction approach in type theory for the propositional fragment of the Metis prover. In Section 4.2, we provided for each Metis inference rule a formal description in type theory following a syntactical approach type-checked in the proof-assistant Agda in [34, 36]. With this formalization, we were able to type-check Agda proof-terms for TSTP derivations generated by Metis. The Agda proof-terms were generated by the Athena translator [35], a tool written in Haskell.

Our approach to reconstruct the proofs for Metis TSTP derivations only used syntactical aspects of the logic. We chose that syntactical treatment instead of using semantics to extend this work towards the support of first-order logic or other non-classical logics. For instance, let us recall that for first-order logic, satisfiability is undecidable and its syntactical aspect plays an important role to reconstruct its proofs.

Lastly, we strongly believe that by justifying proofs by theorem provers, we help to increase their trustworthiness. In this work, we increased the trustworthiness for the automatic prover Metis. The reverse engineering to grasp the reasoning of a theorem prover can help to reveal issues in these systems maybe unkwon even for their own developers. During this research, we had the opportunity to contribute to Metis by reporting some bugs—see Issues No. 2, No. 4, and commit 8a3f11e in Metis' official repository.[7] Fortunately, all these problems were fixed quickly by Hurd in Metis 2.3 (release 20170822).

Acknowledgments. The authors would like to thank to Universidad EAFIT for funding and supporting this publication (grant # 690-000088). We thank Joe Leslie-Hurd for his support and comments about Metis. Special thanks to Juan Carlos Agudelo Agudelo and Håkon Robbestad Gylterud, for reading and suggesting corrections of a preliminary version of this documment.

References

[1] Andreas Abel and Thorsten Altenkirch. A Predicative Analysis of Structural Recursion. Journal of Functional Programming 12.01 (2002), pp. 1–41. DOI: 10.1017/S0956796801004191 (cit. on pp. 3, 4).

[2] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Certified Programs and Proofs (CPP 2011). Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7080. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12 (cit. on p. 18).

[3] Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, 2004. DOI: 10.1007/978-3-662-07964-5 (cit. on p. 3).

[7] https://github.com/gilith/metis.

[4] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated Proof Construction in Type Theory Using Resolution. Journal of Automated Reasoning 29.3-4 (2002), pp. 253–275. DOI: 10.1023/A:1021939521172 (cit. on pp. 2, 3, 18).

[5] Jasmin Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. Journal of Automated Reasoning 51.1 (June 2013), pp. 109–128. DOI: 10.1007/s10817-013-9278-5 (cit. on p. 18).

[6] Sascha Böhme and Tjark Weber. Designing Proof Formats: A User's Perspective - Experience Report. In: First International Workshop on Proof eXchange for Theorem Proving - PxTP 2011. 2011. URL: http://hal.inria.fr/hal-00677244 (cit. on p. 1).

[7] Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In: Interactive Theorem Proving (ITP 2010). Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 179–194. DOI: 10.1007/978-3-642-14052-5_14 (cit. on p. 18).

[8] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In: Automated Deduction (CADE-22). Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Artifical Intellingence. Springer, 2009, pp. 151–156. DOI: 10.1007/978-3-642-02959-2_12 (cit. on p. 18).

[9] Ana Bove. General recursion in type theory. Doktorsavhandlingar vid Chalmers Tekniska Hogskola 1889 (2002), pp. 39–58. DOI: 10.1017/S0960129505004822 (cit. on p. 3).

[10] Ana Bove and Venanzio Capretta. Recursive Functions with Higher Order Domains. In: Typed Lambda Calculi and Applications (TLCA 2005). Ed. by Paweł Urzyczyn. Vol. 3461. Lecture Notes in Computer Science. Springer, 2005, pp. 116–130. DOI: 10.1007/11417170_10 (cit. on p. 3).

[11] Leran Cai, Ambrus Kaposi, and Thorsten Altenkirch. Formalising the Completeness Theorem of Classical Propositional Logic in Agda. Unpublished. 2015. URL: https://akaposi.github.io/proplogic.pdf (cit. on p. 5).

[12] T. Coquand. Pattern Matching With Dependent Types. 1992. DOI: 10.1.1.37.9541 (cit. on pp. 3, 4).

[13] Dirk van Dalen. Logic and Structure. Universitext. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. DOI: 10.1007/978-3-662-02962-6 (cit. on pp. 3, 5).

[14] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT 2003). Ed. by Enrico Giunchiglia and Tacchella. Armando. Vol. 2919. Lecture Notes in Computer Science. Springer, 2004, pp. 116–130. DOI: 10.1007/978-3-540-24605-3_37 (cit. on p. 18).

[15] Burak Ekici et al. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7 (cit. on p. 18).

[16] Michael Färber and Cezary Kaliszyk. Metis-based Paramodulation Tactic for HOL Light. In: Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015. Ed. by Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov. Vol. 36. EPiC Series in Computing. EasyChair, 2015, pp. 127–136. URL: http://www.easychair.org/publications/paper/245314 (cit. on p. 5).

[17] Mathias Fleury and Jasmin Blanchette. Translation of Proofs Provided by External Provers. Tech. rep. Techniche Universität München, 2014. URL: http://perso.eleves.ens-rennes.fr/~mfleur01/documents/Fleury_internship2014.pdf (cit. on pp. 1, 18).

[18] Simon Foster and Georg Struth. Integrating an Automated Theorem Prover in Agda. In: NASA Formal Methods (NFM 2011). Ed. by Mihael Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 116–130. DOI: 10.1007/978-3-642-20398-5_10 (cit. on p. 18).

[19] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. WALDMEISTER - High-Performance Equational Deduction. Journal of Automated Reasoning 18.2 (1997), pp. 265–270. DOI: 10.1023/A:1005872405899 (cit. on p. 18).

[20] 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 (2003), pp. 56–68. URL: http://www.gilith.com/research/papers (cit. on pp. 2, 5).

[21] Joe Hurd. Integrating Gandalf and HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2001). Ed. by Yves Bertot, Gilles Dowek, Laurent Théry, and Christine Paulin. Vol. 1690. Lecture Notes in Computer Science. Springer, 2001, pp. 311–321. DOI: 10.1007/3-540-48256-3_21 (cit. on p. 18).

[22] Clément Hurlin, Amine Chaieb, Pascal Fontaine, Stephan Merz, and Tjark Weber. Practical Proof Reconstruction for First-Order Logic and Set-Theoretical Constructions. In: Proceedings of the Isabelle Workshop 2007. Ed. by Lucas Dixon and Moa Johansson. Bremen, Germany, 2007, pp. 2–13 (cit. on p. 2).

[23] Cezary Kaliszyk and Josef Urban. PRocH: Proof Reconstruction for HOL Light. In: Automated Deduction (CADE-24). Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Artifical Intellingence. Springer, 2013, pp. 267–274. DOI: 10.1007/978-3-642-38574-2_18 (cit. on pp. 2, 19).

[24] Karim Kanso. Agda as a Platform for the Development of Verified Railway Interlocking Systems. PhD thesis. Department of Computer Science. Swansea University, 2012. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.310.1502 (cit. on p. 18).

[25] Karim Kanso and Anton Setzer. A Light-Weight Integration of Automated and Interactive Theorem Proving. Mathematical Structures in Computer Science 26.1 (2016), pp. 129–153. DOI: 10.1017/S0960129514000140 (cit. on pp. 2, 18).

[26] Chantal Keller. A Matter of Trust: Skeptical Communication Between Coq and External Provers. PhD thesis. École Polytechnique, 2013. URL: https://hal.archives-ouvertes.fr/pastel-00838322/ (cit. on pp. 1, 2).

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Vol. 2283. Springer Science & Business Media, 2002 (cit. on p. 18).

[28] Hans de Nivelle. Bliksem 1.10 User Manual. 2003. URL: http://www.ii.uni.wroc.pl/~nivelle/software/bliksem. (cit. on p. 18).

[29] Bengt Nordström, Kent Petersson, and Jan M. Smith. Programming in Martin-Löf's Type Theory. Oxford University Press, 1990 (cit. on p. 2).

[30] Michael Norrish and Konrad Slind. The HOL system description. 2017. URL: https://sourceforge.net/projects/hol/files/hol/kananaskis-11/kananaskis-11-reference.pdf/download (cit. on p. 18).

[31] Lawrence C. Paulson. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In: Proceedings of the 2nd Workshop on

Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010. Ed. by Renate A. Schmidt, Stephan Schulz, and Boris Konev. Vol. 9. EPiC Series in Computing. EasyChair, 2010, pp. 1–10. URL: http://www.easychair.org/publications/paper/52675 (cit. on p. 2).

[32] Lawrence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In: Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Lecture Notes in Computer Science. Springer, 2007, pp. 232–245. DOI: 10.1007/978-3-540-74591-4_18 (cit. on pp. 1, 5).

[33] Jonathan Prieto-Cubides. A Collection of Propositional Problems in TPTP Format. June 2017. DOI: 10.5281/ZENODO.817997 (cit. on pp. 2, 18).

[34] Jonathan Prieto-Cubides. A Library for Classical Propositional Logic in Agda. 2017. DOI: 10.5281/ZENODO.398852 (cit. on pp. 2, 5, 19).

[35] Jonathan Prieto-Cubides. A Translator Tool for Metis Derivations in Haskell. 2017. DOI: 10.5281/ZENODO.437196 (cit. on pp. 2, 18, 19).

[36] Jonathan Prieto-Cubides. Metis Prover Reasoning for Propositional Logic in Agda. 2017. DOI: 10.5281/ZENODO.398862 (cit. on pp. 2, 9, 19).

[37] Jonathan Prieto-Cubides. Reconstructing Propositional Proofs in Type Theory. Universidad EAFIT, 2017 (cit. on pp. 10, 12, 14, 18).

[38] Nik Sultana, Christoph Benzmüller, and Lawrence C. Paulson. Proofs and Reconstructions. In: Frontiers of Combining Systems (FroCoS 2015). Ed. by Carsten Lutz and Silvio Ranise. Vol. 9322. Lecture Notes in Computer Science. Springer, 2015, pp. 256–271. DOI: 10.1007/978-3-319-24246-0_16 (cit. on p. 8).

[39] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. Journal of Automated Reasoning 43.4 (July 2009), p. 337. DOI: 10.1007/s10817-009-9143-8 (cit. on p. 6).

[40] Geoff Sutcliffe and Stuart Melville. The Practice of Clausification in Automatic Theorem Proving. South African Computer Journal 18 (1996), pp. 57–68 (cit. on p. 13).

[41] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allen Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In: International Joint Conference on Automated Reasoning (IJCAR 2006). Ed. by Ulrich Furbach and Natarajan Shankar. Vol. 4130. Lecture Notes in Artifical Intellingence. Springer, 2006, pp. 67–81. DOI: 10.1007/11814771_7 (cit. on p. 6).

[42] The Agda Developement Team. Agda 2.4.2.3. 2015. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 2).

[43] The Coq Developement Team. The Coq Proof Assistant. Reference Manual. 2015 (cit. on p. 18).

[44] Philip Wadler. Propositions as Types. Communications of the ACM 58.12 (2015), pp. 75–84 (cit. on p. 3).

Appendix A Customized TSTP syntax

We adopted a special TSTP syntax to improve the readability of the TSTP examples shown in this document. Some of the modifications to the original presentation of TSTP syntax in Section 3.2 are the following.

- The formulas names are sub indexed (e. g., instead of `axiom_0`, we write $\texttt{axiom}_0$).
- We use `inf` instead of `inference` field.
- We shorten names generated automatically by `Metis`, (e. g., $\texttt{s}_0$ instead of `subgoal_0` or $\texttt{n}_0$ instead of `normalize_0`).
- We remove the `plain` role.
- We remove empty fields in the inference information.
- The brackets in the argument of a unary inference are removed (e. g., instead of `inf(rule, [], [`$\texttt{n}_0$`]))`, we write `inf(rule, [],` $\texttt{n}_0$`))`).
- If the inference rule does not need arguments except its parent nodes, we remove the field of useful information (e. g., `inf(canonicalize, premise)` instead of `inf(canonicalize, [], premise)`).
- We use the symbols ($\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\supset$) for formulas instead of (`$false`, `$true`, `~`, `&`, `|`, `=>`) TPTP symbols.
- When the purpose to show a TSTP derivation does not include some parts of the derivation we use the ellipsis (`...`) to avoid such unnecessary parts.

For example, let us consider the TSTP derivation generated by `Metis` in Fig. 6 and its customized version in Fig. 7

```
fof(premise, axiom, p).
fof(goal, conjecture, p).
fof(subgoal_0, plain, p, inference(strip, [], [goal])).
fof(negate_0_0, plain, ~ p, inference(negate, [], [subgoal_0])).
fof(normalize_0_0, plain, ~ p,
  inference(canonicalize, [], [negate_0_0])).
fof(normalize_0_1, plain, p,
  inference(canonicalize, [], [premise])).
fof(normalize_0_2, plain, $false,
  inference(simplify, [], [normalize_0_0, normalize_0_1]))
cnf(refute_0_0, plain, $false,
  inference(canonicalize, [], [normalize_0_2])).
```

Fig. 6. `Metis`' TSTP derivation for the problem $p \vdash p$.

```
fof(premise, axiom, p).
fof(goal, conjecture, p).
fof(s0, p, inf(strip, goal)).
fof(neg0, ¬ p, inf(negate, s0)).
fof(n0, ¬ p, inf(canonicalize, neg0)).
fof(n1, p, inf(canonicalize, premise)).
fof(n2, ⊥, inf(simplify, [n0, n1]))
cnf(r0, ⊥, inf(canonicalize, n2)).
```

Fig. 7. `Metis`' `TSTP` derivation using a customized syntax