Automatically Translating Proof Systems for SMT Solvers to the λΠ-calculus*

Ciarán Dunne[1,3], Guillaume Burel[2,3]

[1] INRIA, ENS Paris-Saclay;
[2] ensIIE;
[3] SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, 91120 Palaiseau, France

Abstract. Eunoia is an emerging logical framework for specifying the proofs and proof systems of SMT solvers, namely CVC5. We present a translation from Eunoia to the λΠ-*calculus modulo rewriting* as implemented by the LambdaPi proof assistant. The translation is implemented by our tool `eo2lp`, which we use for generating LambdaPi encodings of (a) a large fragment of CVC5's *Cooperating Proof Calculus* (CPC), and (b) proofs produced by CVC5 on `unsat` problems from various fragments of SMT-LIB.

1 Introduction

The area of automated reasoning known as *satisfiability modulo theories* (SMT) focuses on the development of tools for checking the satisfiability of first-order formulas over combinations of mathematical theories [10]. Such tools (known as SMT *solvers*) are increasingly used as back-ends for various tasks: hardware [30] and software verification [7, 33, 26], model checking [16], cyber-security [4, 34], and to improve automation in proof assistants [1, 11]. Because these applications demand a high level of confidence in the results produced by the solver, many solvers are capable of generating *proof certificates* that witness the unsatisfiability of formulas reported as such by the solver.

The *SMT-LIB standard* [9], which specifies the syntax and semantics of a common language used for interacting with solvers. Proof generation is however not covered by the standard, leading to the the development of various proof formats, namely LFSC [37], Alethe [36], and recently Eunoia [21]. Eunoia is a proof output language and a logical framework: it allows specifying the constants, side-condition programs and inference rules of an SMT solver (e.g., CVC5); and Eunoia proof scripts may be checked by the C++ tool Ethos [21].

The λΠ-calculus modulo rewriting [19] and its implementations in the tools Dedukti [3] and LambdaPi [22] are designed to offer a trustworthy proof checker with an emphasis on interoperability of proof systems. This paper presents a translation from Eunoia to LambdaPi and an OCaml tool `eo2lp` that implements the translation. We used our tool to translate a large portion of CVC5's proof system to LambdaPi (figure 1), allowing us to further translate and typecheck over 1,000 proof scripts produced by running CVC5 on `unsat` problems from the SMT-LIB fragments for logic, linear integer arithmetic, arrays, and sets.

* Research reported in this document was supported by an Amazon Research Award, 2022.

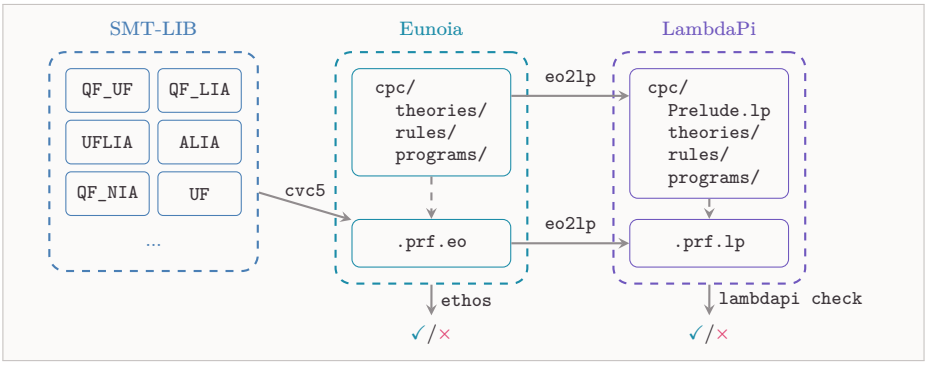

Fig. 1. Overview of the translation pipeline. CVC5 produces Eunoia proof scripts using rules from the CPC signature, which may be checked by Ethos. `eo2lp` translates both the signature and the proofs to LambdaPi, which independently verifies them.

Contribution. Section 2 formalizes the fragment of Eunoia used by CVC5, covering its term syntax, command structure, and elaboration semantics. Section 3 defines a translation from Eunoia to the $\lambda\Pi$-calculus modulo rewriting that targets Eunoia as a *framework*: inference rules, side-condition programs, and elaboration are all translated automatically. Section 4 describes our OCaml tool `eo2lp`, which translates all 22 modules of CPC-MINI (359 rules, 96 programs, 57 constants) and independently verifies 97% of 1,036 proof scripts across 14 SMT-LIB fragments.

Related Work. Coltellacci et al. [17, 18] translate Alethe [36] proofs from CVC5 [5] to LambdaPi by hand-writing lemmas for each Alethe rule, covering UF and linear integer arithmetic. Similarly, LEAN-SMT [31] encodes ∼200 of the 662 CPC rules (∼30%) as hand-written Lean 4 theorems. Our translation instead targets Eunoia as a framework, so all 359 CPC-MINI rules are translated automatically; the trade-off is that the rules are not independently proved sound in LambdaPi but taken as axioms whose well-typedness and subject reduction are verified.

The idea of reducing SMT proof checking to type checking originates with LFSC [37]; Eunoia inherits this approach and extends it with the higher-order features similar to those of the speculative SMT-LIB 3 proposal [8, 23, 6]. Brown [13] argues that this expressive type system calls for proof representations in type-theoretic frameworks; our encoding realizes this idea using the $\lambda\Pi$-calculus. SMT proof reconstruction is also used within proof assistants: Isabelle/HOL replays Alethe proofs [35, 28], SMTCoq [20] integrates solvers into Rocq, and IsaRare [29] verifies CPC rewrite rules in Isabelle/HOL.

The Dedukti ecosystem serves as an interoperability hub for proof assistants (Rocq [12], Isabelle [24], HOL Light [2], Matita [27], Lean [39]) and automated provers (iProverModulo [14], ArchSAT [15], Vampire [25], TPTP [38]); our work extends this ecosystem to SMT via the Eunoia framework.

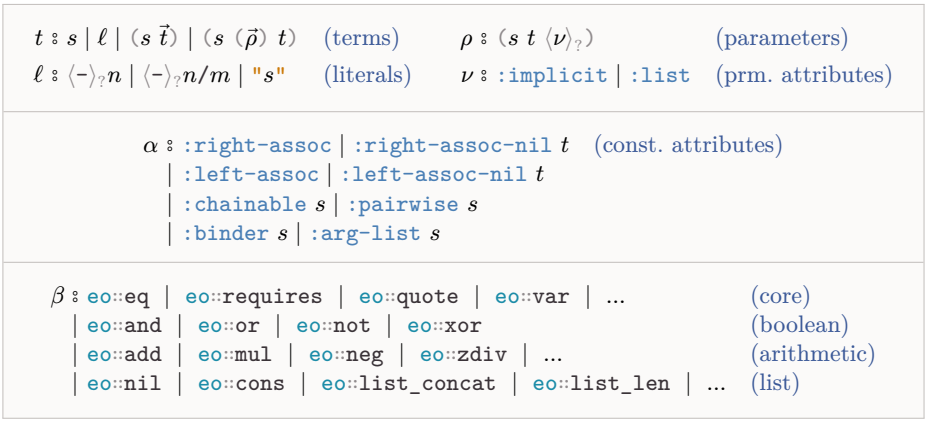

Fig. 2. Syntax for Eunoia: terms, attributes, and built-in operators.

2 An Abstract Syntax for Eunoia

Figure 2 defines syntax for the fragment of Eunoia considered in this document, where the metavariable s ranges over a set of *symbols*. First, we define grammars for the core term syntax. Each term is either a symbol s, an (integer/rational/string) *literal* ℓ, an *application* $(s\ \vec{t})$, or a *binder application* $(s\ (\vec{\rho})\ t)$. Each *parameter* ρ consists of a symbol s, a term t (the type of the parameter), and possibly a *parameter attribute* ν.

A distinctive feature of Eunoia is its support for n-ary applications of binary symbols. A *constant attribute* α may be assigned to a declared symbol to specify how n-ary applications are *elaborated* to binary form (see section 2.2). After elaboration, every application of a declared constant s uses the built-in higher-order application operator `_`, so that $(s\ t_1\ t_2)$ becomes $(\_\ (\_\ s\ t_1)\ t_2)$; only built-in and program symbols are applied directly. For example, given f with attribute :right-assoc-nil t_{nil}, the *f-list* with elements $t_1 \ldots t_n$ is the nested application $(\_\ (\_\ f\ t_1)\ (\_\ (\_\ f\ \ldots)\ (\_\ (\_\ f\ t_n)\ t_{\text{nil}})))$. Similarly, :binder t_{cons} specifies that a binder application $(s\ (\vec{\rho})\ t)$ is elaborated so that each parameter $(x_i\ t_i)$ in $\vec{\rho}$ becomes a term $(\text{eo::var}\ \text{"}x_i\text{"}\ t_i)$ within a t_{cons}-list.

Eunoia provides a fixed set of *built-in symbols* (figure 2) for manipulating terms. The list operators eo::nil, eo::cons, and eo::list_concat construct and combine f-lists; $(\text{eo::requires}\ t_1\ t_2\ t_3)$ acts as a guard, evaluating to t_3 only when t_1 and t_2 are syntactically identical; and $(\text{eo::quote}\ t)$ marks t as a *quoted argument* whose value may appear in the return type of a rule or program. Ethos natively evaluates arithmetic and string built-ins using machine operations, whereas our translation delegates to the integers and rationals of the LambdaPi standard library for encoding Eunoia literals and their built-in operations via rewrite rules.

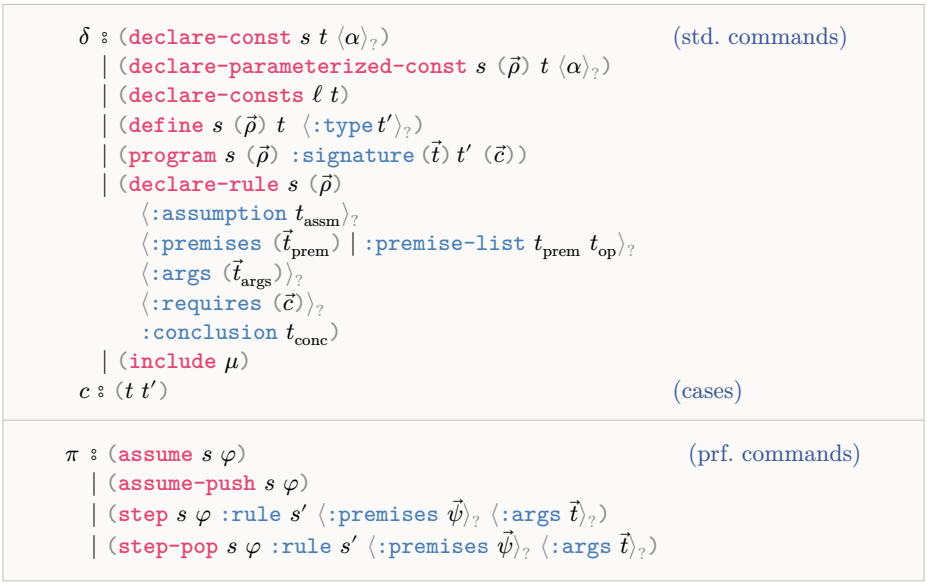

Fig. 3. Eunoia commands and proof commands.

2.1 Signatures and Proof Scripts.

The commands of Eunoia are divided into *standard commands* and *proof commands*. Hereinafter, a Eunoia *signature* is a list Δ of standard commands, and a *proof script* is a list $\Upsilon = (\vec{\delta}\,;\vec{\pi})$ where $\vec{\delta}$ is a list of standard commands called the *preamble* and $\vec{\pi}$ is a list of proof commands called the *body* of the script. In practice, a Eunoia-friendly SMT solver provides a signature Δ encoding the inference rules that may be used by its proof scripts. Then, the correctness of a proof script Υ may then be checked wrt. Δ via Ethos.

In the following discussion, we define some terminology and notation for commands, signatures, and proof scripts. The resulting interface is useful for characterizing the elaboration and translation operators in section 2.2 and section 4. We omit datatype declarations, which are not used by CPC-MINI.

Constant Declaration. Let δ be a *(parameterized) constant declaration* with symbol s, parameters $\vec{\rho}$, and term t. We may write $(\delta \vdash s(\vec{\rho}) : t)$ to express that the command δ declares t as the *type* of s with respect to parameters $\vec{\rho}$. Furthermore, if a constant attribute α is given by δ, we may write $(\delta \vdash s(\vec{\rho}) :: \alpha)$.

Macro Definition. Let δ be a *macro definition* with symbol s, parameters $\vec{\rho}$, and term t. Then δ declares t as the *definiens* of s wrt. $\vec{\rho}$; written $(\delta \vdash s(\vec{\rho}) := t)$. If the attribute `:type` t' is given by δ, then $(\delta \vdash s(\vec{\rho}) : t')$ also holds.

Program Declaration. Let δ be a *program declaration* with symbol s, parameters $\vec{\rho}$, `:signature` $(t_1 \ldots t_n)\, t'$, and cases $c_1 \ldots c_m$. Then, the type and definiens of s is given thus:

$$\delta \vdash s(\vec{\rho}) : \texttt{(->}\ t_1 \ldots t_n\ t'\texttt{)} \quad \text{and} \quad \delta \vdash s(\vec{\rho}) := \textbf{cases}[c_1, \ldots, c_m]$$

Proof Commands. Each π is either an *assumption* (i.e., `(assume` $s\ \varphi$`)` or `(assume-push` $s\ \varphi$`)`) or a *step* (i.e., `(step` $s\ \varphi$ `...)` or `(step-pop` $s\ \varphi$ `...)`). In any case, we call s the *name* of π and φ the *conclusion*, written:

$$\pi \vdash s : \texttt{(Proof}\ \varphi\texttt{)}$$

When π is a step with `:rule` r, the definiens of s is the result of applying any terms given by `:args` $(t_1 \ldots t_m)$, and `:premises` $(p_1 \ldots p_n)$, e.g.;

$$\pi \vdash s := \texttt{(}r\ t_1 \ldots t_m\ p_1 \ldots p_n\texttt{)}$$

Rule Declaration. Let δ be a *rule declaration* with symbol s, parameters $\vec{\rho}$, and `:conclusion` φ. By considering the attributes that may be supplied by δ, an appropriate type for s can be derived as follows:

1. Given `:requires` $((l_1\ r_1) \ldots (l_k\ r_k))$, define $\varphi^\star$ below. Otherwise, $\varphi^\star := \varphi$.

   $$\varphi^\star := \texttt{(eo::requires}\ l_1\ r_1\ \texttt{(} \ldots\ \texttt{(eo::requires}\ l_k\ r_k\ \varphi\texttt{)))}$$

2. If δ provides no premises, let $\tau^\star := \texttt{(Proof}\ \varphi^\star\texttt{)}$. Otherwise, define $\tau^\star$ thus:
   (a) Given `:premises` $(\psi_1 \ldots \psi_m)$, let:

   $$\tau^\star := \texttt{(->}\ \texttt{(Proof}\ \psi_1\texttt{)} \ \ldots\ \texttt{(Proof}\ \psi_m\texttt{)}\ \texttt{(Proof}\ \varphi^\star\texttt{))}$$

   (b) Given `:premise-list` $\psi\ f$ for some associative symbol f, redefine:

   $$\varphi^\star := \texttt{(eo::requires (eo::is\_list}\ f\ \psi\texttt{) true}\ \varphi^\star\texttt{)}$$

   and let $\tau^\star := \texttt{(->}\ \texttt{(Proof}\ \psi\texttt{)}\ \texttt{(Proof}\ \varphi^\star\texttt{))}$.
3. If δ provides `:args` $(t_1 \ldots t_n)$, then the type of s is given thus:

   $$\delta \vdash s(\vec{\rho}) : \texttt{(->}\ \texttt{(eo::quote}\ t_1\texttt{)} \ \ldots\ \texttt{(eo::quote}\ t_n\texttt{)}\ \tau^\star\texttt{)}$$

   Otherwise, $\delta \vdash s(\vec{\rho}) : \tau^\star$.

Example 1. Consider the inference rule declarations in figure 4. The `modus_ponens` rule has only `:premises` and `:conclusion`. By item 2a:

$$\texttt{modus\_ponens}(\vec{\rho}) : \texttt{(->}\ \texttt{(Proof F1)}\ \texttt{(Proof (=> F1 F2))}\ \texttt{(Proof F2))}$$

The declaration of `reordering` additionally uses `:args` and `:requires`. By item 3, `C2` becomes an `eo.quote`-type, and item 1 guards the conclusion thus $\texttt{C2}^\star := \texttt{(eo::requires (eo::list\_minclude or C1 C2) true C2)}$. Then:

$$\texttt{reordering}(\vec{\rho}) : \texttt{(->}\ \texttt{(eo::quote C2)}\ \texttt{(Proof C1)}\ \texttt{(Proof C2}^\star\texttt{))}$$

```
(declare-rule modus_ponens ((F1 Bool) (F2 Bool))
  :premises (F1 (=> F1 F2))
  :conclusion F2)
```

```
(declare-rule reordering ((C1 Bool) (C2 Bool))
  :premises (C1)
  :args (C2)
  :requires (((eo::list_minclude or C1 C2) true))
  :conclusion C2)
```

```
(declare-rule cnf_implies_pos ((F1 Bool) (F2 Bool))
  :args ((=> F1 F2))
  :conclusion (or (not (=> F1 F2)) (not F1) F2))
```

```
(declare-rule and_intro ((F Bool))
  :premise-list F and
  :conclusion F)
```

Fig. 4. Rule declarations from `cpc/rules/Booleans.eo`, see example 1.

The declaration of `cnf_implies_pos` has `:args ((=> F1 F2))`, where the argument is a compound term. By item 3, the `eo.quote`-typed parameter demands pattern matching on this term at the point of use, where φ is the `:conclusion`:

$$\texttt{cnf\_implies\_pos}(\vec{\rho}) : \texttt{(-> (eo::quote (=> F1 F2)) (Proof } \varphi\texttt{))}$$

Finally, `and_intro` uses `:premise-list F and`. By item 2b, the conclusion is guarded thus, $\texttt{F}^\star :=$ `(eo::requires (eo::is_list and F) true F)`. Hence:

$$\texttt{and\_intro}(\vec{\rho}) : \texttt{(-> (Proof F) (Proof } \texttt{F}^\star\texttt{))}$$

2.2 Elaboration of Terms

Terms are *elaborated* to eliminate all n-ary applications and distinguish between constant-level and program-level function applications. The elaborated form of $(f\ \vec{t})$ is determined by the attribute assigned to f by Δ. We define the **elab** operator below with the aim of supporting the elaboration strategies corresponding to the constant attributes of figure 2.

Definition 1. For any Δ and $\vec{\rho}$, let $\mathbf{elab}_{(\Delta,\vec{\rho})}$ be the least operator such that:

1. For any symbol s, $\mathbf{elab}_{(\Delta,\vec{\rho})}[s] = s$.
2. For any binder application $(s\ (\vec{v})\ t)$ where $\Delta \vdash s :: \texttt{:binder}\ t_{\text{cons}}$,

$$\mathbf{elab}_{(\Delta,\vec{\rho})}[(s\ (\vec{v})\ t)] = (s\ t_{\text{vars}}\ t_{\text{body}})$$

where t_{vars} encodes $\vec{v} = (x_1\ t_1)\ \ldots\ (x_n\ t_n)$ as a t_{cons}-list of `eo::var` terms, and t_{body} is the result of substituting each x_i in t for its `eo::var` counterpart:

$$\begin{array}{l} t_{\text{vars}} := \mathbf{elab}_{(\Delta,\vec{\rho})}[(t_{\text{cons}}\ X_1 \ldots X_n)] \\ t_{\text{body}} := \mathbf{elab}_{(\Delta,\vec{\rho})}[\mathbf{subst}_{(\vec{x},\vec{X})}[t]] \end{array} \qquad \text{where } X_i := \texttt{(eo::var "}x_i\texttt{"}\ t_i\texttt{)}$$

3. For any application $(s\ \vec{t})$, let $t'_i := \mathbf{elab}_{(\Delta,\vec{\rho})}[t_i]$ in:
   (a) If s is a built-in or $s \in \Delta.\mathsf{prog}$, then $\mathbf{elab}_{(\Delta,\vec{\rho})}[(s\ \vec{t})] = (s\ \vec{t'})$.
   (b) If $s \in \vec{\rho}$, then $\mathbf{elab}_{(\Delta,\vec{\rho})}[(s\ \vec{t})] = (\texttt{\_}\ \ldots\ (\texttt{\_}\ (\texttt{\_}\ s\ t'_1)\ \ldots)\ t'_n)$.
   (c) If $\Delta \vdash s :: \alpha$ for some associative attribute α, then:

$$\mathbf{elab}_{(\Delta,\vec{\rho})}[(s\ \vec{t})] = \begin{cases} \mathbf{foldr}(G, t_{\text{nil}}, \vec{t'}) & (\alpha = \texttt{:right-assoc-nil}\ t_{\text{nil}}) \\ \mathbf{foldl}(G, t_{\text{nil}}, \vec{t'}) & (\alpha = \texttt{:left-assoc-nil}\ t_{\text{nil}}) \\ \mathbf{foldr}(G, t'_n, t'_1 \ldots t'_{n-1}) & (\alpha = \texttt{:right-assoc}) \\ \mathbf{foldl}(G, t'_1, t'_2 \ldots t'_n) & (\alpha = \texttt{:left-assoc}) \end{cases}$$

   where[4] $G(x,y) := \mathbf{glue}_{(\vec{\rho},s)}[x,y]$ is defined as follows:

$$\mathbf{glue}_{(\vec{\rho},s)}[x,y] := \begin{cases} (\texttt{eo::list\_concat}\ s\ x\ y) & (\vec{\rho} \vdash x :: \texttt{:list}) \\ (\texttt{\_}\ (\texttt{\_}\ s\ x)\ y) & \text{otherwise.} \end{cases}$$

   (d) If $\Delta \vdash s :: \alpha$ for some non-associative attribute α, then:

$$\mathbf{elab}_{(\Delta,\vec{\rho})}[(s\ \vec{t})] = \mathbf{elab}_{(\Delta,\vec{\rho})}[T]$$

   where T is defined using **chain** and **pairs** thus:

$$T := \begin{cases} (t_{\text{cons}}\ \mathbf{chain}(s,\vec{t})) & (\alpha = \texttt{:chainable}\ t_{\text{cons}}) \\ (t_{\text{cons}}\ \mathbf{pairs}(s,\vec{t})) & (\alpha = \texttt{:pairwise}\ t_{\text{cons}}) \\ (t_{\text{cons}}\ \vec{t}) & (\alpha = \texttt{:arg-list}\ t_{\text{cons}}) \end{cases}$$

$$\mathbf{chain}(s,\ t_1 \ldots t_n) := (s\ t_1\ t_2)\ (s\ t_2\ t_3)\ \ldots\ (s\ t_{n-1}\ t_n)$$
$$\mathbf{pairs}(s,\ t_1 \ldots t_n) := (s\ t_1\ t_2)\ \ldots\ (s\ t_1\ t_n)\ (s\ t_2\ t_3)\ \ldots\ (s\ t_{n-1}\ t_n)$$

Example 2. Since $\Delta \vdash$ `or` $::$ `:right-assoc-nil false`, item 3c applies with $G(x,y) =$ `(_ (_ or` x`)` y`)` and the right fold gives:

$$\begin{aligned} &\mathbf{elab}_{(\Delta,\vec{\rho})}[\texttt{(or x y z)}] = \mathbf{foldr}(G, \texttt{false}, \texttt{x y z}) \\ &= \texttt{(\_ (\_ or x) (\_ (\_ or y) (\_ (\_ or z) false)))} \end{aligned}$$

Similarly, since $\Delta \vdash$ `=` $::$ `:chainable and`, item 3d applies with $\mathbf{chain}($`=, a b c d`$) =$ `(= a b) (= b c) (= c d)`, so:

$$\begin{aligned} &\mathbf{elab}_{(\Delta,\vec{\rho})}[\texttt{(= a b c d)}] \\ &= \mathbf{elab}_{(\Delta,\vec{\rho})}[\texttt{(and (= a b) (= b c) (= c d))}] \end{aligned}$$

which is then elaborated recursively via the `:right-assoc-nil false` attribute of `and`.

[4] $\mathbf{foldl}(G,a,l)$ and $\mathbf{foldr}(G,a,l)$ are the standard left and right folds using combining function G and accumulator a on list l.

$$\mu ::= \texttt{TYPE} \mid \texttt{KIND} \qquad \text{(universes)}$$
$$t ::= x \mid \kappa \mid \mu \mid (t\,t') \mid (\lambda\, x : t.\, t') \mid (\Pi\, x : t.\, t') \qquad \text{(terms)}$$

$$\text{(WF0)}\ \frac{}{\mathsf{wf}\,\emptyset} \qquad \text{(WF+)}\ \frac{\Gamma \vdash_\Sigma t : \mu}{\mathsf{wf}\,(\Gamma, (x : t))}\ x \notin \mathbf{dom}(\Gamma)$$

$$\text{(VAR)}\ \frac{\mathsf{wf}\,\Gamma}{\Gamma \vdash_\Sigma x : t}\ (x : t) \in \Gamma \qquad \text{(CON)}\ \frac{\mathsf{wf}\,\Gamma \qquad \vdash_\Sigma t : \mu}{\Gamma \vdash_\Sigma \kappa : t}\ (\kappa : t) \in \Sigma$$

$$\text{(UNIV)}\ \frac{\mathsf{wf}\,\Gamma}{\Gamma \vdash_\Sigma \texttt{TYPE} : \texttt{KIND}} \qquad \text{(PROD)}\ \frac{\Gamma \vdash_\Sigma t : \texttt{TYPE} \qquad \Gamma, (x : t) \vdash_\Sigma t' : \mu'}{\Gamma \vdash_\Sigma (\Pi\, x : t.\, t') : \mu'}$$

$$\text{(FUN)}\ \frac{\Gamma, (x : t) \vdash_\Sigma e : t' \qquad \Gamma \vdash_\Sigma (\Pi\, x : t.\, t') : \mu}{\Gamma \vdash_\Sigma (\lambda\, x : t.\, e) : (\Pi\, x : t.\, t')}$$

$$\text{(APP)}\ \frac{\Gamma \vdash_\Sigma e : (\Pi\, x : t.\, t') \qquad \Gamma \vdash_\Sigma e' : t}{\Gamma \vdash_\Sigma (e\, e') : t'[x \mapsto e']}$$

$$\text{(CONV)}\ \frac{\Gamma \vdash_\Sigma e : t \qquad \Gamma \vdash_\Sigma t' : \mu}{\Gamma \vdash_\Sigma e : t'}\ (t \equiv_{\beta\Sigma} t')$$

Fig. 5. Syntax and typing rules for the λΠ-calculus modulo rewriting.

3 Encoding Eunoia in the λΠ-calculus

Figure 5 provides syntax and typing rules for the λΠ-*calculus modulo rewriting*. Each term is either a *variable* x, a *constant* κ, a *universe* from $\{\texttt{TYPE}, \texttt{KIND}\}$, an *application* of two terms $(t\,t')$, or an *abstraction* $(\mathscr{B}\, x : t.\, t')$ where $\mathscr{B}$ is a *binder* from $\{\lambda, \Pi\}$. Terms are identified up to α-conversion (i.e., renaming of bound variables), and we assume appropriate definitions for *substitution* $(t[x \mapsto t'])$ and β-*reduction* $(t \rightsquigarrow_\beta t')$. Hereinafter, we may write $(t_1 \to t_2)$ for the term $(\Pi\,(x : t_1).\, t_2)$, where x is some 'fresh' variable that does not occur free in t_2.

A *typing* is an expression of the form $(t : t')$, and a *context* is a list of typings each of the form $(x : t)$. A *rewrite rule* is an expression of the form $(t \hookrightarrow t')$, where t has the form $((\kappa\, t_1)\ \ldots t_n)$. A *signature* is a list of typings and rewrite rules, where each typing has the form $(\kappa : t)$ where t has no free variables. For any signature Σ, let R_Σ be the smallest binary relation such that:

1. $(\ell \hookrightarrow r) \in \Sigma$ implies $(\ell, r) \in R_\Sigma$, and
2. R_Σ is *congruent* under application, abstraction, and substitution.

Then, *equality modulo rewriting* $(\equiv_{\beta\Sigma})$ is defined as the least equivalence relation containing R_Σ and the β-reduction relation. The rules in figure 5 provide a (mutually inductive) definition of a *well-formedness* relation (wf_Σ) on contexts and a *typing relation* $(\vdash_\Sigma)$ between contexts and typings, where $(\Gamma \vdash_\Sigma e : t)$ may be read as "e has type t with respect to signature Σ and context Γ".

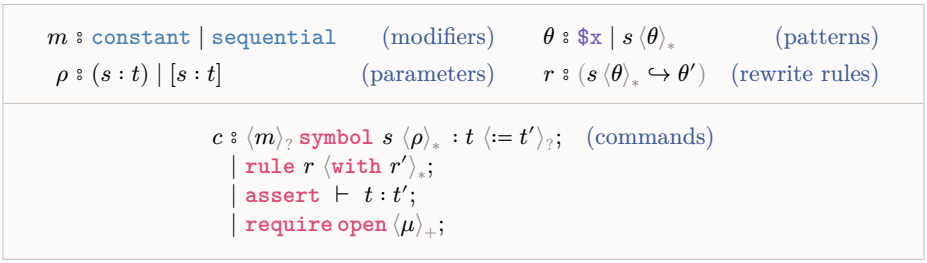

Fig. 6. Syntax for the LambdaPi proof assistant.

Remark 1. The typing rules of the $\lambda\Pi$-calculus disallow abstractions on types (i.e., any $(\texttt{TYPE} \to t)$ is ill-typed in any context). Instead, the types of Eunoia are identified with $\lambda\Pi$-terms of type Set, which in turn encode $\lambda\Pi$-types with an injective symbol $\tau : \mathsf{Set} \to \texttt{TYPE}$ and an (infix) function space symbol $(\rightsquigarrow) : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}$ with the rewrite rule $\tau\,(A \rightsquigarrow B) \hookrightarrow \tau\,A \to \tau\,B$. For dependent types arising from `eo.quote` arguments, we use a dependent function space $(\rightsquigarrow^{\mathsf{d}}) : \Pi\,T : \mathsf{Set}.\,(\tau\,T \to \mathsf{Set}) \to \mathsf{Set}$ with the rule $\tau\,(T \rightsquigarrow^{\mathsf{d}} F) \hookrightarrow \Pi\,x : \tau\,T.\,\tau\,(F\,x)$.

3.1 The LambdaPi Proof Assistant

Figure 6 defines syntax for a small fragment of the LambdaPi proof assistant. Each *file* is a list of *commands* that (a) specify the typings and rewrite rules of a particular $\lambda\Pi$-signature and (b) provide information for guiding the typechecker.

Symbol Declaration. Each *symbol declaration* provides a symbol s and a term t, which alone corresponds to the typing $(s : t)$. A list of *parameters* $\vec{\rho}$ may also be given, each of which are either *explicit* or *implicit* (written $(x : t)$ and $[x : t]$ resp.). Given $\vec{\rho}$ with n leading implicits followed by m explicits, LambdaPi registers the following typing and internally records $\zeta(s) := n$.

$$(\texttt{@}s : (\Pi\,x_1 : t_1.\ \ldots\ \Pi\,x_{n+m} : t_{n+m}.\,t))$$

Hereinafter, the symbol s is an alias for $(\texttt{@}s\ \_\ \ldots\ \_)$, where _ is a *placeholder*. Terms containing placeholders can be *resolved* with respect to a context Γ and a signature Σ, which we write as $\mathbf{resolve}_{(\Sigma,\Gamma)}[t]$ for any valid[5] term t.

Optionally, a *modifier* may be given. The `constant` modifier forbids rewrite rules with s at the head, and `sequential` ensures rewrite rules with head s are 'matched' in the order they are given in the file. Finally, a *definition* $(:= t')$ may

[5] Internally, resolution is achieved by inserting fresh metavariables $?\alpha_1 \ldots ?\alpha_n$ in place of each placeholder in a term, and solving via higher-order unification using the set of constraints produced during type-checking. For example, given the context $\Gamma := [(n, m : \tau\,\mathsf{Int}), (x, y : \tau\,\mathsf{Real})]$ and some appropriate Σ, $\mathbf{resolve}_{(\Sigma,\Gamma)}$ is well-defined for $t := (\mathsf{and}\ (= n\ m)\ (= x\ y))$ since $(\mathsf{and}\ (\texttt{@}=\ ?\alpha\ n\ m)\ (\texttt{@}=\ ?\beta\ x\ y))$ entails the (solveable) constraints $\{(\tau\,?\alpha \equiv \tau\,\mathsf{Int}), (\tau\,?\beta \equiv \tau\,\mathsf{Real})\}$.

be given when the `constant` modifier is not, which provides the rewrite rule $(s \hookrightarrow t)$ (resp. $(s \hookrightarrow \lambda\, \vec{\rho}.\, t)$ given parameters $\vec{\rho}$).

Rewrite Rule Declaration. Each *rewrite rule declaration* provides a list $r_1 \dots r_n$ of rewrite rules. The left and right sides of each rewrite rule are *patterns*, where each pattern is either a *pattern variable* (`$x`) or a *pattern application* $(s\ \langle\theta\rangle_*)$, where s is called the *head* of the pattern. A rewrite rule $(\theta \hookrightarrow \theta')$ is valid when θ is a pattern application and every pattern variable in θ' also occurs in θ. The rewrite rule(s) given by the declaration are added to the signature, effectively augmenting the typechecking procedure of LambdaPi to behave 'modulo' those rule(s).

Assertion. An *assertion* `assert` $\vdash\ t : t'$ checks that t has type t' in the current signature, resolving any placeholders via type inference. Unlike a symbol declaration, an assertion does not extend the signature.

3.2 Translating Eunoia to LambdaPi

We define a family of *translation operators* that act on the terms, types, parameters, and commands of Eunoia to produce corresponding LambdaPi syntax. Crucially, the translation assumes that all terms have already been processed by the **elab** operator of section 2.2.

Translating Terms. Because some Eunoia symbols are not valid LambdaPi identifiers, the translation on symbols uses LambdaPi syntax for escaping identifiers. That is, $\bar{s} := \{s\}$ iff s contains forbidden characters, and $\bar{s} := s$ otherwise.

Definition 2. Let $[\![\cdot]\!]_{\mathsf{tm}}$ be the least operator such that:

1. For any symbol s or literal ℓ, the following hold:

$$[\![s]\!]_{\mathsf{tm}} = \begin{cases} \texttt{eo.Type} & \text{if } s = \texttt{Type}, \\ \bar{s} & \text{otherwise.} \end{cases} \qquad [\![\ell]\!]_{\mathsf{tm}} = \begin{cases} \texttt{of\_Z}\ [\![n]\!]_{\mathsf{tm}} & \text{if } \ell = \mathsf{int}(n), \\ \texttt{of\_Q}\ [\![x]\!]_{\mathsf{tm}}\ [\![y]\!]_{\mathsf{tm}} & \text{if } \ell = \mathsf{rat}(x, y) \\ \bar{\ell} & \text{if } \ell \in \mathsf{String}. \end{cases}$$

2. For arrow types, the following hold, where $(x\ T) \in \vec{\rho}$:

$$\begin{aligned} [\![\texttt{(-> (eo::quote } x\texttt{) } \vec{t}\texttt{)}]\!]_{\mathsf{tm}} &= [\![T]\!]_{\mathsf{tm}} \rightsquigarrow^{\mathsf{d}} \lambda\, x.\ [\![\texttt{(-> } \vec{t}\texttt{)}]\!]_{\mathsf{tm}} \\ [\![\texttt{(-> } t'\ \vec{t}\texttt{)}]\!]_{\mathsf{tm}} &= [\![t']\!]_{\mathsf{tm}} \rightsquigarrow [\![\texttt{(-> } \vec{t}\texttt{)}]\!]_{\mathsf{tm}} \\ [\![\texttt{(-> } t\texttt{)}]\!]_{\mathsf{tm}} &= [\![t]\!]_{\mathsf{tm}} \end{aligned}$$

3. For binary and n-ary applications, the following hold:

$$\begin{aligned} [\![\texttt{(\_ } t_1\ t_2\texttt{)}]\!]_{\mathsf{tm}} &= \texttt{eo.app}\ [\![t_1]\!]_{\mathsf{tm}}\ [\![t_2]\!]_{\mathsf{tm}} \\ [\![\texttt{(Proof } t\texttt{)}]\!]_{\mathsf{tm}} &= \texttt{eo.Proof}\ [\![t]\!]_{\mathsf{tm}} \\ [\![\texttt{(}s\ \vec{t}\texttt{)}]\!]_{\mathsf{tm}} &= \bar{s}\ [\![t_1]\!]_{\mathsf{tm}}\ \dots\ [\![t_n]\!]_{\mathsf{tm}} \end{aligned}$$

4. The only normal-form binder application is `eo::define`, so:

$$[\![(\texttt{eo::define}\ (\vec{v})\ t')]\!]_{\mathsf{tm}} = (\mathsf{let}\ \vec{w}\ \mathsf{in}\ [\![t']\!]_{\mathsf{tm}})$$

where $w_i = (\bar{x}_i := [\![t_i]\!]_{\mathsf{tm}})$ for each $v_i = (x_i\ t_i)$.

Definition 3. Given $\vec{\rho} = (x_1\ t_1\ \langle \nu_1 \rangle_?)\ \ldots\ (x_n\ t_n\ \langle \nu_n \rangle_?)$, define:

$$[\![\vec{\rho}]\!]_{\mathsf{ctx}} := \rho_1^\star\ \ldots\ \rho_n^\star \quad \text{where} \quad \rho_i^\star = \begin{cases} [\bar{x}_i : \tau\ [\![t_i]\!]_{\mathsf{tm}}] & \text{if } \nu_i = \texttt{:implicit}, \\ (\bar{x}_i : \tau\ [\![t_i]\!]_{\mathsf{tm}}) & \text{otherwise.} \end{cases}$$

Translating Commands. In the following text, we write $[\![t]\!]^{\$}_{\mathsf{tm}}$ for the result of applying $[\![\cdot]\!]_{\mathsf{tm}}$ to t and replacing each free variable x with the pattern variable `$x`. Also, we write $\mathbf{impl}(\vec{\rho}, t)$ for the sublist of $[\![\vec{\rho}]\!]_{\mathsf{ctx}}$ containing each $x_i \in \vec{\rho}$ occurring free in t, each marked as implicit.

Definition 4. Let δ be a standard Eunoia command. Then $[\![\delta]\!]_{\mathsf{cmd}}$ is the list of LambdaPi commands described as follows:

1. If δ is a constant declaration, then $\delta \vdash s(\vec{\rho}) : t$ and the head of $[\![\delta]\!]_{\mathsf{cmd}}$ is:

$$\texttt{symbol}\ \bar{s}\ [\![\vec{\rho}]\!]_{\mathsf{ctx}} : \tau\ [\![t]\!]_{\mathsf{tm}}\ ;$$

If $\delta \vdash s :: \alpha$ and α is `:right-assoc-nil` t_{nil} or `:left-assoc-nil` t_{nil}, then the following rewrite rule is also produced:

$$\texttt{rule}\ (\texttt{eo.nil}\ \bar{s}\ \texttt{\$T}) \hookrightarrow (\texttt{eo.as}\ [\![t_{\text{nil}}]\!]_{\mathsf{tm}}\ \texttt{\$T})$$

2. If δ is a macro declaration, then $\delta \vdash s(\vec{\rho}) := t$. If the type t' is supplied then $[\![\delta]\!]_{\mathsf{cmd}}$ is defined as below, otherwise the typing is omitted.

$$\texttt{symbol}\ \bar{s}\ [\![\vec{\rho}]\!]_{\mathsf{ctx}} : \tau\ [\![t']\!]_{\mathsf{tm}} := [\![t]\!]_{\mathsf{tm}}$$

3. If δ is a program declaration with $\delta \vdash s(\vec{\rho}) : (\texttt{->}\ t_1 \ldots t_n\ t')$ and cases $(l_1\ r_1)\ \ldots\ (l_m\ r_m)$, then $[\![\delta]\!]_{\mathsf{cmd}}$ declares $\bar{s}$, where $T := \tau\ [\![(\texttt{->}\ t_1 \ldots t_n\ t')]\!]_{\mathsf{tm}}$:

$$\begin{aligned} &\texttt{sequential symbol}\ \bar{s}\ \vec{\rho}^\star : T; \\ &\texttt{rule}\ [\![l_1]\!]^{\$}_{\mathsf{tm}} \hookrightarrow [\![r_1]\!]^{\$}_{\mathsf{tm}} \\ &\ldots \\ &\texttt{with}\ [\![l_m]\!]^{\$}_{\mathsf{tm}} \hookrightarrow [\![r_m]\!]^{\$}_{\mathsf{tm}}\ ; \end{aligned}$$

where $\vec{\rho}^\star := \mathbf{impl}(\vec{\rho}, T)$ binds all of the free parameters in T as implicits.

4. If δ is a rule declaration with `:args` $(t_1 \ldots t_n)$ of type $\tau_1 \ldots \tau_n$, then:

$$\delta \vdash s(\vec{\rho}) : (\texttt{->}\ (\texttt{eo::quote}\ t_1)\ \ldots\ (\texttt{eo::quote}\ t_n)\ \tau^\star)$$

where $\tau^\star$ encodes the premises and conclusion of the rule, as per section 2.1. The `eo.quote` types are eliminated by declaring an auxiliary symbol $s^\star$ that

```
constant symbol = [A : Set] : τ (A ⇝ (A ⇝ Bool));
constant symbol or : τ (Bool ⇝ (Bool ⇝ Bool));
rule eo.nil or $T ↪ eo.as false $T;
```

```
constant symbol modus_ponens [F1 F2 : τ Bool]
  : eo.Proof F1
  → eo.Proof (eo.app (eo.app => F1) F2)
  → eo.Proof F2;
```

```
symbol cnf_implies_pos_aux : τ Bool → TYPE;
rule @cnf_implies_pos_aux
  (eo.app (eo.app => $F1) $F2)
  ↪ eo.Proof φ;
constant symbol cnf_implies_pos
  (x1 : τ Bool) : cnf_implies_pos_aux x1;
```

Fig. 7. LambdaPi output for the declarations in figure 4. See example 3.

rewrites to $[\![\tau^\star]\!]_{\mathsf{tm}}$ when applied to terms matching the argument patterns given by $[\![t_1]\!]^{\$}_{\mathsf{tm}} \dots [\![t_n]\!]^{\$}_{\mathsf{tm}}$:

$$\begin{aligned}&\texttt{symbol } \bar{s}^\star\ \vec{\sigma} : [\![\tau_1]\!]_{\mathsf{tm}} \to \dots \to [\![\tau_n]\!]_{\mathsf{tm}} \to \texttt{TYPE};\\ &\texttt{rule } \bar{s}^\star\ [\![t_1]\!]^{\$}_{\mathsf{tm}} \dots [\![t_n]\!]^{\$}_{\mathsf{tm}} \hookrightarrow [\![\tau^\star]\!]^{\$}_{\mathsf{tm}}\end{aligned}$$

Finally, s is declared as follows:

$$\texttt{constant symbol } \bar{s}\ (\alpha_1 : [\![\tau_1]\!]_{\mathsf{tm}}) \dots (\alpha_n : [\![\tau_n]\!]_{\mathsf{tm}})\ : (\bar{s}^\star\ \alpha_1 \dots \alpha_n)$$

Example 3. Figure 7 shows $[\![\delta]\!]_{\mathsf{cmd}}$ applied to the declarations in figure 4. The polymorphic equality = translates by case 1: its implicit parameter becomes an implicit LambdaPi binder (top). Since `or` has attribute `:right-assoc-nil` `false` (example 2), case 1 also produces a rewrite rule for the nil terminator. The `modus_ponens` rule has no `:args`, so case 1 applies directly: its derived type (example 1) translates to the declaration in the middle section. The `cnf_implies_pos` rule has `:args((=> F1 F2))`, a compound argument pattern. By case 4, an auxiliary symbol is declared whose rewrite rule matches the argument pattern and rewrites to `eo.Proof` φ where $\varphi := [\![\texttt{(or (not (=> F1 F2)) (not F1) F2)}]\!]_{\mathsf{tm}}$; the rule itself takes the argument as an explicit parameter whose type is given by the auxiliary (bottom).

Translating Proofs.

Definition 5. Let π be a proof command. Then $[\![\pi]\!]_{\mathsf{cmd}}$ is the list of LambdaPi commands such that:

```
(declare-const U Type)
(declare-const x0 U)
(define @t1 () (= x0 x0))
(define @t2 () (not @t1))
(assume @p1 @t2)
(step @p2 (= @t1 true) :rule eq-refl :args (x0))
  ...
(step @p6 false :rule eq_resolve :premises (@p1 @p5))
```

```
require open cpc.Cpc;
constant symbol U : Set;
constant symbol x0 : τ U;
symbol {|@t1|} ≔ eo.app (eo.app = x0) x0;
symbol {|@t2|} ≔ eo.app not {|@t1|};
constant symbol {|@p1|} : eo.Proof {|@t2|};
symbol {|@p2|} ≔ eq-refl (x0);
assert ⊢ {|@p2|} : eo.Proof (eo.app (eo.app = {|@t1|}) true);
  ...
symbol {|@p6|} ≔ eq_resolve ({|@p1|}) ({|@p5|});
assert ⊢ {|@p6|} : eo.Proof false;
```

Fig. 8. Eunoia proof script (top) and LambdaPi output (bottom). See example 4.

1. If π is an assumption, then $\pi \vdash s : (\texttt{Proof}\ \varphi)$:

$$\texttt{constant symbol}\ \bar{s} : \texttt{eo.Proof}\ [\![\varphi]\!]_{\texttt{tm}}\ ;$$

2. If π is a step, then $\pi \vdash s : (\texttt{Proof}\ \varphi)$ and $\pi \vdash s \coloneqq (r\ t_1 \ldots t_m\ p_1 \ldots p_n)$:

$$\texttt{symbol}\ \bar{s} \coloneqq \bar{r}\ [\![t_1]\!]_{\texttt{tm}}\ \ldots\ [\![t_m]\!]_{\texttt{tm}}\ \bar{p}_1\ \ldots\ \bar{p}_n\ ;$$
$$\texttt{assert}\ \vdash\ \bar{s} : \texttt{eo.Proof}\ [\![\varphi]\!]_{\texttt{tm}}\ ;$$

   The body applies rule $\bar{r}$ to its arguments then premises. The type is checked separately via `assert`, which uses LambdaPi's type inference to fill implicit arguments.

Example 4. Figure 8 shows an excerpt of a proof script from `QF_UF` that derives `false` by showing `(= x0 x0)` is trivially true (`eq-refl`), then propagating this through negation to reach a contradiction (`eq_resolve`). The preamble declarations translate by definition 4: `U` and `x0` become constants (case 1), while the `define`s `@t1` and `@t2` become symbol definitions (case 2). By case 1 of definition 5, the assumption `@p1` becomes a `constant` symbol of type `eo.Proof` $[\![\varphi]\!]_{\texttt{tm}}$. By case 2, each step is defined by applying the rule to its arguments and premises; the `assert` that follows uses type inference to verify the conclusion.

4 Implementation and Results

Our tool `eo2lp`[6] is an OCaml program (∼5,500 lines) that implements the translation of section 3. The parser is generated by Menhir [32] and the back-end is built against the LambdaPi OCaml API. A hand-written `Prelude.lp` module (∼335 lines) bootstraps the type embedding of section 3: it declares `Set`, `τ`, the function space ⇝, the higher-order application symbol `eo.app`, Boolean logic, arithmetic builtins, and the heterogeneous list type used by quantifier binders.

4.1 CPC-MINI

The Cooperating Proof Calculus (CPC) is the Eunoia signature that formalizes CVC5's proof system. The full CPC comprises 35 core modules (∼11,000 lines of Eunoia) plus 16 expert modules for additional theories (separation logic, bags, finite fields, floating points, transcendentals). To validate our approach on a manageable portion, we define *CPC-MINI*,[7] a fork of the upstream CPC included as a git submodule of `eo2lp`. It is a modified subset of 22 core modules (∼5,000 lines) covering logic, integer arithmetic, arrays, sets, quantifiers, and uninterpreted functions — 6 theory modules (57 constants), 8 rule modules (353 rules, 50 programs), and 7 program modules (38 programs, 36 macros), plus a root module `Cpc`.

CPC-MINI applies three changes to the upstream CPC: (i) unsupported theories are removed; (ii) type parameters are tightened to eliminate mixed-type arithmetic; and (iii) several upstream bugs are fixed (e.g. incorrect type parameters in array rewrites). We intend to contribute (iii) upstream and to extend the translation to the remaining theories and expert modules. All 22 modules pass `lambdapi check` with full subject-reduction verification; the signature is translated in ∼100 ms and checked in ∼160 ms.

4.2 Proof Benchmarks

To evaluate the translation on proofs, we sampled up to 100 unsatisfiable problems from SMT-LIB for each of the 14 fragments covered by CPC-MINI and obtained Eunoia proof scripts from CVC5, yielding 1,036 proofs. Each proof is translated by `eo2lp` and the result is checked by `lambdapi check` with subject-reduction verification, under a 5-second timeout. Table 1 summarizes the results: 1,002 of 1,036 proofs (97%) are successfully translated and verified, with an average checking time of 0.47 s.

The seven equality and array fragments achieve a near-perfect pass rate (582 of 583), with the single failure being a timeout. The seven linear arithmetic fragments pass 420 of 453 proofs (93%): QF_UFIDL, UFLIA, and LIA reach 98–100%; the remaining failures are timeouts in QF_LIA and QF_IDL, where large proof scripts exceed the 5-second limit, and 15 type-checking errors in QF_UFLIA caused by CPC rules absent from CPC-MINI.

[6] https://github.com/deducteam/eo2lp
[7] https://github.com/ciaran-matthew-dunne/cpc-mini

Table 1. Benchmark results on 1,036 CVC5 proofs across 14 SMT-LIB fragments, grouped by theory. Each proof is translated by `eo2lp` and the generated LambdaPi file is checked with subject-reduction verification under a 5 s timeout. Columns: number of proofs (**Tot.**), proofs passing (**Pass**), type-checking errors (**Chk.**), timeouts (**T/O**), average time for passing proofs (**Avg.**).

Fragment	Tot.	Pass	%	Chk.	T/O	Avg. (s)
Equality						
`QF_UF`	100	99	99	0	1	0.37
`UF`	100	100	100	0	0	0.51
Arrays						
`QF_AX`	72	72	100	0	0	0.50
`QF_ALIA`	33	33	100	0	0	0.86
`QF_AUFLIA`	100	100	100	0	0	0.33
`ALIA`	78	78	100	0	0	0.59
`AUFLIA`	100	100	100	0	0	0.40
Linear integer arithmetic						
`QF_IDL`	45	43	96	0	2	0.90
`QF_LIA`	56	42	75	0	14	0.32
`LIA`	100	99	99	0	1	0.44
`QF_UFIDL`	15	15	100	0	0	0.28
`QF_UFLIA`	95	80	84	15	0	0.41
`UFIDL`	42	41	98	1	0	0.49
`UFLIA`	100	100	100	0	0	0.45
Total	**1,036**	**1,002**	**97**	**16**	**18**	**0.47**

5 Conclusion and Future Work

We have presented a translation from Eunoia to the $\lambda\Pi$-calculus modulo rewriting that encodes CVC5's proof system—including inference rules, side-condition programs, and elaboration semantics—automatically, unlike approaches that hand-write lemmas for each rule [17, 31]. The trade-off is that the translated rules are taken as axioms rather than independently proved sound. Our tool `eo2lp` translates all 22 modules of CPC-MINI (359 rules, 96 programs, 57 constants) and independently verifies 97% of 1,036 proof scripts across 14 SMT-LIB fragments.

The most direct future work is extending CPC-MINI to the full CPC (bitvectors, strings, datatypes, reals). Because the encoding lives in LambdaPi, it can serve as a basis for proving the soundness of individual rules directly, or—via the Dedukti ecosystem—for exporting the CPC representation to proof assistants such as Rocq or Isabelle/HOL, analogously to IsaRare [29]. With the full CPC translated, LambdaPi could serve as a robust independent proof checker for CVC5, and a thorough comparison with Ethos—in terms of coverage, performance, and error detection—would help evaluate the practical benefits of this redundancy.

References

[1] Michaël Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. LNCS. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[2] Ali Assaf and Guillaume Burel. "Translating HOL to Dedukti". In: *Fourth International Workshop on Proof eXchange for Theorem Proving*. Ed. by Cezary Kaliszyk and Andrei Paskevich. Vol. 186. EPTCS. 2015, pp. 74–88. DOI: 10.4204/EPTCS.186.8.

[3] Ali Assaf et al. *Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory*. 2023. arXiv: 2311.07185 [cs.LO]. URL: https://arxiv.org/abs/2311.07185.

[4] John Backes et al. *Semantic-based automated reasoning for AWS access policies using SMT*. 2018. URL: https://www.amazon.science/publications/semantic-based-automated-reasoning-for-aws-access-policies-using-smt.

[5] Haniel Barbosa et al. "Flexible Proof Production in an Industrial-Strength SMT Solver". In: *IJCAR*. Ed. by Jasmin Blanchette, Laura Kovács, and Dirk Pattinson. Vol. 13385. LNCS. Springer, 2022, pp. 15–35. DOI: 10.1007/978-3-031-10769-6_3.

[6] Haniel Barbosa et al. "Language and Proofs for Higher-Order SMT (Work in Progress)". In: *PxTP*. Vol. 262. EPTCS. 2017, pp. 15–22. DOI: 10.4204/EPTCS.262.3.

[7] Michael Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *FMCO*. Ed. by Frank S. de Boer et al. Vol. 4111. LNCS. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. "SMT-LIB 3: Bringing Higher-Order Logic to SMT". In: ().

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2024-09-20.pdf.

[10] Clark Barrett et al. "Chapter 33: Satisfiability modulo theories". In: *Frontiers in Artificial Intelligence and Applications* 336 (May 2021), pp. 1267–1329. DOI: 10.3233/FAIA201017.

[11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers". In: *J. Autom. Reason.* 51.1 (2013), pp. 109–128. DOI: 10.1007/S10817-013-9278-5.

[12] Mathieu Boespflug and Guillaume Burel. "CoqInE: Translating the Calculus of Inductive Constructions into the λΠ-calculus Modulo". In: *Second International Workshop on Proof Exchange for Theorem Proving*. Ed. by David Pichardie and Tjark Weber. Vol. 878. CEUR Workshop Proceedings. 2012, pp. 44–50. URL: https://ceur-ws.org/Vol-878/paper3.pdf.

[13] Chad E. Brown. *Notes on Semantics of and Proofs for SMT*. 2021. URL: http://grid01.ciirc.cvut.cz/~chad/smtsempfs.pdf.

[14] Guillaume Burel et al. “First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice”. In: *J. Autom. Reason.* 64.6 (2020), pp. 1001–1050. DOI: 10.1007/s10817-019-09533-z.

[15] Guillaume Bury. “Integrating rewriting, tableau and superposition into SMT”. PhD thesis. Université Sorbonne Paris Cité, Jan. 2019. URL: https://theses.hal.science/tel-02612985.

[16] Adrien Champion et al. “The Kind 2 Model Checker”. In: *CAV*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. LNCS. Springer, 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29.

[17] Alessio Coltellacci, Gilles Dowek, and Stephan Merz. “Reconstruction of SMT Proofs with Lambdapi”. In: *International Workshop on Satisfiability Modulo Theories*. Vol. 3725. CEUR Workshop Proceedings. 2024, pp. 13–23. URL: https://ceur-ws.org/Vol-3725/paper8.pdf.

[18] Alessio Coltellacci and Stephan Merz. “Checking Linear Integer Arithmetic Proofs in Lambdapi”. In: *FroCoS*. Vol. 15979. LNCS. Springer, 2025. DOI: 10.1007/978-3-032-04167-8_20.

[19] Denis Cousineau and Gilles Dowek. “Embedding Pure Type Systems in the lambda-Pi-calculus modulo”. In: *TLCA*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. LNCS. Springer, 2007, pp. 102–117. DOI: 10.1007/978-3-540-73228-0_9.

[20] Burak Ekici et al. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq”. In: *CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7.

[21] *Ethos User Manual*. URL: https://github.com/cvc5/ethos/blob/main/user_manual.md (visited on 08/12/2024).

[22] Gabriel Hondet and Frédéric Blanqui. “The New Rewriting Engine of Dedukti”. In: *FSCD*. 167. June 2020, p. 16. DOI: 10.4230/LIPIcs.FSCD.2020.35.

[23] The SMT-LIB Initiative. *SMT-LIB Verion 3.0 - A Preliminary Proposal*. Dec. 2021. URL: https://smt-lib.org/version3.shtml.

[24] *isabelle_dedukti: Isabelle component exporting Isabelle proofs to Dedukti*. URL: https://github.com/Deducteam/isabelle_dedukti.

[25] Anja Petković Komel, Michael Rawson, and Martin Suda. *Case Study: Verified Vampire Proofs in the LambdaPi-calculus Modulo*. 2025. arXiv: 2503.15541 [cs.LO]. URL: https://arxiv.org/abs/2503.15541.

[26] Nikolai Kosmatov and Julien Signoles. “Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis”. In: *RV*. Ed. by Yliès Falcone and César Sánchez. Vol. 10012. LNCS. Springer, 2016, pp. 92–115. DOI: 10.1007/978-3-319-46982-9_7.

[27] *Krajono: A Matita to Dedukti translator*. URL: https://deducteam.gitlabpages.inria.fr/krajono/.

[28] Hanna Lachnitt et al. “Improving the SMT Proof Reconstruction Pipeline in Isabelle/HOL”. In: *ITP*. Vol. 332. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 26:1–26:19. DOI: 10.4230/LIPIcs.ITP.2025.26.

[29] Hanna Lachnitt et al. “IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL”. In: *TACAS*. Vol. 14572. LNCS. Springer, 2024, pp. 311–330. DOI: 10.1007/978-3-031-57246-3_17.

[30] Cristian Mattarei et al. “CoSA: Integrated Verification for Agile Hardware Design”. In: *FMCAD*. IEEE, 2018. DOI: 10.23919/FMCAD.2018.8603014.

[31] Abdalrhman Mohamed et al. “lean-smt: An SMT Tactic for Discharging Proof Goals in Lean”. In: *CAV*. Vol. 15893. LNCS. Springer, 2025, pp. 196–220. DOI: 10.1007/978-3-031-98682-6_11.

[32] François Pottier and Yann Régis-Gianas. “Menhir Reference Manual”. In: (2024). Version 20240715. URL: http://gallium.inria.fr/~fpottier/menhir/.

[33] Léa Riant. “Debugging Support in Atelier B”. In: *SEFM 2022 Collocated Workshops Revised Selected Papers*. Ed. by Paolo Masci et al. Springer, 2023, pp. 148–155. DOI: 10.1007/978-3-031-26236-4_12.

[34] Neha Rungta. “A Billion SMT Queries a Day”. In: *Computer Aided Verification (CAV)*. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 3–18. DOI: 10.1007/978-3-031-13185-1_1.

[35] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. “Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant”. In: *CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. LNCS. Springer, 2021, pp. 450–467. DOI: 10.1007/978-3-030-79876-5_26.

[36] Hans-Jörg Schurr et al. “Alethe: Towards a Generic SMT Proof Format (Extended Abstract)”. In: *Electronic Proceedings in Theoretical Computer Science* 336 (2021), pp. 49–54. DOI: 10.4204/EPTCS.336.6.

[37] Aaron Stump et al. “SMT proof checking using a logical framework”. In: *Formal Methods Syst. Des.* 42.1 (2013), pp. 91–118. DOI: 10.1007/S10703-012-0163-3.

[38] Geoff Sutcliffe, Frédéric Blanqui, and Guillaume Burel. “Proof Verification with GDV and LambdaPi - It’s a Matter of Trust”. In: *FLAIRS*. Ed. by Douglas A. Talbert and Ismaïl Biskri. Florida Online Journals, 2025. DOI: 10.32473/FLAIRS.38.1.138642.

[39] Rishikesh Vaishnav. “Lean4Less: Eliminating Definitional Equalities from Lean via an Extensional-to-Intensional Translation”. In: *ICTAC 2025*. LNCS. Accepted, to appear. Springer, 2025. URL: https://rish987.github.io/files/lean4less.pdf.