Reconstruction of SMT proofs with Lambdapi

Alessio Coltellacci[1], Gilles Dowek[2] and Stephan Merz[1]

[1]*University of Lorraine, CNRS, Inria, Nancy, France*
[2]*University of Paris-Saclay, Inria, ENS Paris-Saclay, CNRS, LMF, Gif-sur-Yvette, France*

Abstract
The Alethe format is a representation of unsatisfiability proofs that has been adopted by several SMT solvers. We describe work in progress for interpreting Alethe proofs and generating corresponding proofs that are accepted by the Lambdapi proof checker, a foundational proof assistant based on dependent type theory and rewriting rules that serve as a pivot for exchanging proofs between several interactive proof assistants. We give an overview of the encoding of SMT logic and Alethe proofs in Lambdapi and present initial results of the evaluation of the checker on benchmark examples.

1. Introduction

SMT solvers are widely used as automatic proof engines within interactive theorem provers or program verification tools. When they are used as trusted proof engines, any bugs in the SMT solver could lead to inconsistent theorems in the interactive prover, where correctness is paramount. State-of-the-art SMT solvers have been found to have bugs [1] due in part to error-prone optimizations, despite the best efforts of developers.

State-of-the-art SMT solvers can produce certificates (or proof traces) that can be checked independently, thus avoiding integrators to place blind trust in proof backends. This approach presents a good compromise between formally verifying the correctness of the solvers and not affecting their performance. For example, it has been adopted in [2, 3] to reconstruct the proof trace in the proof assistant Isabelle/HOL.

In this paper, we describe how SMT proofs can be reconstruced in the proof assistant Lambdapi [4], an offspring of Dedukti. Lambdapi is an interactive proof system based on the $\lambda\Pi$-calculus modulo rewriting, featuring dependent types as in Martin-Löf's type theory and allowing users to define rewriting rules in order to reason modulo equations. It is intended as an assembly language for proof assistants, enabling mechanical conversions of proofs between different systems through its built-in export or with its galaxy of external tools that provide interoperability between Lambdapi and other theorem provers (Figure 1). Consequently, the aims of our approach are twofold: reconstructing SMT proofs in Lambdapi to guarantee their correctness as well as translating the proofs so that they can be accepted by proof assistants such as Coq or Lean so that they can benefit from SMT proof support.

Our work is based on the proof checker Carcara [5] implemented in Rust, an independent checker and elaborator for the SMT proofs format Alethe. This proof format is supported by the veriT solver, and more recently by the CVC5 solver [6]. We present an extension of Carcara for translating the Alethe proof into Lambdapi. We took advantage of Carcara's elaboration of Alethe's proof, which helps increase the success rate of proof reconstruction. The Alethe format allows steps of different granularity, facilitating proof production, but verifying coarse-grained steps can require expensive proof search and may lead to verification failures. Proof elaboration by Carcara transforms coarse-grained steps into more fine-grained ones, increasing the potential success rate of reconstructing Alethe proofs.

Overview of the paper

Section 2 introduces the Alethe proof format and describes the elaborated proof produced by Carcara. In Section 3, we present first the embedding of Alethe logic in Lambdapi, and then how we extended

22nd International Workshop on Satisfiability Modulo Theories (SMT 2024)

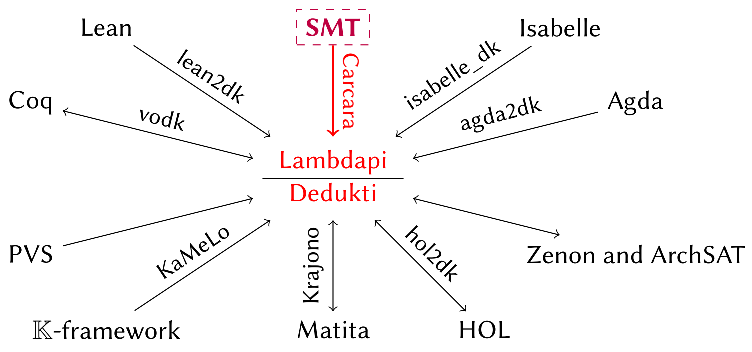

Figure 1: Lambdapi, an assembly language for proof systems

Carcara to mechanically translate an elaborated Alethe proof into a Lambdapi proof script. In Section 4, we argue the soundness property of our work, and Section 5 provides an evaluation on a set of proofs from the SMT-LIB benchmarks and proof obligations of a case study from the proof assistant TLAPS. We conclude and outline future work in Section 6.

Related work

Hammers are components of some proof assistants such as Sledgehammer [7] for Isabelle/HOL, and CoqHammer [8, 9] for Coq, that employ first-order automatic theorem provers (ATPs), including SMT solvers, to discharge goals. The hammer translates the conjecture and any user-supplied facts to the input language of the back-end, invokes it, and in the case of success attempts to reconstruct the proof in the logic of the proof assistant, based on a trace of the proof found by the back-end. For example, adoption by Sledgehammer of the Alethe format generated by the SMT solver veriT [3, 10] cut the failure rate of reconstruction by 50% and reduced the checking time by 13%. These results encouraged us to select the Alethe proof format to construct our solution.

SMTCoq is a Coq plugin written in Coq and fully certified [11] that checks proof witnesses coming from external SAT and SMT solvers. For proof reconstruction, SMTCoq relies on computational reflection: the certificate is directly processed by the reduction mechanism of Coq's kernel. In our first investigations, we attempted to convert the SMTCoq proof certificate to Lambdapi. However, we found the proof term generated by computational reflection, including subterms corresponding to arithmetic decision procedures in Micromega [12], to be too complex to be converted to Lambdapi. Moreover, SMTCoq supports at this point an old version of the Alethe proof format.

Carcara [5] is an independent proof checker and proof elaborator for SMT proofs in the Alethe format that is implemented in Rust. Although Carcara is not a certified checker like SMTCoq, it allows a coarse-grained proof, containing implicit steps, to be elaborated to a fine-grained one that includes more detailed steps or adds missing parameters. The resulting proof is intended to be easier to check by an external proof assistant, in particular given the lack of meta-programming in the vernacular language of Lambdapi.

2. Alethe

The Alethe proof format [13] for SMT solvers comprises two parts: the proof language based on SMT-LIB and a collection of proof rules. Alethe proofs are a sequence $a_1 \dots a_m,\ t_1, \dots, t_n$ where the a_is correspond to the original SMT instance being refuted, each t_i is a clause inferred from previous elements of the sequence, and t_n is $\bot$ (the empty clause). In the following sections, we designate the SMT-LIB problem as the *input problem*.

```
(declare-sort U 0)
(declare-fun a () U)
(declare-fun b () U)
(declare-fun p (U) Bool)
(assert (p a))
(assert (= a b))
(assert (not (p b)))
(get-proof)
```

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
(step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
(step t4 (cl) :rule resolution :premises (a2 t3))
```

Listing 1: Guiding *input problem* and its Alethe proof found by CVC5

In the following, we will use the *input problem* example Listing 1 with its Alethe proof (found by CVC5) to provide an overview of Alethe concepts and to illustrate our embedding in Lambdapi.

An Alethe proof inherits the declarations of its *input problem*. All symbols (sorts, functions, assertions, etc.) declared or defined in the input problem remain declared or defined, respectively. Furthermore, the syntax for terms, sort, and annotations uses the syntactic rules defined in SMT-LIB [14, §3] and the SMT signature context Σ defined in [14, §5.1 and §5.2].

2.1. The Alethe Language

An Alethe proof is a list of steps representing forward reasoning whose general form is as follows:

$$\underset{\text{index}}{i}.\quad \underset{\text{context}}{\Gamma} \triangleright \overset{\text{clause}}{l_1,\dots,l_n} \quad (\underset{\text{rule}}{\mathcal{R}}\ \underset{\text{premises}}{p_1,\dots,p_m})\ \underset{\text{arguments}}{[a_1,\dots,a_r]} \tag{1}$$

A step consists of an index $i \in \mathbb{I}$ where $\mathbb{I}$ is a countable infinite set of indices (e.g. `a0, t1`), and a clause of literals $l_1,\dots,l_n$ representing an n-ary disjunction. A proof rule $\mathcal{R}$ depends on a possibly empty set of premises $\{p_1,\dots,p_m\} \subseteq \mathbb{I}$ that refer to earlier steps. A rule might also depend on a list of arguments $[a_1,\dots,a_r]$ where each argument a_i is either a term or a pair (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The context Γ is a list $c_1,\dots,c_l$ where each element c_j is either a variable or a variable-term tuple denoted $c_j \mapsto t_j$. Therefore steps with a non-empty context contain variables c_j in l_i that can be substituted by t_j. Proof rules $\mathcal{R}$ are structured around the introduction of theory lemmas and `resolution` which captures hyper-resolution on ground first-order clauses.

We now have the key components to explain the guiding proof Listing 1 that consists of seven steps. The proof starts with several `assume` steps `a0`, `a1`, `a2` that restate the assertions from the input problem. Step `t1` introduces with the rule `equiv_pos2` a tautology of the general form $\neg(\varphi_1 \approx \varphi_2) \lor \neg\varphi_1 \lor \varphi_2$. Steps `t2`, `t3`, `t4` use earlier premises that correspond to previous steps. Step `t2` prove $p(a) \approx p(b)$ by congruence (rule `cong`) by using the assumption `a1`. Step `t3` derives $p(b)$ after applying the `resolution` rule of propositional logic to the premises `t1, t2, a0`. Lastly, the step `t4` concludes the proof by generating the empty clause $\bot$, concretely denoted as `(cl)` in Listing 1. Notice that the contexts Γ of each step are all empty in this proof.

Unfortunately, Alethe proofs provided by SMT solvers such as veriT and CVC5 can be challenging to reconstruct in a proof assistant. For instance, the order of literals in the clauses is not determined, symmetry of equality is sometimes used implicitly, and pivots for the `resolution` proof rule are not indicated explicitly.

2.2. Elaborated proof with Carcara

Carcara provides an elaboration mechanism for Alethe proofs and adds details that can make proof reconstruction easier. For example, one possible elaboration is to mention the pivot(s) for resolution steps. In our guiding example, Carcara elaborates part of the proof of Listing 1 by exposing the pivots of the steps `t3` and `t4` as arguments of `resolution` proof rule with a Boolean flag to indicate if the negation of the pivot is in the first or second premise:

```
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0)
    :args ((= (p a) (p b)) false (p a) false))
(step t4 (cl) :rule resolution :premises (a2 t3) :args ((p b) false))
```

Carcara can also shorten proofs by removing some trivial transient steps and rewriting the order of literals in a clause. The list of elaborations performed by Carcara can be found in [5, §3.2]. Our translation of Alethe proofs into Lambdapi is based on the elaboration performed by Carcara, relying on it for pre-checking the proof and applying the transformations for making it explicit.

3. Proof reconstruction

We now describe our embedding of Alethe in Lambdapi, and how we extended Carcara with a new module that can export elaborated Alethe proofs to Lambdapi. Our work applies to *input problems* expressed in the logics UFLIA, UFNIA or their sub-logics.

Notation (Alethe signature and judgment). *We use Θ instead of Σ to denote the SMT signature context to avoid conflicts with the Lambdapi signature context. Therefore, $\Theta^{\mathcal{S}}$ represents the set of SMT sort symbols, $\Theta^{\mathcal{F}}$ the set of function symbols, and $\Theta^{\mathcal{X}}$ the set of variables. We refer to* step *and* assume *as* commands *or sometimes as* Alethe judgments.

3.1. Lambdapi

Lambdapi is an implementation of λΠ modulo theory (λΠ/ ≡) [15], an extension of λΠ, i.e., the Edinburgh Logical Framework [16], a simply typed λ-calculus with dependent types. λΠ/ ≡ adds user-defined higher-order rewrite rules. Its syntax is given by

Universes	$u ::= \mathrm{TYPE} \mid \mathrm{KIND}$
Terms	$t, v, A, B, C ::= c \mid x \mid u \mid \Pi\, x\colon A.\, B \mid \lambda\, x\colon A.\, t \mid t\ v$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x\colon A$
Signatures	$\Sigma ::= \langle\rangle \mid \Sigma, c\colon C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v$

where c is a constant and x is a variable (ranging over disjoint sets), C is a closed term. *Universes* are constants used to verify if a type is well-formed – more details can be found in [16, §2.1]. $\Pi\ x\colon A.\, B$ is the dependent product, and we write $A \to B$ when x does not appear free in B, $\lambda\, x\colon A.\, t$ is an abstraction, and $t\ v$ is an application. Signature Σ (global context) and contexts Γ (local contexts) are finite sequences and $\langle\rangle$ denotes the empty sequence. Assumptions are written $c\colon C$, indicating that c is of type C. Definitions in Σ are written $c := t : C$, indicating that c has the value t and type C. In a Lambdapi typing judgment $\Gamma \vdash_\Sigma t\colon A$ a term t has type A in the context Γ and the signature Σ.

A signature may contain rewrite rules $t \hookrightarrow v$ such that $t = c\, v_1 \ldots v_n$ with c a constant. The relation $\hookrightarrow_{\beta\Sigma}$ is generated by β-reduction and by the rewrite rules of Σ. Conversion $\equiv_{\beta\Sigma}$ is the reflexive, symmetric, and transitive closure of $\hookrightarrow_{\beta\Sigma}$. Let $\hookrightarrow^*_{\beta\Sigma}$ be the reflexive and transitive closure of $\hookrightarrow_{\beta\Sigma}$. The relation $\hookrightarrow_{\beta\Sigma}$ must be confluent, i.e., whenever $t \hookrightarrow^*_{\beta\Sigma} v_1$ and $t \hookrightarrow^*_{\beta\Sigma} v_2$, there exists a term w such that $v_1 \hookrightarrow^*_{\beta\Sigma} w$ and $v_2 \hookrightarrow^*_{\beta\Sigma} w$, and it must preserve typing, i.e., whenever $\Gamma \vdash_\Sigma t : A$ and $t \hookrightarrow_{\beta\Sigma} v$ then $\Gamma \vdash_\Sigma v : A$ [17].

The typing rules in $\lambda\Pi/\equiv$ are similar to those of $\lambda\Pi$ [16, §2], except for the additional rule (Conv) that identifies types modulo $\equiv_{\beta\Sigma}$ instead of just modulo $\equiv_\beta$.

$$\frac{\Gamma, \vdash_\Sigma B : u \qquad \Gamma \vdash_\Sigma t : A \qquad A \equiv_{\beta\Sigma} B}{\Gamma \vdash_\Sigma t : B}\ \text{Conv}$$

3.2. A Prelude Encoding for Alethe

Definition 1 (Prelude Encoding)**.** *Our signature context Σ contains the following definitions and rewrite rules furnished by the standard library of Lambdapi that we use to encode Alethe proofs:*

```
1 constant symbol Set : TYPE;
2 injective symbol El : Set → TYPE;
3 constant symbol ⤳ : Set → Set → Set;
4 rule El ($x ⤳ $y) ↪ El $x → El $y;
5 symbol o: Set;
6 constant symbol Prop : TYPE;
7 injective symbol Prf : Prop → TYPE;
8 rule El o ↪ Prop;
```

The constants `Set` and `Prop` (lines 1 and 6) are type universes “à la Tarski” [18, §Universes] in $\lambda\Pi/\equiv$. The type `Set` represents the universe of *small types*. We characterize *small types* as a subclass of types for which we can define equality. SMT sorts are represented in $\lambda\Pi/\equiv$ as elements of type `Set`. Since elements of type `Set` are not types themselves, we also introduce a decoding function `El`: `Set` $\rightarrow$ `TYPE` which interpret SMT sorts as $\lambda\Pi/\equiv$ types. Thus, we represent the terms of sort `Bool` of SMT by elements of type `El` o. The constructor $\rightsquigarrow$ is used to encode SMT functions and predicates.

The type `Prop` represents the universe of propositions in $\lambda\Pi/\equiv$. Like `Set`, elements of type `Prop` are not types themselves, so we introduce the decoding function `Prf`: `Prop` $\rightarrow$ `TYPE`. By analogy with the Curry-de-Brujin-Howard isomorphism, it embeds propositions into types, mapping each proposition A to the type `Prf` A of its proofs. Hence, `Bool` formulas of SMT are rewritten to $\lambda\Pi/\equiv$ propositions with the rewrite rule defined in line 8. Thereafter, we add the following definitions to those of the standard library:

```
symbol Bool ≔ o;
injective symbol Index [a: Set] : ℕ → El a;
```

For convenience, we define an alias `Bool` for o. The function `Index` is used to assign to SMT translated terms a unique identifier, so to compare that two terms are equal it is sufficient to compare their identifier. In Lambdapi syntax, an argument enclosed in square brackets e.g. `[a]` is declared implicit.

3.3. Classical connectives, quantifiers and facts

Since SMT-solvers are based on classical logic, we use the constructive connectives and quantifiers from the Lambdapi standard library and define the classical ones from them using the double-negation translation [19] as a definition.

```
1  injective symbol Prfᶜ p ≔ Prf (¬¬ p);
2  symbol ∨ᶜ p q ≔ ¬¬ p ∨ ¬¬ q;
3  constant symbol = [a] : El a → El a → Prop;
```

Therefore, Alethe classical proofs will be decoded by the decoding function $\texttt{Prf}^c$ (line 1), defined as intuitionistic proof `Prf` of the doubly negated predicate. Similarly, classical connectives and quantifiers will be defined as illustrated in line 2. Since we want to define equality restricted to *small types*, equality has a single implicit parameter `a: Set` and two indices of type `El` a.

We prove the law of excluded middle and add the proposition of Boolean extensionality stating that classical equivalence coincides with equality over Booleans.

```
constant symbol classic [p] : Prfᶜ (p ∨ᶜ ¬ p);
constant symbol prop_ext [p: El o] [q: El o]: Prfᶜ (p ⇔ᶜ q) → Prfᶜ (p = q);
```

3.4. Translating functions

We now describe how we reconstruct *input problem* definitions and an Alethe proof with Carcara. The translation of Alethe to Lambdapi is built around four functions:

- $\mathcal{S}$ maps sorts from $\Theta^{\mathcal{S}}$ to Σ types,
- $\mathcal{F}$ translates terms and formulas to $\lambda\Pi/\equiv$ terms,
- $\mathcal{D}$ translates declarations of sorts and functions in $\Theta^{\mathcal{S}}$ and $\Theta^{\mathcal{F}}$ into constants in Σ,
- $\mathcal{C}(c_1 \dots c_n)$ translates a list of commands $c_1 \dots c_n$ of the form $i.\ \Gamma \rhd\ \varphi\ (\mathcal{R}\ P)[A]$ to typing judgments $\Gamma \vdash_{\Sigma} i := M : \texttt{Prf}^{\bullet}(N)$, where $\texttt{Prf}^{\bullet}$ represents the proof of a clause and will be introduced in the next section.

In the following we will only present examples of the application of these functions on Listing 1.

Function $\mathcal{S}$ is a mapping function from sort in $\Theta^{\mathcal{S}}$ to Σ type. Sorts `Bool` and `Int` are mapped to predefined `Bool` and `int` types. User sorts such as `U` or sort predicate `(U Bool)` are mapped to `Set` and `Set` $\rightarrow$ `Bool` respectively.

The function $\mathcal{F}$ is recursively defined on the constructors for Alethe terms and formulas. The logical connectives of SMT are mapped to the classical operators presented in the previous section. For example, the formula `(or x y (not z))` is translated into the term (`x` $\vee^c$ `y` $\vee^c$ `¬z`). Terms are translated to lambda terms, e.g. `(f x y)` is translated to `f x y`. The equality of Alethe noted $x \approx y$ is translated to $x = y$.

We translate declarations (`declare-sort` and `declare-fun`) to Lambdapi symbols by iterating over elements in context Θ and using the function $\mathcal{D}$. This function creates a constant in the context Σ for each sort and function declared. To illustrate how context embedding operates, the code below depicts the translation of sort and function declarations of our guiding example Listing 1. The context Θ for our example is as follows: $\Theta^{\mathcal{S}} = \{U, \texttt{Bool}\}$ with $ar = \{U \mapsto 0, \texttt{Bool} \mapsto 0\}$, $\Theta^{\mathcal{F}} = \{(a, U), (b, U), (p, U\ \texttt{Bool})\}$ and ar the map of sorts arity.

```
symbol U : Set;
symbol a : El U ≔ Index 0;
symbol b : El U ≔ Index 1;
symbol p : El (U ⤳ Bool) ≔ Index 2;
```

Remark (`assert` statement)**.** *The assertions at the end of Listing 1 remain untranslated initially, as they will undergo translation when we process the assume command.*

3.5. Embedding Clauses

Before presenting the function $\mathcal{C}$, we have to outline the challenge of formalizing the Alethe resolution rule in $\lambda\Pi/\equiv$. Alethe identifies clause $(\text{cl}\ l_1, \ldots, l_n)$ in Equation (1) as a set of literals which can be interpreted as an n-ary disjunction of literals. Following this, an arbitrary clause such as $cl\ (l_1\ l_2\ l_3\ l_4)$ will be then translated into $l_1 \lor^c (l_2 \lor^c (l_3 \lor^c l_4))$ by our function $\mathcal{F}$. As mentioned in Section 2, Alethe identifies clauses that differ only in the order of literals. Therefore, the concatenation of two clauses (what is happening in the conclusion of `resolution`) need not unify with the translated clause of the step given by $\mathcal{F}$. For example, taking 3 arbitrary steps:

$$A.\ \rhd\ (\text{cl}\ x_1, x_2, x_3)(..)[..]$$
$$B.\ \rhd\ (\text{cl}\ y_2, \neg x_1, y_3)(..)[..]$$
$$C.\ \rhd\ (\text{cl}\ x_2, x_3, y_2, y_3)(\texttt{resolution A B})[(x_1\ \texttt{true})]$$

Considering the interpretation of clause as disjunction we obtain the *concatenation* as follows $((x_2 \lor^c x_3) \lor^c (y_2 \lor^c y_3))$ with the conclusion of `resolution` inference rule traditionally formalized as $x \lor Y;\ \neg x \lor Z \vdash Y \lor Z$. However, the clause in step C will be translated as $(x_2 \lor^c (x_3 \lor^c (y_2 \lor^c y_3)))$, hence we obtain two different representations of the clause. Moreover, the pivot $\neg x_1$ in step B does not appear at the head of the clause whereas the inference rule for resolution expects the pivot to be at the head. Thus, the resolution of clauses makes the structure of clauses involve reasoning modulo associativity and commutativity. To address the problem of associativity, we provide a structure of clauses with a canonical representation. We define clauses as *lists* à la Church:

```
1   constant symbol Clause : TYPE;
2   symbol ■ : Clause; // Nil
3   injective symbol ⩒ : Prop → Clause → Clause; // Cons head tail
4
5   symbol ⩒_to_∨^c_rw : Clause → Prop;
6   rule ⩒_to_∨^c_rw ($x ⩒ $y) ↪ $x ∨^c (⩒_to_∨^c_rw $y)
7   with ⩒_to_∨^c_rw ■ ↪ ⊥;
8
9   symbol ++ : Clause → Clause → Clause; // concatenation
10  rule ■ ++ $m ↪ $m
11  with ($x ⩒ $l) ++ $m ↪ $x ⩒ ($l ++ $m);
12
13  injective symbol Prf• cl ≔ Prf^c (⩒_to_∨^c_rw cl); // Proof of clause
```

At lines 1 to 3 above we define the `Clause` type with the two constructors similar to the common algebraic data type of lists. The rewrite rules $⩒$`_to_`$\lor^c$`_rw` at lines 5-7 rewrite a clause into a disjunction terminating with $\bot$ symbolizing the empty list constructor. The symbol ++ defined at lines 9-11 computes the concatenation of two clauses. To solve the issue of commutativity when Alethe performs a resolution, we introduce intermediate lemmas of rearrangement of a clause where the pivot is moved, so pivots appear at the head of the clause. A clause proof will be encoded as a proof of $\texttt{Prf}^\bullet\ l_1, \ldots, l_n$ defined as a classical proof of disjunctions of literals with a trailing $\bot$.

3.6. Translation of Alethe proofs

In the previous sections, we outlined how we embed Alethe logic in $\lambda\Pi/\equiv$ and how we translate the *input problem* definitions by using the function $\mathcal{D}$. We now provide an overview of how we reconstruct the commands with function $\mathcal{C}$. Informally, the function represents each command $i.\Gamma \rhd l_1 \ldots l_n\ (R\ P)[A]$ as an intermediate lemma $\Gamma \vdash_\Sigma i := N : \texttt{Prf}^\bullet(\mathcal{F}(l_1)\ ⩒ \cdots ⩒\ \mathcal{F}l_n\ ⩒\ ■))$ where the proof term N is constructed depending on the rule R, the premises P and arguments A. For example, if R = `equiv_pos2`, we apply our generic proved lemma `equiv_pos2`: $\Pi a, \Pi b, \neg(a = b) \lor^c \neg a \lor^c b$ to produce a proof term for judgment i. In the case that R = `resolution`, we refashion n-ary resolution into a chain of binary resolution steps. To do that, we *fold in* premises P from left to right, combining intermediate lemmas to conclude the clause $l_1 \dots l_n$ of step i.

To illustrate the result of our main function $\mathcal{C}$, we introduce the translation of Listing 1 proof. In the code below, all the `assume` commands are transformed as constants in Σ by the function $\mathcal{C}$. They are treated as *axioms* and correspond to the `asserts` of the *input problem*.

```
constant symbol a0 : Prf• (p a ⩒ ■);
constant symbol a1 : Prf• (a = b ⩒ ■);
constant symbol a2 : Prf• (¬ p b ⩒ ■);

opaque symbol qf_unsat_05_predcc : Prf^c (¬ (p b)) ≔
begin
apply contradiction; assume goal;
have t1 : Prf• ((¬ (((p a) = (p b))))  ⩒ (¬ ((p a)))  ⩒ (p b) ⩒ ■) {  apply equiv_pos2; };
have t2 : Prf• (((p a) = (p b)) ⩒ ■) { apply ∨^c_i1; apply feq^c p (Prf•_l a1); };
have t3 : Prf• ((p b) ⩒ ■) {
    have t1_t2 : Prf• ((¬ ((p a)))  ⩒ (p b) ⩒ ■) { apply resolution_r _ _ _ t1 t2; };
    have t1_t2_a0 : Prf• ((p b) ⩒ ■) { apply resolution_r _ _ _ t1_t2 a0; };
    refine t1_t2_a0;
};
have t4 : Prf• ■ { have a2_t3 : Prf• ■ { apply resolution_r _ _ _ a2 t3; }; refine a2_t3;
};
apply t4;
end;
```

Each judgment translated as an intermediate lemma $\Gamma \vdash_{\Sigma} i := N : \texttt{Prf}^{\bullet}(\mathcal{F}(l_1) \dot{\vee} \cdots \dot{\vee} \mathcal{F}(l_n) \dot{\vee} \blacksquare))$ generated by $\mathcal{C}$ is represented by the tactic `have` x: t of Lambdapi that applies the *cut* rule. We wrap all the translated steps in a lemma symbol where the last `assert` is the type and the last step (t_4 here) should conclude $\texttt{Prf}^{c}(\bot)$ and $\texttt{Prf}^{\bullet}\blacksquare \equiv_{\beta\Sigma} \texttt{Prf}^{c}\bot$. We derive a proof by `contradiction` because SMT solvers try to prove that the negation of the formula is unsatisfiable. The `goal` hypothesis assumed line 3 is left unused because we use its equivalent constant `a2` coming from the translation of `assume` commands by $\mathcal{C}$.

4. Soundness of the translation

Intuitively, proving soundness of the translation amounts to showing that for any Alethe judgment c, the translation $\mathcal{C}(c)$ produces a well-typed proof term whose type corresponds to the clause asserted by c. This is formally expressed by the following theorem.

Theorem 1 (Soundness). *For any Alethe judgment* $c = i.\,\Gamma\ \triangleright\ l_1 \dots l_n\ (\mathcal{R}\ P)[A]$ *the translation* $\mathcal{C}(c)$ *produce the typing judgment* $\Gamma \vdash_{\Sigma} i := M : \texttt{Prf}^{\bullet}(\mathcal{F}(l_1) \dot{\vee} \cdots \dot{\vee} \mathcal{F}(l_n) \dot{\vee} \blacksquare)$ *that is well-typed in context* Γ *with the signature context* Σ.

5. Evaluation

Our benchmark is composed of files from the SMT-LIB benchmark[1] using the (sub-)logic UFNIA, and proof obligations from TLAPS SMT backend verifier. TLA$^+$ [20] is a language based on set theory and a linear-time temporal logic for formally specifying and verifying systems, in particular concurrent and distributed algorithms. Its interactive proof environment TLAPS [21] relies on users guiding the proof effort, it integrates automatic backends, including SMT solvers to discharge proof obligations [22, 23].

A particular goal of our development was the reconstruction of TLA$^+$ theorems in Lambdapi. The Allocator module [24] is a case study for the specification and analysis of reactive systems in TLA$^+$. The SMT proofs for the individual steps in this case study contain between 20 and 600 steps. The Cantor

[1]https://smtlib.cs.uiowa.edu/benchmarks.shtml

Name	Logic	Samples	Reconstructed	Timeout	Memout
Allocator	UFNIA	36	31	3	2
Cantor	UF	11	11	0	0
Rodin	QF_UF	34	34	0	0
TypeSafe	QF_UF	3	3	0	0

Table 1
Benchmarks results with time limit: 1200 seconds and memory limit: 30 GB

benchmark is the TLAPS proof of Cantor's theorem that asserts that there is no surjection from any set to its powerset. The SMT proofs for the corresponding proof obligations contain between 10 and 100 steps. Rodin and TypeSafe are samples from the SMT-LIB benchmark that we can reconstruct. The time needed for translation to Lambdapi by Carcara is negligibly small in this benchmark. For now, our work is not able to reconstruct larger samples (up to 70k steps) in Goel, PEQ, or Sledgehammer benchmarks from SMT-LIB repository because we are facing scalability problems such as high memory consumption and our Lambdapi checker is running only on a single process. Also, we can not for now benefit from term sharing because Lambdapi can not create local definitions in a proof, but we intend to implement this feature. However, we believe that our encoding should be able to reconstruct these proofs if we can scale because only the volume of steps and the size of terms increase but not the complexity. For benchmarks using the LIA (sub-)logic, we are currently only reconstructing those proof steps where arithmetic reasoning does not play an essential role, beyond simplification.

Our work revealed that some proof obligation in Allocator and Cantor failed to be reconstructed, i.e., the proof found by the SMT-solver had become incorrect after elaboration by Carcara. This revealed multiple bugs in the Carcara checker and proof elaborator that have since been corrected. Moreover, one reconstruction failure helped to detect that one of the set theory axioms of TLA$^+$ was incorrectly encoded in SMT, and this issue has also been fixed.

6. Conclusion and perspectives

We presented an extension of Carcara to reconstruct Alethe proofs in the foundational proof assistant Lambdapi. Currently, this extension can reconstruct some SMT proofs that originate from the TLAPS proof assistant, as well as some samples from the SMT-lib benchmarks in the UF (sub-)logic. Failure to reconstruct some proofs revealed bugs in the Carcara checker and in the SMT encoding of set theory in the TLAPS proof assistant that have since been corrected, thus demonstrating the value of independent proof checking.

At this point in time, our proof checker is limited to handling relatively small proofs due to scalability issues. We plan to address them by following the approach in [25, §4] that translates HOL-Light to Lambdapi, using multiple processes for parallel proof checking. The idea is to generate individual files corresponding to disjoint segments of a proof, compute the dependencies between those segments, check each file in a separate process, and finally merge all the results. Furthermore, we are restricted to proofs without arithmetic reasoning. In the future, we intend to support reconstruction for linear integer arithmetic. Lambdapi does not have a built-in decision procedure for it, but we consider using Zenon Modulo [26], a tableau-based first-order automated theorem prover that can generate proof certificates in Lambdapi.

A further perspective that we plan to address in future work is to exploit Lambdapi's capability for exporting proofs to other proof assistants that will then be able to benefit from automation through SMT solving.

References

[1] R. Brummayer, A. Biere, Fuzzing and delta-debugging SMT solvers, in: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT ’09, 2009.

[2] S. Böhme, T. Weber, Fast LCF-style proof reconstruction for Z3, in: M. Kaufmann, L. C. Paulson (Eds.), First Intl. Conf. Interactive Theorem Proving (ITP 2010), volume 6172 of *LNCS*, Springer, 2010, pp. 179–194.

[3] H.-J. Schurr, M. Fleury, M. Desharnais, Reliable reconstruction of fine-grained proofs in a proof assistant, in: A. Platzer, G. Sutcliffe (Eds.), Automated Deduction – CADE 28, Springer International Publishing, Cham, 2021, pp. 450–467.

[4] G. Hondet, F. Blanqui, The New Rewriting Engine of Dedukti, in: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 35:1–35:16.

[5] B. Andreotti, H. Lachnitt, H. Barbosa, Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format, in: Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Springer-Verlag, 2023, p. 367–386. URL: https://doi.org/10.1007/978-3-031-30823-9_19. doi:10.1007/978-3-031-30823-9_19.

[6] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, et al., Flexible proof production in an industrial-strength SMT solver, in: International Joint Conference on Automated Reasoning, Springer International Publishing Cham, 2022, pp. 15–35.

[7] J. C. Blanchette, S. Böhme, L. C. Paulson, Extending Sledgehammer with SMT solvers, in: N. S. Bjørner, V. Sofronie-Stokkermans (Eds.), 23rd Intl. Conf. Automated Deduction (CADE-23), volume 6803 of *LNCS*, Springer, Wroclaw, Poland, 2011, pp. 116–130.

[8] Ł. Czajka, C. Kaliszyk, Hammer for Coq: Automation for dependent type theory, Journal of Automated Reasoning 61 (2018) 423–453.

[9] L. Czajka, A Shallow Embedding of Pure Type Systems into First-Order Logic, in: 22nd International Conference on Types for Proofs and Programs (TYPES 2016), volume 97 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2018, pp. 9:1–9:39.

[10] H. Barbosa, J. C. Blanchette, M. Fleury, P. Fontaine, Scalable fine-grained proofs for formula processing, Journal of Automated Reasoning 64 (2020) 485–510.

[11] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Thery, B. Werner, A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses, in: Jouannaud, Jean-Pierre, Shao, Zhong (Eds.), First Intl. Conf. Certified Programs and Proofs (CPP 2011), volume 7086 of *LNCS*, Springer, Lenting, Taiwan, 2011, pp. 135–150. doi:10.1007/978-3-642-25379-9_12.

[12] F. Besson, Fast reflexive arithmetic tactics the linear case and beyond, in: T. Altenkirch, C. McBride (Eds.), Intl. Workshop Types for Proofs and Programs (TYPES 2006), LNCS, Springer, 2006, pp. 48–62.

[13] H. Barbosa, M. Fleury, P. Fontaine, H.-J. Schurr, The Alethe Proof Format An Evolving Specification and Reference, 2024. https://verit.gitlabpages.uliege.be/alethe/specification.pdf.

[14] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, Department of Computer Science, The University of Iowa, 2017. URL: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf, available at www.SMT-LIB.org.

[15] D. Cousineau, G. Dowek, Embedding pure type systems in the Lambda-Pi-Calculus Modulo, in: Typed Lambda Calculi and Applications, Springer, Berlin, Heidelberg, 2007, pp. 102–117.

[16] R. Harper, F. Honsell, G. D. Plotkin, A framework for defining logics, J. ACM 40 (1993) 143–184. URL: https://api.semanticscholar.org/CorpusID:13375103.

[17] F. Blanqui, Type Safety of Rewrite Rules in Dependent Types, in: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2020, pp. 13:1–13:14.

[18] P. Martin-Löf, Intuitionistic Type Theory, volume 1 of *Studies in proof theory*, Bibliopolis, 1980.

[19] G. Dowek, On the definition of the classical connectives and quantifiers, 2016. arXiv:1601.01782.

[20] L. Lamport, Specifying Systems, Addison-Wesley, Boston, Mass., 2002.

[21] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, H. Vanzetto, TLA$^+$ proofs, in: D. Giannakopoulou, D. Méry (Eds.), 18th Intl. Symp. Formal Methods (FM 2012), volume 7436 of *LNCS*, Springer, 2012, pp. ‘47–154.

[22] S. Merz, H. Vanzetto, Automatic verification of TLA$^+$ proof obligations with SMT solvers, in: Logic for Programming, Artificial Intelligence, and Reasoning - 18th , LPAR-18, volume 7180 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 289–303. doi:10.1007/978-3-642-28717-6_23.

[23] R. Defourné, Encoding TLA$^+$ proof obligations safely for SMT, in: Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings, volume 14010 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 88–106. URL: https://doi.org/10.1007/978-3-031-33163-3_7. doi:10.1007/978-3-031-33163-3_7.

[24] S. Merz, The Specification Language TLA$^+$, in: D. Bjørner, M. Henson (Eds.), Logics of specification languages, Springer, 2004, pp. 401–452.

[25] F. Blanqui, Translating HOL-Light proofs to Coq, in: 25rd Intl. Conf. Logic for Programming, Artificial Intelligence and Reasoning (LPAR-25), EPiC Series in Computing, Mauritius, 2024, p. 18 pages.

[26] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, O. Hermant, Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo, in: LPAR - Logic for Programming Artificial Intelligence and Reasoning - 2013, volume 8312 of *LNCS*, Springer, 2013, pp. 274–290. doi:10.1007/978-3-642-45221-5_20.