Automatically Translating Proof Systems for SMT Solvers to the λΠ-calculus[⋆]

Ciarán Dunne[1], 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 a logical framework used formalizing the proof production facilities of SMT solvers. We present an encoding of Eunoia signatures and theories into the λΠ-*calculus modulo rewriting* as implemented by the LambdaPi proof assistant. Our encoding is demonstrated by the development of a tool `eo2lp`, which we used for (a) translating the portion of `cvc5`'s *co-operating proof calculus* (CPC) corresponding to the QF-UF fragment of SMT-LIB; and (b) translating proofs produced by running `cvc5` on a set of QF-UF problems from the SMT-LIB benchmark library.

1 Background

The area of automated reasoning research known as *satisfiability modulo theories* (SMT) aims to develop tools capable of deciding the satisfiability of logical specifications within a curated selection of mathematical theories [9]. Such tools (known as SMT *solvers*) are increasingly used as back-ends for various tasks, in particular for hardware verification [26], for program verification [7, 27, 23], for model checking [15], to increase automation in proof assistants [2, 10], and to check the security of access policies [5]. Most of theses applications require a high level of confidence in the answers produced by the solver. For this purpose, the specification of the proof system of the solvers needs to be clearly formalized, and one should be able to check their output using trusted tools.

Towards the aim of standardizing and benchmarking SMT solvers, the *SMT library initiative* oversee the development of the *SMT-LIB standard* [8]. The standard specifies a common language used for interacting with solvers, including a detailed description of the mathematical foundations of the SMT problem.

For specifications deemed unsatisfiable, many solvers can generate *proof certificates* that demonstrate the absurdity of the assertions made by the specification. Proof generation is however not covered by the SMT-LIB standard, which has led to the development of various proof formats for SMT solvers. One of the first attempt to output proofs was done by the solver CVC3 and its successors. They can produce proofs in the LFSC format [30]. LFSC extends the Edinburgh Logical Framework (LF) with side conditions, programs that are executed to restrict applications of some inference rules. However, several proof rules of CVC5 are not

[⋆] Research reported in this document was supported by an Amazon Research Award, 2022.

supported by this back-end, and such steps are given as axioms and need to be checked. Another limitation is that the rules that are used are specific to CVC5, and do not necessarily exists in this form in other solvers. This need for a more interoperable proof format lead to the design of the *Alethe* format [29]. Alethe draws from a fixed set of rules designed to reflect the reasoning mechanisms of SMT solvers. The common proof interface provided by Alethe has enabled interoperability between solvers and other automated reasoning tools; particularly *proof assistants*, where it is used for the reconstruction of proofs obtained automatically [2, 28]. Among solvers that output proofs in the Alethe format, one can cite VeriT [12] and CVC5 [6]. Proofs in the Alethe format can be checked by the standalone tool *Carcara* [1]. Unfortunately, long-term interoperability can be challenging, as developers must rewrite aspects of their tools to maintain parity with the evolving Alethe specification. Furthermore, from the point of view of the developer of a solver, the fixed set of available inference rules of Alethe hinders the design of new reasoning techniques. *Eunoia* was proposed to tackle these issues. *Eunoia* is a logical framework that allows formalizing the inference rules used by the proof production facilities of an SMT solver. Eunoia shares aspects with the speculative proposal for SMT-LIB 3 [20]: it uses e.g. dependent types and binders. Proofs in the Eunoia format can be checked by the C++ tool *Ethos* [18]. Eunoia and Ethos are work in progress, and they are consequently still evolving. CVC5 can output proofs in the Eunoia format. For this purpose, an encoding of the proof calculus of CVC5, namely the Cooperating Proof Calculus (CPC), has been implemented in Eunoia. Using this, proofs of CVC5 can be checked by Ethos.

Carcara and Ethos are relatively small tools, so that their source can be inspected to persuade oneself of their correction. However, one could want to go a step further to gain another level of trust, by translating proofs in Alethe or Eunoia format into formats that are more mature, for instance the format of proof assistants such as Rocq or Isabelle/HOL. Such a translation could also help with reusing the proofs in another context, in an interoperability perspective. The $\lambda\Pi$-calculus modulo rewriting [17], as implemented in the tools Dedukti [4] and LambdaPi [19], was designed to offer a trustworthy proof checker with an emphasis on interoperability of proof systems. Proofs from various systems can be embedded into Dedukti: proofs assistants such as Rocq [11], Isabelle [21], HOL Light [3], Matita [24] and Lean [33]; automated theorem provers such as Zenon Modulo, iProverModulo [13] and ArchSAT [14] and Vampire [22], and more generally any prover outputting proofs in the TPTP format [31]; and even the programming language semantic framework $\mathbb{K}$ [25]. Conversely, some proofs in Dedukti can be exported back into Rocq, Matita, or HOL/Light [32]. Such a formalism seems therefore a prime target for translating proofs of SMT solvers. Coltellacci [16] developed a translation of Alethe proofs into LambdaPi, by adding a new back-end in the Carcara tool. Roughly, to each inference rule of the Alethe specification corresponds a lemma in LambdaPi, and lemmas are combined to reconstruct the proof.

$$t \mathrel{⸬} s \mid (s\ \vec{t}) \quad \text{(terms)} \qquad \rho \mathrel{⸬} (s\ t\ \langle \nu \rangle_?) \quad \text{(parameters)}$$

$$\begin{array}{ll}
\nu \mathrel{⸬} \texttt{:implicit} \mid \texttt{:list} & \text{(var. attributes)} \\
\alpha \mathrel{⸬} \texttt{:right-assoc} \mid \texttt{:right-assoc-nil}\ \langle t \rangle & \text{(const. attributes)} \\
\quad \mid \texttt{:left-assoc} \mid \texttt{:left-assoc-nil}\ \langle t \rangle & \\
\quad \mid \texttt{:chainable}\ \langle s \rangle \mid \texttt{:pairwise}\ \langle s \rangle &
\end{array}$$

$$\begin{array}{ll}
r \mathrel{⸬} (t\ t') & \text{(term pairs)} \\
\delta \mathrel{⸬} (\texttt{declare-const}\ s\ t\ \langle \alpha \rangle_?) & \text{(std. commands)} \\
\quad \mid (\texttt{declare-parameterized-const}\ s\ (\vec{\rho})\ t\ \langle \alpha \rangle_?) & \\
\quad \mid (\texttt{define}\ s\ (\vec{\rho})\ t\ \ \langle \texttt{:type}\ t' \rangle_?) & \\
\quad \mid (\texttt{program}\ s\ (\vec{\rho})\ \texttt{:signature}\ (\vec{t})\ t'\ (\vec{r})) & \\
\quad \mid (\texttt{declare-rule}\ s\ (\vec{\rho}) & \\
\qquad \langle \texttt{:premises}\ (\vec{t}_{\text{prem}}) \rangle_? & \\
\qquad \langle \texttt{:args}\ (\vec{t}_{\text{args}}) \rangle_? & \\
\qquad \langle \texttt{:requires}\ (\vec{r}) \rangle_? & \\
\qquad \texttt{:conclusion}\ t_{\text{conc}}) & \\
\quad \mid (\texttt{include}\ \mu) &
\end{array}$$

$$\begin{array}{ll}
\pi \mathrel{⸬} (\texttt{assume}\ s\ \varphi) & \text{(prf. commands)} \\
\quad \mid (\texttt{step}\ s\ \varphi\ \langle \texttt{:rule}\ s' \rangle\ \langle \texttt{:premises}\ \vec{\psi} \rangle_?\ \langle \texttt{:args}\ \vec{t} \rangle_?) &
\end{array}$$

Fig. 1. Syntax for Eunoia: terms, attributes, and commands.

As Eunoia is a logical framework, the set of rules is not fixed and cannot be implemented once for all in LambdaPi. To be able to translate Eunoia proofs into Dedukti or LambdaPi, one need a way to encode also how inference rules are defined in Eunoia, in order to be as generic as it. This is the purpose of this paper, together with the actual translation of SMT proofs from CVC5 into LambdaPi.

In the next section, we present Eunoia, Ethos and CPC. Section 3 defines the $\lambda\Pi$-calculus modulo rewriting and its implementation in LambdaPi. The translation from Eunoia to LambdaPi is given in Section 4, as well as actual results. We conclude in Section 5 and discuss future work.

2 Eunoia

With respect to a fixed set $\mathcal{S}_{\mathbf{eo}}$ of *symbols*, the rules in figure 1 define syntax for a fragment of Eunoia. In particular, we define sets of expressions for *terms*, *parameters*, and *attributes*. Each term is either a symbol s or an *application* $(s\ \vec{t})$ for some list of terms $\vec{t}$. Let ν and α range over *variable attributes* and *constant attributes* respectively. Then, each *parameter* ρ consists of a symbol s, a term t (the type of the paramater), and possibly a variable attribute ν.

The rules of figure 1 define a subset of Eunoia *commands*, which 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. The preamble of Υ should be understood as the encoding of an 'input problem', and the body understood as a proof of the unsatisfiability of the problem.

In practice, a Eunoia-friendly solver should have a trusted signature Δ declaring a bespoke set of constants and inference rules. The correctness of a generated proof script Υ may then be checked with respect to Δ using the Ethos tool.

2.1 Commands and their Declarations

We proceed to define an abstract interface for 'reading' information from signatures and proof scripts. This interface is useful for characterizing the *elaboration* and *translation* operators found in section 2.2 and section 4 respectively.

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 a 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` $(t_1 \dots t_n)\, t'$, and cases $c_1 \dots c_m$. Then, δ declares the type and definiens of s as follows, where $t_1 \dots t_n$ are the *domain* types of s and t' is *range*:

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

Inference Rule Declaration. Let δ be a *rule declaration* with symbol s, parameters $\vec{\rho}$, and *conclusion* φ. Also, suppose δ provides `:premises` $(\psi_1 \dots \psi_n)$, `:args` $(t_1 \dots t_m)$ with types $\tau_1 \dots \tau_m$, and `:requires` $((x_1\ y_1) \dots (x_o\ y_o))$. First, let δ declare the type and definiens of an *auxiliary symbol* for s thus:

$$\begin{aligned}&\delta \vdash s^\star(\vec{\rho}) : (\texttt{->}\ \tau_1 \dots \tau_m\ \texttt{Bool})\\ &\delta \vdash s^\star(\vec{\rho}) := \textbf{cases}[\ ((s^\star\ t_1 \dots t_m)\ \varphi) \mid (x_1\ y_1) \dots (x_k\ y_k)\ ]\end{aligned}$$

Then, the type of s is given by δ with an extended list of parameters[4] thus:

$$\delta \vdash s(\vec{\rho}, (\alpha_1\ \tau_1) \dots (\alpha_n\ \tau_m)) : (\texttt{->}\ (\texttt{Proof}\ \psi_1) \dots (\texttt{Proof}\ \psi_n)\ (\texttt{Proof}\ \varphi^\star))$$

[4] The symbols chosen for $\alpha_1 \dots \alpha_m$ must be *fresh* with respect to δ. That is, each α_i is distinct from any symbol occurring in any of the terms or parameters supplied by δ.

where $\varphi^\star := (s^\star\ \alpha_1 \ldots \alpha_n)$ ensures that a term of type $(\texttt{Proof}\ \varphi)$ may only be obtained iff $(\alpha_i = t_i)$ for $1 \leq i \leq m$ and $(x_j = y_j)$ for $1 \leq j \leq o$.

Signature Inclusion. Hereinafter, let Θ be a *(global) environment* mapping from filepaths to Eunoia signatures. Let δ be an inclusion with valid filepath μ. Then, any judgement J made by a command δ' in Θ_μ is also made by δ. That is:

$$\forall\, \delta' \in \Theta_\mu, \quad \delta' \vdash J \quad \implies \quad (\texttt{include}\ \mu) \vdash J$$

Proof Scripts. Two basic forms of *proof commands* are given, which are called *assumption* and *step* respectively. Given $\pi = (\texttt{assume}\ s\ \varphi)$ or $\pi = (\texttt{step}\ s\ \varphi\ \ldots)$, we call s the *name* of π and φ the *conclusion*. In either case, we may write:

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

Furthermore, let π be a step with `:rule` s', `:premises` $(\vec{p})$, and `:args` $(\vec{t})$. Then, π judges the definiens of s as the application of s' to the premises $p_1 \ldots p_n$ and arguments $t_1 \ldots t_m$, i.e.;

$$\pi \vdash s := (s'\ t_1 \ldots t_m\ p_1 \ldots p_n)$$

2.2 Elaboration of Terms

In Eunoia, the ‘de-sugared’ meaning of an application $(s\ \vec{t})$ depends on the constant attribute assigned to s within some signature Δ. For example, consider a constant declaration δ with:

$$\delta \vdash \texttt{or} : (\texttt{->}\ \texttt{Bool}\ \texttt{Bool}\ \texttt{Bool}) \quad \text{and} \quad \delta \vdash \texttt{or} :: \texttt{:right-assoc-nil}\ \texttt{false}$$

From the type of `or`, we may expect its only valid uses to be of the form $(\texttt{or}\ t_1\ t_2)$. However, the assignment of the constant attribute means that the `or` symbol is treated as *right-associative* with *nil-terminator* `false`. Thus, n-ary applications of `or` are *elaborated* to a normal form, e.g.;

$$(\texttt{or x y z}) \dashrightarrow (\texttt{or x (or y (or z false))})$$

Furthermore, the elaboration of such applications may also depend on the attributes of ‘locally bound’ symbols. For example, consider $\vec{\rho}$ containing parameters `(x Bool)`, `(y Bool :list)`, and `(z Bool)`. Then, observe:

$$(\texttt{or x y z}) \dashrightarrow (\texttt{or x (eo::concat or y (or z false))})$$

The `:list` attribute alters the elaboration strategy under the assumption that y will (eventually) be substituted for some `or`-list. It may be helpful for the reader to consider the result of substituting $\texttt{y} \mapsto (\texttt{or w false})$ thus:

$$\begin{gathered}(\texttt{or x (eo::concat or (or w false) (or z false))}) \\ \Big\downarrow \\ (\texttt{or x (or w (or z false))})\end{gathered}$$

With the aim of supporting the elaboration strategies corresponding to the constant attributes given in figure 1, we define an *elaboration* operator below.

Definition 1. *For any symbol f and list of parameters $\vec{\rho}$, let $\mathbf{glue}_{(\vec{\rho},f)}$ be the binary operator such that for any terms t_1, t_2, the following holds:*

$$\mathbf{glue}_{(\vec{\rho},f)}[t_1, t_2] = \begin{cases} \text{(}\texttt{eo::concat}\ f\ t_1\ t_2\text{)} & \textit{if } \vec{\rho} \vdash t_1 :: \texttt{:list}, \\ (f\ t_1\ t_2) & \textit{otherwise.} \end{cases}$$

Then for any signature Δ, let $\mathbf{elab}_{(\Delta,\vec{\rho})}$ be the least (unary) operator such that for any symbol f and terms $\vec{t} = t_1 \dots t_n$, the following holds:

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

$$\mathbf{elab}_{(\Delta,\vec{\rho})}[(f\ \vec{t})] = \begin{cases} \mathbf{foldr}(G, t_{nil}, \vec{t}') & \textit{if } \Delta \vdash f :: \texttt{:right-assoc-nil}\ t_{nil}, \\ \mathbf{foldl}(G', t_{nil}, \vec{t}') & \textit{if } \Delta \vdash f :: \texttt{:left-assoc-nil}\ t_{nil}, \\ \mathbf{foldr}(G, t'_n, t'_1 \dots t'_{n-1}) & \textit{if } \Delta \vdash f :: \texttt{:right-assoc}, \\ \mathbf{foldl}(G', t'_1, t'_2 \dots t'_n) & \textit{if } \Delta \vdash f :: \texttt{:left-assoc}, \\ (f\ t'_1 \dots t'_n) & \textit{otherwise.} \end{cases}$$

where $G := \mathbf{glue}_{(\vec{\rho},f)}[x, y]$, and $G'(x, y) := G(y, x)$, and $t'_i := \mathbf{elab}_{(\Delta,\vec{\rho})}[t_i]$.

$\mathbf{foldl}(G, a, l)$ (resp. $\mathbf{foldr}(G, a, l)$) is the standard left fold (resp. right fold) using combining function G and accumulator a on list l.

3 λΠ-calculus modulo rewriting

Figure 2 provides abstract syntax for the λΠ-*calculus modulo rewriting.* In particular, the rules in figure 2 define the *terms* of the λΠ-calculus. Each term is either a *variable* x, a *constant* κ, a *universe* from $\{\texttt{type}, \texttt{kind}\}$, an *application* of two terms $(t \cdot t')$, 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 the usual definitions for *substitution* $(t[x \mapsto t'])$ and β-*reduction* $(t \rightsquigarrow_\beta t')$.

A *typing* is an expression of the form $(t : t')$ for some terms t, t', and a *context* is a list of typings of the form $(x : t)$ for some variable x and term t. A *rewrite rule* is an expression of the form $(\ell \hookrightarrow r)$, where ℓ and r are terms such that ℓ has the form $((\kappa \cdot t_1) \cdot \dots \cdot t_n)$ for some terms $t_1 \dots t_n$. A *signature* is a list of typings and rewrite rules, where each typing has the form $(\kappa : t)$ for some constant κ and term t with no free variables. Given some 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* $(\equiv_\Sigma)$ is defined as the least equivalence relation containing R_Σ and the β-reduction relation. Furthermore, the rules in figure 2 provide a (mutually inductive) definition of a *well-formedness* relation (wf) on contexts and a *typing relation* $(\vdash_\Sigma)$ between contexts and typings, where $(\Gamma \vdash_\Sigma e : t)$ may be read as "Γ proves e has type t with respect to Σ".

$$\begin{array}{lr} \mu \mathrel{::=} \texttt{type} \mid \texttt{kind} & \text{(universes)} \\ t \mathrel{::=} x \mid \kappa \mid \mu \mid (t \cdot t') \mid (\lambda\, x : t.\, t') \mid (\Pi\, x : t.\, t') & \text{(terms)} \end{array}$$

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

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

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

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

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

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

Fig. 2. Syntax and typing rules for the λΠ-calculus.

$$\begin{array}{lr} \rho \mathrel{::=} (s : t) \mid [s : t] & \text{(parameters)} \\ t \mathrel{::=} s \mid [t] \mid (t \cdot t') \mid (\lambda\, \rho.\, t) \mid (\Pi\, \rho.\, t) & \text{(terms)} \end{array}$$

$$\begin{array}{lr} \theta \mathrel{::=} \texttt{\$x} \mid s\ \langle\theta\rangle_* & \text{(patterns)} \\ r \mathrel{::=} (s\ \langle\theta\rangle_* \hookrightarrow \theta') & \text{(rewrite rules)} \end{array}$$

$$\begin{array}{lr} m \mathrel{::=} \texttt{constant} \mid \texttt{sequential} \mid \texttt{injective} & \text{(modifiers)} \\ c \mathrel{::=} \langle m\rangle_?\ \texttt{symbol}\ s\ \langle\rho\rangle_* : t\ \langle := t'\rangle_?; & \text{(commands)} \\ \quad \mid \texttt{rule}\ r\ \langle\texttt{with}\ r'\rangle_*; & \\ \quad \mid \texttt{require open}\ \langle\mu\rangle_+; & \end{array}$$

Fig. 3. Syntax for the LambdaPi proof assistant.

3.1 The LambdaPi Proof Assistant

Figure 3 gives the syntax for a fragment of the LambdaPi proof assistant. Note that the set of terms differs from that of the 'pure' λΠ-calculus presented earlier. The main differences are made to support *implicit bindings*, which allow the user to omit some subterms and have them automatically 'inferred' by LambdaPi. To facilitate this, abstractions now bind a *parameter* which may be either *explicit* or *implicit* (written $(s : t)$ and $[s : t]$ resp.), and we have terms of the form $[t]$

which are said to be *explicated*. Hereinafter, we may write $(t_1 \to t_2)$ for the term $(\Pi\,(x : t_1).\, t_2)$, where x is some 'fresh' symbol that does not occur free in t_2.

We also introduce a syntax of *patterns* which is used within declarations of rewrite rules. Each pattern is either a *pattern variable* (`$x`) or a *pattern application* $(s\ \langle p\rangle_*)$, where s is called the *head* of the pattern. Hereinafter, let $\mathbf{pvars}(p)$ denote the set of pattern variables occurring in some pattern p.

A LambdaPi *file* is a list of *commands*, which should be seen as a representation of a $\lambda\Pi$-signature. To make this correspondence a bit clearer, the syntax and behaviour of each command is discussed in the following passages of text:

Symbol Declaration. Each *symbol declaration* consists of a symbol s, a list of parameters $\vec{\rho}$, and a term t. In general, this has the effect of adding $(s : (\Pi\,\rho_1.\ \ldots\ \Pi\,\rho_n.\, t))$ to the signature. Optionally, a *modifier* may be given. The (`constant`) modifier forbids the user from later adding rewrite rules with s at the head, and (`sequential`) alters the rewriting strategy of LambdaPi so that rewrite rules with head s are 'matched' in the order they are given in the file. A *definition* $(\coloneqq t)$ may be given when the (`constant`) modifier is not present, which has the effect of adding the rewrite rule $(s\,\vec{\rho} \hookrightarrow t)$ to the signature.

Rewrite Rule Declaration. Rewrite rules are introduced with the (`rule`) command. For user convenience, a list of rewrite rules may be introduced using the keyword (`with`). In practice, a rewrite rule $(p \hookrightarrow p')$ will only be accepted by LambdaPi if p is a pattern application and $\mathbf{pvars}(p') \subseteq \mathbf{pvars}(p)$. The rewrite rule(s) will be added to the signature in this case, effectively augmenting the typechecking procedure of LambdaPi to behave 'modulo' those rule(s).

Theory Import. LambdaPi uses a lightweight module system which allows users to develop formalizations across multiple files, possibly making use of third-party libraries. Within this document, we use the command (`require open` $\vec{\mu}$) where $\vec{\mu}$ is a list of (.)-delimited *paths*. This command has the effect of importing all symbol declarations and rewrite rules from the specified files.

Type Universes. Because the type system of LambdaPi does not allow 'quantifying' over types (i.e., (`type` $\to$ `type`) is not well-typed), most LambdaPi developments make use of 'Tarski-style' universes to support treating types as 'first-class citizens'. In particular, we use the following symbol declarations:

```
symbol Set  : type;
symbol El   : Set → type;
symbol (⤳) : Set → Set → Set;
```

We work 'within' Set when embedding 'many-sorted' logics in LambdaPi. Namely, the translation outlined in this document identifies Eunoia's types with terms of type Set and uses El to 'lift' these to top-level LambdaPi types. Moreover, the (infix) symbol $(\rightsquigarrow)$ is used as a 'set-level' type constructor, where the following rewrite rule specifies how these sets are 'lifted'.

$$\texttt{rule}\ \mathsf{El}\ (\$\alpha \rightsquigarrow \$\beta) \hookrightarrow (\mathsf{El}\ \$\alpha \to \mathsf{El}\ \$\beta);$$

Implicit Parameters. As mentioned above, the presence of 'implicit' bindings in abstractions can enable the user to omit some terms in applications, and have them automatically 'inferred' by LambdaPi. To make this notion more precise, consider the following symbol declaration:

$$\texttt{symbol}\ \mathsf{foo}\ [\alpha : \mathsf{Set}] : \texttt{El}\ (\alpha \rightsquigarrow \alpha \rightsquigarrow \alpha);$$

Modulo rewriting, the type of foo is $(\Pi\, [\alpha : \texttt{Set}].\, \texttt{El}\ \alpha \to \texttt{El}\ \alpha \to \texttt{El}\ \alpha)$. Because this type contains implicit bindings, LambdaPi also registers an 'explicit version' of foo named @foo whose type contains only explicit bindings. Hereinafter, an application like $(\mathsf{foo}\ x\ y)$ generates a 'schematic term' $(\texttt{@}\mathsf{foo}\ ?\alpha\ x\ y)$ with constraints $\{?\alpha : \texttt{Set},\ x : \texttt{El}\ ?\alpha,\ y : \texttt{El}\ ?\alpha\}$. Given a suitable context, LambdaPi is able to 'solve' these constraints to infer a 'concrete' value for $?\alpha$. Alternatively, a user can 'force' values that would otherwise be automatically inferred by using explicated terms (e.g., $(\mathsf{foo}\ [\mathbb{N}]\ x\ y)$ is equivalent to $(\texttt{@}\mathsf{foo}\ \mathbb{N}\ x\ y)$).

4 Translation and Results

Recall the definition of Eunoia terms and commands given by section 2, and similarly those of LambdaPi from section 3. We define a *translation* operator below, which may act on the terms, types, and commands of Eunoia. First, we define an injection from Eunoia symbols to those of LambdaPi.

Definition 2. *For any Eunoia symbol $s \in \mathcal{S}_{\mathsf{eo}}$, define $\bar{s}$ thus:*

$$\bar{s} = \begin{cases} \{\!|s|\!\} & \text{if } s \text{ contains any of } \{\$, @, \ldots\}, \\ s & \text{otherwise.} \end{cases}$$

Definition 3. *Let $[\![\cdot]\!]_{\mathsf{tm}}$ be the least operator such that $[\![s]\!]_{\mathsf{tm}} = \bar{s}$ and:*

$$\left[\!\left[\,(s\ \vec{t})\,\right]\!\right]_{\mathsf{tm}} = \begin{cases} [\![t_1]\!]_{\mathsf{tm}} \rightsquigarrow \ldots \rightsquigarrow [\![t_n]\!]_{\mathsf{ty}} & \text{if } s = (\texttt{->}), \\ ((\bar{s} \cdot [\![t_1]\!]_{\mathsf{tm}}) \ldots \cdot [\![t_n]\!]_{\mathsf{tm}}) & \text{otherwise.} \end{cases}$$

Furthermore, let $[\![\cdot]\!]_{\mathsf{ty}}$ be the least operator such that:

$$[\![s]\!]_{\mathsf{ty}} = \begin{cases} \mathtt{Set} & \text{if } s = \mathtt{Type}, \\ \mathtt{El}\ [\![s]\!]_{\mathsf{tm}} & \text{otherwise.} \end{cases}$$

$$\left[\!\left[\,(s\ \vec{t})\,\right]\!\right]_{\mathsf{ty}} = \begin{cases} [\![t_1]\!]_{\mathsf{ty}} \to \ldots \to [\![t_n]\!]_{\mathsf{ty}} & \text{if } s = (\texttt{->}), \\ \mathtt{El}\ \left[\!\left[\,(s\ \vec{t})\,\right]\!\right]_{\mathsf{tm}} & \text{otherwise.} \end{cases}$$

Now, recall the abstract interface for Eunoia commands defined in section 2.1, and also those of LambdaPi.

Definition 4. *Let $[\![\cdot]\!]_{\mathsf{cmd}}$ be the least operator such that for any standard command δ, $[\![\delta]\!]_{\mathsf{cmd}}$ is a set of LambdaPi commands satisfying the following:*

$$\delta \vdash s(\vec{\rho}) : t \quad \Longrightarrow \quad \begin{cases} (\texttt{symbol}\ \bar{s}\ [\![\vec{\rho}]\!]_{\mathsf{ctx}} : [\![t]\!]_{\mathsf{ty}} \coloneqq [\![t']\!]_{\mathsf{tm}}) \in [\![\delta]\!]_{\mathsf{cmd}} & \textit{if } \delta \vdash s(\vec{\rho}) \coloneqq t', \\ (\texttt{symbol}\ \bar{s}\ [\![\vec{\rho}]\!]_{\mathsf{ctx}} : [\![t]\!]_{\mathsf{ty}}) \in [\![\delta]\!]_{\mathsf{cmd}} & \textit{otherwise.} \end{cases}$$

$$\delta = (\texttt{include}\ \mu) \quad \Longrightarrow \quad [\![\delta]\!]_{\mathsf{cmd}} = \{(\texttt{require open}\ \mu)\}$$

The translation on proof commands $[\![\pi]\!]_{\mathsf{cmd}}$ is defined similarly, where assumptions and steps are mapped to symbol declarations in LambdaPi that reflect their names, conclusions, and (if applicable) rules, premises, and arguments, using the judgements $\pi \vdash s : (\texttt{Proof}\ \varphi)$ and $\pi \vdash s \coloneqq (s'\ t_1 \dots t_m\ p_1 \dots p_n)$.

We have implemented this translation as an OCaml program called `eo2lp`, which uses the Menhir parser generator for parsing Eunoia signatures and proof scripts. The tool reads Eunoia files, elaborates them using the operator from section 2.2, applies the translation operators defined above, and outputs corresponding LambdaPi code.

Testing and Benchmarks. Because Eunoia is a complex and evolving system, we created a fork of the CPC signature called CPC-MINI, which corresponds to the fragment of CPC needed for formalizing the constants and inference rules used in cvc5 proofs whose input problems are from the QF-UF fragment of SMT-LIB. We translated all of CPC-MINI into LambdaPi code, mirroring the directory tree of CPC-MINI.

For proof scripts, we used a small benchmark library (called 'rodin') of 30 unsatisfiable problems from the SMT-LIB benchmark suite, restricted to the QF-UF fragment. We ran cvc5 on these problems with the option `--proof-format=cpc` to dump their proofs in Eunoia format. We then ran our `eo2lp` tool on these proof scripts and obtained LambdaPi files that typecheck successfully.

5 Conclusion and Future Work

We have presented a translation from Eunoia specifications and proofs to the $\lambda\Pi$-calculus, as implemented in the LambdaPi proof assistant. Our tool `eo2lp` automates this translation and has been demonstrated on a subset of the CPC signature and proofs from QF-UF benchmarks.

It should be stressed that the presentation of Eunoia in this paper only represents the core features, and there is more work to do in order to (a) translate all of the CPC signature and therefore be able to translate arbitrary cvc5 proofs, and (b) robustly cover the entirety of Eunoia to be able to translate arbitrary Eunoia signatures and proof scripts.

We should also aim to support larger proofs, as those from the rodin benchmark have fewer than 70 proof steps. Some cvc5 proofs can be massive, so computational and efficiency aspects of the translation must be investigated.

For future work, we should also investigate using our translated signatures and proof scripts in the realm of proof interoperability. For example, to gain even more confidence in the results of cvc5, one could attempt to prove the consistency of CPC, either within LambdaPi or by exporting to some other proof assistant like Isabelle/HOL or Rocq.

References

[1] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. "Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format". In: *TACAS*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. LNCS. Springer, 2023, pp. 367–386. DOI: `10.1007/978-3-031-30823-9_19`.

[2] 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`.

[3] 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`.

[4] 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`.

[5] 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`.

[6] 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`.

[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. *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`.

[9] 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`.

[10] 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`.

[11] Mathieu Boespflug and Guillaume Burel. "CoqInE: Translating the Calculus of Inductive Constructions into the $\lambda\Pi$-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`.

[12] Thomas Bouton et al. "veriT: An Open, Trustable and Efficient SMT-Solver". In: *CADE-22*. Ed. by Renate A. Schmidt. Vol. 5663. LNCS. Springer, 2009, pp. 151–156. DOI: `10.1007/978-3-642-02959-2_12`.

[13] 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`.

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

[15] 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`.

[16] 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`.

[17] 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`.

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

[19] 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`.

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

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

[22] 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`.

[23] 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`.

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

[25] Amélie Ledein, Valentin Blot, and Catherine Dubois. “A Semantics of $\mathbb{K}$ into Dedukti”. In: *TYPES post-proceedings.* Ed. by Delia Kesner and Pierre-Marie Pédrot. Vol. 269. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 12:1–12:22. DOI: 10.4230/LIPICS.TYPES.2022.12.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] François Thiré. “Sharing a Library between Proof Assistants: Reaching out to the HOL Family”. In: *LFMTP.* Ed. by Frédéric Blanqui and Giselle Reis. Vol. 274. EPTCS. 2018, pp. 57–71. DOI: 10.4204/EPTCS.274.5.

[33] 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.