`eo2lp` — *from Eunoia to LambdaPi*

September 11, 2025

Ciarán Dunne and Guillaume Burel

ENS Paris-Saclay, INRIA

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.

🇺🇸 development led by Andrew Reynolds, at University of Iowa.

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  - 🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  - 👻 the 'spiritual successor' of the Alethe proof format.

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  - 🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  - 👻 the 'spiritual successor' of the Alethe proof format.
  - 📜 covers theory signatures & proof scripts.

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.

🇺🇸 development led by Andrew Reynolds, at University of Iowa.
👻 the ‘spiritual successor’ of the Alethe proof format.
📜 covers theory signatures & proof scripts.
✔️ paired with the Ethos checker.

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  👻 the 'spiritual successor' of the Alethe proof format.
  📜 covers theory signatures & proof scripts.
  ✔️ paired with the Ethos checker.
- Extends SMT-LIB by adding:

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  - 🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  - 👻 the 'spiritual successor' of the Alethe proof format.
  - 📜 covers theory signatures & proof scripts.
  - ✔️ paired with the Ethos checker.
- Extends SMT-LIB by adding:
  - (dependent) types, parametric polymorphism,

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  👻 the 'spiritual successor' of the Alethe proof format.
  📜 covers theory signatures & proof scripts.
  ✔️ paired with the Ethos checker.
- Extends SMT-LIB by adding:
  - (dependent) types, parametric polymorphism,
  - 'programs' (i.e., constants with rewrite rules),

Eunoia and Ethos

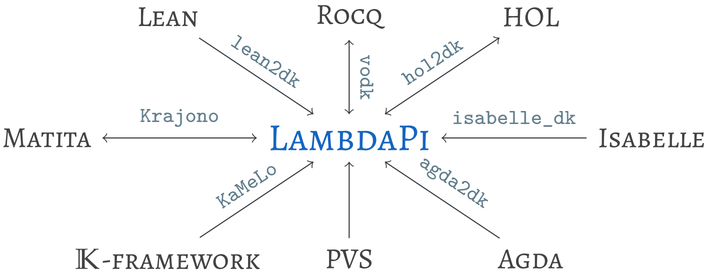
- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  👻 the 'spiritual successor' of the Alethe proof format.
  📜 covers theory signatures & proof scripts.
  ✔️ paired with the Ethos checker.
- Extends SMT-LIB by adding:
  - (dependent) types, parametric polymorphism,
  - 'programs' (i.e., constants with rewrite rules),
  - inference rule declarations,

Eunoia and Ethos

- Eunoia is an emerging *logical framework* aimed at formalizing the proof systems used by SMT solvers.
  🇺🇸 development led by Andrew Reynolds, at University of Iowa.
  👻 the 'spiritual successor' of the Alethe proof format.
  📜 covers theory signatures & proof scripts.
  ✔️ paired with the Ethos checker.
- Extends SMT-LIB by adding:
  - (dependent) types, parametric polymorphism,
  - 'programs' (i.e., constants with rewrite rules),
  - inference rule declarations,
  - commands for building proof scripts.

LambdaPi

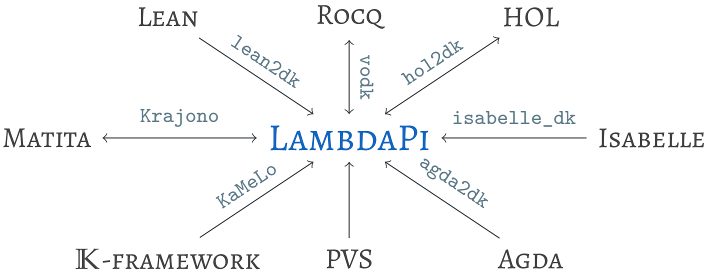
- Logical framework based on the λΠ-*calculus modulo rewriting*.



LambdaPi

- Logical framework based on the λΠ-*calculus modulo rewriting*.

🇫🇷 development led by Frédéric Blanqui, INRIA Paris-Saclay

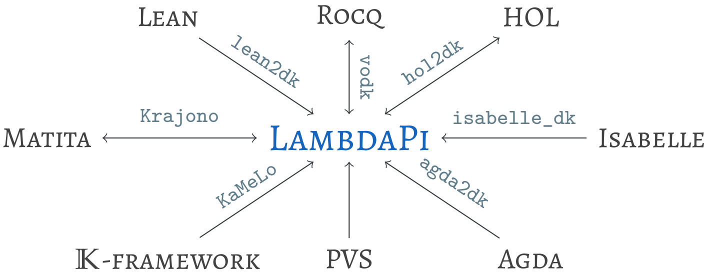


LambdaPi

- Logical framework based on the λΠ-*calculus modulo rewriting*.

🇫🇷 development led by Frédéric Blanqui, INRIA Paris-Saclay

🔒 small code base, trusted foundations.



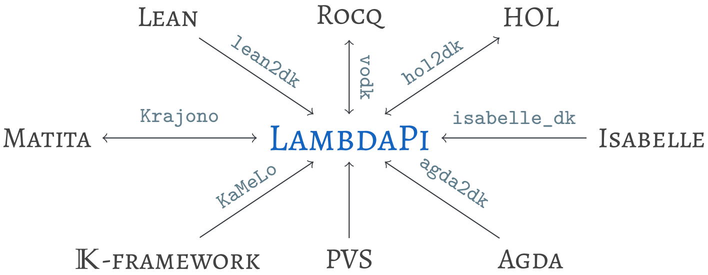
LambdaPi

- Logical framework based on the λΠ-*calculus modulo rewriting*.

🇫🇷 development led by Frédéric Blanqui, INRIA Paris-Saclay

🔒 small code base, trusted foundations.

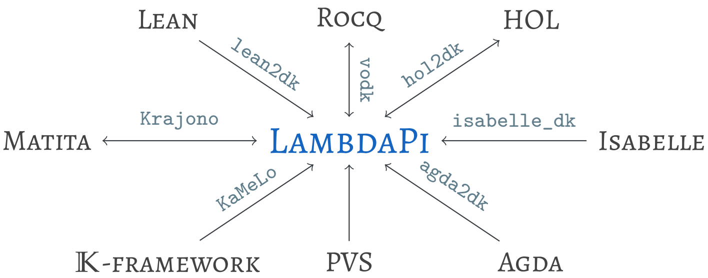
🐆 fast typechecker.



LambdaPi

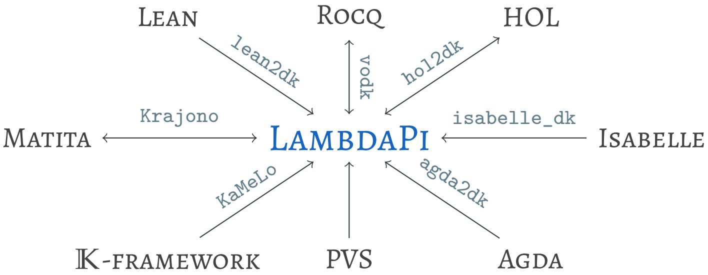
- Logical framework based on the λΠ-*calculus modulo rewriting*.

🇫🇷 development led by Frédéric Blanqui, INRIA Paris-Saclay
🔒 small code base, trusted foundations.
🐆 fast typechecker.
🤖 interactive theorem proving via LSP!



LambdaPi

- Logical framework based on the λΠ-*calculus modulo rewriting*.
  - 🇫🇷 development led by Frédéric Blanqui, INRIA Paris-Saclay
  - 🔒 small code base, trusted foundations.
  - 🐆 fast typechecker.
  - 🤖 interactive theorem proving via LSP!
- Primarily focused on proof assistant interoperability.



The Co-operating Proof Calculus

- The co-operating proof calculus (CPC) is cvc5's proof system.

🧩 some rules take arguments, some have side-conditions.

The Co-operating Proof Calculus

- The co-operating proof calculus (CPC) is cvc5's proof system.

✨ formalized as a Eunoia signature Σ_{CPC}.

🧩 some rules take arguments, some have side-conditions.

The Co-operating Proof Calculus

- The co-operating proof calculus (CPC) is cvc5's proof system.
  - ✨ formalized as a Eunoia signature Σ_{CPC}.
  - 🐘 not small (> 600 inference rules).
  - 🧩 some rules take arguments, some have side-conditions.

The Co-operating Proof Calculus

- The co-operating proof calculus (CPC) is cvc5's proof system.
  - ✨ formalized as a Eunoia signature Σ_{CPC}.
  - 🐘 not small (> 600 inference rules).
  - 🧩 some rules take arguments, some have side-conditions.
- Proofs produced by cvc5 are Eunoia proof scripts that exclusively use the rules from Σ_{CPC}.

Example. A CPC rule for elimination on n-ary conjunctions, where $\varphi_1 \ldots \varphi_n$ are formulas and $i \in \mathbb{N}$.

$$\frac{(\varphi_1 \wedge \ldots \wedge \varphi_n) \mid i}{\varphi_i} \quad \text{(and\_elim)}$$

Example. A CPC rule for elimination on n-ary conjunctions, where $\varphi_1 \ldots \varphi_n$ are formulas and $i \in \mathbb{N}$.

$$\frac{(\varphi_1 \wedge \ldots \wedge \varphi_n) \mid i}{\varphi_i} \quad \text{(and\_elim)}$$

The rule is formalized in Eunoia thus:

```
(declare-rule and_elim ((Fs Bool) (i Int))
    :premises (Fs)
    :args (i)
    :conclusion (eo::list_nth and Fs i)
)
```

Example. The following problem is unsatisfiable. In this case, `cvc5` can provide a proof demonstrating this.

Example. The following problem is unsatisfiable. In this case, cvc5 can provide a proof demonstrating this.

```
(set-logic QF_UF)
(set-option :produce-proofs true)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat)
(get-proof)
(exit)
```

Example. The following problem is unsatisfiable. In this case, cvc5 can provide a proof demonstrating this.

```
(set-logic QF_UF)
(set-option :produce-proofs true)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat)
(get-proof)
(exit)
```

```
unsat
(declare-fun p () Bool)
(assume @p1 (and p (not p)))
(step @p2 :rule and_elim :premises (@p1) :args (1))
(step @p3 :rule and_elim :premises (@p1) :args (0))
(step @p4 false :rule contra :premises (@p3 @p2))
```

- *Goal:* Design a translation procedure T such that;

- *Goal:* Design a translation procedure T such that;
  - if Σ is a Eunoia signature implementing some logic L,

- *Goal:* Design a translation procedure T such that;
  - if Σ is a Eunoia signature implementing some logic L,
  - then $T(\Sigma)$ is a LambdaPi signature also implementing L.

- *Goal:* Design a translation procedure T such that;
  - if Σ is a Eunoia signature implementing some logic L,
  - then $T(\Sigma)$ is a LambdaPi signature also implementing L.
- Thus, if Π is a valid Eunoia proof script depending on Σ, then $T(\Pi)$ should be well-typed wrt. $T(\Sigma)$.

Eunoia

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.
- Define **eo** as the set of Eunoia expressions thus:

$$
\begin{aligned}
e \in \textbf{eo} ::= {} & s && \text{(symbol)} \\
\mid {} & (s\ e_1\ \ldots\ e_n) && \text{(application)}
\end{aligned}
$$

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.
- Define **eo** as the set of Eunoia expressions thus:

$$
\begin{aligned}
e \in \textbf{eo} ::= {} & s && \text{(symbol)} \\
\mid {} & (s\ e_1\ \ldots\ e_n) && \text{(application)}
\end{aligned}
$$

- In general, expressions are either:

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.
- Define **eo** as the set of Eunoia expressions thus:

$$
\begin{aligned}
e \in \textbf{eo} ::= {} & s && \text{(symbol)} \\
\mid {} & (s\ e_1\ \ldots\ e_n) && \text{(application)}
\end{aligned}
$$

- In general, expressions are either:
  - terms (e.g., `true`, `false`)

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.
- Define **eo** as the set of Eunoia expressions thus:

$$\begin{aligned} e \in \mathbf{eo} ::= {} & s && \text{(symbol)} \\ \mid {} & (s\ e_1\ \ldots\ e_n) && \text{(application)} \end{aligned}$$

- In general, expressions are either:
  - terms (e.g., `true`, `false`)
  - types (e.g., `Bool`, `(-> Bool Bool)`)

- Fix a set of symbols $\mathcal{S}$, and let $s \in \mathcal{S}$.
- Define **eo** as the set of Eunoia expressions thus:

$$\begin{aligned} e \in \mathbf{eo} ::= {} & s && \text{(symbol)} \\ \mid {} & (s\ e_1\ \ldots\ e_n) && \text{(application)} \end{aligned}$$

- In general, expressions are either:
  - terms (e.g., `true`, `false`)
  - types (e.g., `Bool`, `(-> Bool Bool)`)
  - kinds (e.g., `Type`, `(-> Type Type)`)

Eunoia has type declarations.

$$(\texttt{declare-type}\ s\ (e_1\ \ldots\ e_n))$$

Example. The `Array` symbol declared a (binary) type constructor.

```
(declare-type Array (Type Type))
```

Eunoia has constant declarations of the form:

$$(\texttt{declare-const}\ s\ e\ \langle\alpha\rangle_?)$$

where α is a constant attribute. i.e.,

$$\begin{aligned}\alpha \in \mathbf{attr_c} ::= &\ \texttt{:right-assoc} \mid \texttt{:right-assoc-nil}\langle t\rangle \\ \mid &\ \texttt{:left-assoc} \mid \texttt{:left-assoc-nil}\langle t\rangle \\ \mid &\ \texttt{:chainable}\langle s\rangle \mid \texttt{:pairwise}\langle s\rangle \mid \texttt{:binder}\langle s\rangle\end{aligned}$$

Example. Declare `and` right-associative, with *nil terminator* `true`.

```
(declare-const and (-> Bool Bool Bool)
  :right-assoc-nil true
)
```

Example. Declare `and` right-associative, with *nil terminator* `true`.

```
(declare-const and (-> Bool Bool Bool)
  :right-assoc-nil true
)
```

Example. Declare `and` right-associative, with *nil terminator* `true`.

```
(declare-const and (-> Bool Bool Bool)
  :right-assoc-nil true
)
```

The following *n*-ary application of `and` is elaborated thus:

$$\texttt{(and p q r)} \implies \texttt{(and p (and q (and r true)))}$$

We can also declare parameterized constants:

$$(\texttt{declare-parameterized-const}\ s\ (\rho_1\ \ldots\ \rho_n)\ e\ \langle\alpha\rangle_?)$$

where ρ is a (typed) parameter. i.e.,

$$\begin{aligned} \rho \in \mathbf{param} &::= (s\ t\ \langle\nu\rangle_?) \\ \nu \in \mathbf{attr}_\nu &::= \texttt{:implicit} \mid \texttt{:list} \end{aligned}$$

We can also declare parameterized constants:

$$(\texttt{declare-parameterized-const}\ s\ (\rho_1 \ldots \rho_n)\ e\ \langle\alpha\rangle_?)$$

where ρ is a (typed) parameter. i.e.,

$$\begin{aligned} \rho \in \textbf{param} &::= (s\ t\ \langle\nu\rangle_?) \\ \nu \in \textbf{attr}_\nu &::= \texttt{:implicit} \mid \texttt{:list} \end{aligned}$$

Example. Implicit type parameter and `:chainable` attribute.

```
(declare-parameterized-const =
  ((A Type :implicit)) (-> A A Bool)
  :chainable and
)
```

Example. Implicit type parameter and `:chainable` attribute.

```
(declare-parameterized-const =
  ((A Type :implicit)) (-> A A Bool)
  :chainable and
)
```

Example. Implicit type parameter and `:chainable` attribute.

```
(declare-parameterized-const =
  ((A Type :implicit)) (-> A A Bool)
  :chainable and
)
```

The following *n*-ary application of `=` is elaborated thus:

$$\texttt{(= x y z)} \Longrightarrow \texttt{(and (= x y) (= y z))}$$

Eunoia can define symbols (with an optional type annotation):

$$(\texttt{define}\; s \; (\rho_1 \;\ldots\; \rho_n) \; e \; \langle \texttt{:type}\; t \rangle_?)$$

Example. Some definition from cpc/rules/Booleans.eo.

```
(define $remove_maybe_self ((l Bool) (C Bool))
  (eo::ite (eo::eq l C) false (eo::list_erase or C l))
)
```

Eunoia has user-defined programs.

$$\left(\begin{array}{l}\texttt{program}\ s\ (\rho_1 \ldots \rho_n) \\ \quad \texttt{:signature}\ (t_1 \ldots t_m)\ t' \\ \quad ((e_1\ e_1') \ldots (e_k\ e_k'))\end{array}\right)$$

Example. Some program from cpc/rules/Booleans.eo.

```
(program $to_clause
  ((F1 Bool) (F2 Bool :list))
  :signature (Bool) Bool
  (
    (($to_clause (or F1 F2)) (or F1 F2))
    (($to_clause false)      false)
    (($to_clause F1)         (or F1))
  )
)
```

Eunoia has rule declarations.

$$\left(\begin{array}{l} \texttt{declare-rule}\ s\ (\rho_1 \ldots \rho_n) \\ \quad \langle \texttt{:premises}\ (\varphi_1 \ldots \varphi_m) \rangle_? \\ \quad \langle \texttt{:args}\ (e_1 \ldots e_k) \rangle_? \\ \quad \texttt{:conclusion}\ \psi \end{array}\right)$$

Example. Resolution rule from cpc/rules/Booleans.eo.

```
(declare-rule resolution
  ((C1 Bool) (C2 Bool) (pol Bool) (L Bool))
  :premises (C1 C2)
  :args (pol L)
  :conclusion ($resolve C1 C2 pol L)
)
```

For proof scripts, we have two main commands:

$$\pi \in \mathbf{prf} \;::=\; (\texttt{assume}\; s\; \varphi) \;\Big|\; \left(\begin{array}{l}\texttt{step}\; s\; \langle\psi\rangle_?\; \texttt{:rule}\; s' \\ \quad\langle \texttt{:premises}\; (\varphi_1 \ldots \varphi_n)\rangle_? \\ \quad\langle \texttt{:args}\; (e_1 \ldots e_m)\rangle_? \end{array}\right) \;\Big|\; \ldots$$

Example.

```
(assume @p1 (and p (not p)))
(step @p2 :rule and_elim :premises (@p1) :args (1))
(step @p3 :rule and_elim :premises (@p1) :args (0))
(step @p4 false :rule contra :premises (@p3 @p2))
```

LambdaPi

- LambdaPi terms are those of the λΠ-calculus.

$$t \in \mathbf{term}_{\mathbf{lp}} ::= x \mid t_1 \cdot t_2 \mid \lambda x : t_1 .\, t_2 \mid \Pi x : t_1 .\, t_2$$

- LambdaPi terms are those of the λΠ-calculus.

$$t \in \mathbf{term}_{\mathbf{lp}} ::= x \mid t_1 \cdot t_2 \mid \lambda x : t_1 . t_2 \mid \Pi x : t_1 . t_2$$

- Symbols are declared thus:

$$\texttt{symbol}\ s\ \langle \rho \rangle_{*} : t;$$

where ρ ranges over LambdaPi parameters:

$$\rho \in \mathbf{param}_{\mathbf{lp}} ::= (x : t) \mid [x : t]$$

- Symbols can also be defined:

$$\texttt{symbol}\ s\ \langle : t \rangle_{?} := t';$$

- Symbols can also be defined:

$$\texttt{symbol}\ s\ \langle : t \rangle_{?} := t';$$

Note that providing the type of s is optional.

- Symbols can also be defined:

$$\texttt{symbol}\; s\; \langle : t \rangle_? := t';$$

Note that providing the type of s is optional.

- Rewrite rules are declared as follows:

$$\texttt{rule}\; r\; \langle \texttt{with}\; r' \rangle_* ;$$

Where r ranges over $\mathbf{rw} ::= (t \hookrightarrow t')$.

Type universes *a la Tarski*; closed under $(\rightsquigarrow)$.

$$\texttt{Set} : \texttt{TYPE}; \qquad \texttt{El} : \texttt{Set} \to \texttt{TYPE};$$

$$(\rightsquigarrow) : \texttt{Set} \to \texttt{Set} \to \texttt{Set};$$

Type universes *a la Tarski*; closed under $(\rightsquigarrow)$.

$$\texttt{Set} : \texttt{TYPE}; \qquad \texttt{El} : \texttt{Set} \to \texttt{TYPE};$$

$$(\rightsquigarrow) : \texttt{Set} \to \texttt{Set} \to \texttt{Set};$$

Proofs are encoded similarly:

$$\texttt{Prop} : \texttt{TYPE}; \qquad \texttt{Prf} : \texttt{Prop} \to \texttt{TYPE};$$

Type universes *a la Tarski*; closed under $(\rightsquigarrow)$.

$$\texttt{Set} : \texttt{TYPE}; \qquad \texttt{El} : \texttt{Set} \to \texttt{TYPE};$$
$$(\rightsquigarrow) : \texttt{Set} \to \texttt{Set} \to \texttt{Set};$$

Proofs are encoded similarly:

$$\texttt{Prop} : \texttt{TYPE}; \qquad \texttt{Prf} : \texttt{Prop} \to \texttt{TYPE};$$

Example.

$$\begin{aligned}
&\texttt{symbol}\ (=)\ [a : \texttt{Set}] && : \texttt{El}\,(a \rightsquigarrow a \rightsquigarrow \texttt{Bool});\\
&\texttt{symbol}\ \mathsf{refl}\ [a : \texttt{Set}]\ [x : \texttt{El}\ a] && : \texttt{Prf}\,(x = x);
\end{aligned}$$

Translation

Goal: Given a Eunoia signature Σ, generate the corresponding LambdaPi signature $T(\Sigma)$.

- Process each command in Σ, updating an environment Θ as we go:

$$T_{\Theta}(c\,;\Sigma) = c\,; T_{\Theta'}(\Sigma)$$

Goal: Given a Eunoia signature Σ, generate the corresponding LambdaPi signature $T(\Sigma)$.

- Process each command in Σ, updating an environment Θ as we go:

$$T_{\Theta}(c\,;\Sigma) = c\,;T_{\Theta'}(\Sigma)$$

- Our translation tool `eo2lp` is written in OCaml.

Goal: Given a Eunoia signature Σ, generate the corresponding LambdaPi signature $T(\Sigma)$.

- Process each command in Σ, updating an environment Θ as we go:

$$T_{\Theta}(c\,;\Sigma) = c\,; T_{\Theta'}(\Sigma)$$

- Our translation tool `eo2lp` is written in OCaml.
- The following is a high-level overview.

Expressions are first elaborated with $\mathbf{elab}_\gamma : \mathbf{eo} \to \mathbf{eo}$.

$$\gamma : \mathcal{S} \rightharpoonup (\mathbf{attr_c} \cup \mathbf{attr_v})$$

Where γ attributes of symbols during translation.

- Eunoia has a built-in symbol _ for (higher-order) application.

Expressions are first elaborated with $\mathbf{elab}_\gamma : \mathbf{eo} \to \mathbf{eo}$.

$$\gamma : \mathcal{S} \rightharpoonup (\mathbf{attr_c} \cup \mathbf{attr_v})$$

Where γ attributes of symbols during translation.

- Eunoia has a built-in symbol _ for (higher-order) application.
- The default elaboration strategy is to left-fold:

$$\begin{aligned}\mathbf{elab}_\gamma(s\ e_1\ \ldots\ e_n) &= ((s * e_1) * \ldots * e_n) \\ &= (\_\ (\ldots(\_\ s\ e_1)\ldots)\ e_n)\end{aligned}$$

Expressions are first elaborated with $\mathbf{elab}_\gamma : \mathbf{eo} \to \mathbf{eo}$.

$$\gamma : \mathcal{S} \rightharpoonup (\mathbf{attr_c} \cup \mathbf{attr_v})$$

Where γ attributes of symbols during translation.

- Eunoia has a built-in symbol _ for (higher-order) application.
- The default elaboration strategy is to left-fold:

$$\begin{aligned}\mathbf{elab}_\gamma(s\; e_1 \;\ldots\; e_n) &= ((s * e_1) * \ldots * e_n) \\ &= (\_\;(\ldots(\_\; s\; e_1)\ldots)\; e_n)\end{aligned}$$

- In general, strategy depends on attributes, e.g.,

$$\mathbf{elab}_\gamma(\texttt{and}\; p\; q\; r) = \texttt{and}\; p * (\texttt{and}\; q * (\texttt{and}\; r * \texttt{false}))$$

Translate kinds into LambdaPi types via $[\![\cdot]\!]_{\mathbf{ty}} : \mathbf{eo} \to \mathbf{lp}$;

$$[\![\texttt{Type}]\!]_{\mathbf{ty}} = \texttt{Set}$$

$$[\![((\texttt{->}) * e) * e']\!]_{\mathbf{ty}} = [\![e]\!]_{\mathbf{ty}} \to [\![e']\!]_{\mathbf{ty}}$$

$$[\![e']\!]_{\mathbf{ty}} = \texttt{El}\ [\![e']\!]_{\mathbf{tm}}$$

Translate kinds into LambdaPi types via $[\![\cdot]\!]_{\mathbf{ty}} : \mathbf{eo} \to \mathbf{lp}$;

$$[\![\texttt{Type}]\!]_{\mathbf{ty}} = \texttt{Set}$$
$$[\![((\texttt{->}) * e) * e']\!]_{\mathbf{ty}} = [\![e]\!]_{\mathbf{ty}} \to [\![e']\!]_{\mathbf{ty}}$$
$$[\![e']\!]_{\mathbf{ty}} = \texttt{El}\ [\![e']\!]_{\mathbf{tm}}$$

Example. Consider translating the following Eunoia kind.

$$\begin{aligned}[\![(\texttt{-> Int Type})]\!]_{\mathbf{ty}} &= [\![(\texttt{->} * \texttt{Int}) * \texttt{Type}]\!]_{\mathbf{ty}} \\ &= [\![\texttt{Int}]\!]_{\mathbf{ty}} \to [\![\texttt{Type}]\!]_{\mathbf{ty}} \\ &= \texttt{El}\ [\![\texttt{Int}]\!]_{\mathbf{tm}} \to \texttt{Set}\end{aligned}$$

Now, we can easily translate type declarations:

$$[\![(\texttt{declare-type}\ t\ (e_1 \ldots e_n))]\!]$$

$$\Downarrow$$

$$\texttt{symbol}\ \{\!|\ t\ |\!\}\ :\ [\![(\texttt{->}\ e_1 \ldots e_n\ \texttt{Type})]\!]_{\mathbf{ty}};$$

$$\Downarrow$$

$$\texttt{symbol}\ \{\!|\ t\ |\!\}\ :\ [\![e_1]\!]_{\mathbf{ty}} \to \ldots \to [\![e_n]\!]_{\mathbf{ty}} \to \texttt{Set};$$

Now, we can easily translate type declarations:

$$[\![\, (\texttt{declare-type} \; t \; (e_1 \ldots e_n)) \,]\!]$$

$$\Downarrow$$

$$\texttt{symbol} \; \{\!|\, t \,|\!\} : [\![\, (\texttt{->} \; e_1 \ldots e_n \; \texttt{Type}) \,]\!]_{\mathbf{ty}};$$

$$\Downarrow$$

$$\texttt{symbol} \; \{\!|\, t \,|\!\} : [\![\, e_1 \,]\!]_{\mathbf{ty}} \to \ldots \to [\![\, e_n \,]\!]_{\mathbf{ty}} \to \texttt{Set};$$

Example.

```
(declare-type Array (Type Type))
```

```
symbol {|Array|} : Set → Set;
```

Use $[\![\cdot]\!]_{\mathbf{tm}} : \mathbf{eo} \to \mathbf{lp}$ to translate terms/types to LambdaPi terms.

$$[\![s]\!]_{\mathbf{tm}} = \begin{cases} (\rightsquigarrow) & \text{if } s = (\texttt{->}), \\ \{\!| s |\!\} & \text{otherwise} \end{cases}$$

$$[\![e * e']\!]_{\mathbf{tm}} = [\![e]\!]_{\mathbf{tm}} \cdot [\![e']\!]_{\mathbf{tm}}$$

Use $[\![\cdot]\!]_{\mathbf{tm}} : \mathbf{eo} \to \mathbf{lp}$ to translate terms/types to LambdaPi terms.

$$[\![s]\!]_{\mathbf{tm}} = \begin{cases} (\leadsto) & \text{if } s = (\texttt{->}), \\ \{\!| s |\!\} & \text{otherwise} \end{cases}$$

$$[\![e * e']\!]_{\mathbf{tm}} = [\![e]\!]_{\mathbf{tm}} \cdot [\![e']\!]_{\mathbf{tm}}$$

Example. Consider translating the following type.

$$\begin{aligned} [\![(\texttt{->}\ \texttt{Bool}\ (\texttt{BitVec}\ 5))]\!]_{\mathbf{tm}} &= [\![(\texttt{->} * \texttt{Bool}) * (\texttt{BitVec} * 5)]\!]_{\mathbf{tm}} \\ &= [\![\texttt{Bool}]\!]_{\mathbf{tm}} \leadsto [\![\texttt{BitVec} * 5]\!]_{\mathbf{tm}} \\ &= \{\!| \texttt{Bool} |\!\} \leadsto (\{\!| \texttt{BitVec} |\!\} \cdot \{\!| 5 |\!\}) \end{aligned}$$

Now, we can translate constant declarations, e.g.;

$$(\texttt{declare-const}\ s\ (\texttt{->}\ e_1\ \ldots\ e_n)\ \langle\alpha\rangle_?)$$

$$\Downarrow$$

$$\texttt{constant symbol}\ \{\!|\, s\,|\!\}\ :\ \texttt{El}\ [\![\,(\texttt{->}\ e_1\ \ldots\ e_n)\,]\!]_{\mathbf{tm}};$$

$$\Downarrow$$

$$\texttt{constant symbol}\ \{\!|\, s\,|\!\}\ :\ \texttt{El}\ ([\![\,e_1\,]\!]_{\mathbf{tm}} \rightsquigarrow \ldots \rightsquigarrow [\![\,e_n\,]\!]_{\mathbf{tm}});$$

Now, we can translate constant declarations, e.g.;

$$(\texttt{declare-const}\ s\ (\texttt{->}\ e_1 \ldots e_n)\ \langle \alpha \rangle_?)$$

$$\Downarrow$$

$$\texttt{constant symbol}\ \{\!|\, s \,|\!\}: \texttt{El}\ [\![(\texttt{->}\ e_1 \ldots e_n)]\!]_{\mathbf{tm}};$$

$$\Downarrow$$

$$\texttt{constant symbol}\ \{\!|\, s \,|\!\}: \texttt{El}\ ([\![e_1]\!]_{\mathbf{tm}} \leadsto \ldots \leadsto [\![e_n]\!]_{\mathbf{tm}});$$

Also, update the attribute map γ with $(s \mapsto \alpha)$.

Translation of (implicit) parameters is easy.

$$[\![\, (s\; e) \,]\!]_{\mathbf{param}} = (\, [\![\, s \,]\!]_{\mathbf{tm}} : [\![\, e \,]\!]_{\mathbf{ty}} \,)$$

$$[\![\, (s\; e \; \texttt{:implicit}) \,]\!]_{\mathbf{param}} = [\, [\![\, s \,]\!]_{\mathbf{tm}} : [\![\, e \,]\!]_{\mathbf{ty}} \,]$$

Translate parameterized constant declarations thus:

$$(\texttt{declare-parameterized-const}\; s \; (\rho_1 \;\ldots\; \rho_n) \; e)$$

$$\Downarrow$$

$$\texttt{constant symbol} \; \{|s|\} \; [\![\rho_1]\!] \ldots [\![\rho_n]\!] : \texttt{El} \; [\![e]\!]_{\mathbf{tm}};$$

Translate parameterized constant declarations thus:

$$(\texttt{declare-parameterized-const}\; s\; (\rho_1 \ldots \rho_n)\; e)$$

$$\Downarrow$$

$$\texttt{constant symbol}\; \{\!|\, s \,|\!\}\; [\![\, \rho_1 \,]\!] \ldots [\![\, \rho_n \,]\!] : \texttt{El}\; [\![\, e \,]\!]_{\mathbf{tm}};$$

Example. Consider translating the following declaration.

```
(declare-parameterized-const =
  ((A Type :implicit)) (-> A A Bool)
  :chainable and
)
```

```
constant symbol {|=|} [A : Set] : El (A ~> A ~> Bool)
```

Definitions are translated thus:

$$(\texttt{define}\ s\ (\rho_1\ \ldots\ \rho_n)\ e\ \langle\texttt{:type}\ e'\rangle_?)$$

$$\Downarrow$$

$$\texttt{symbol}\ \{\!|\, s \,|\!\}\ [\![\, \rho_1\ \ldots\ \rho_n \,]\!]\ \langle :\ [\![\, e' \,]\!]_{\mathbf{tm}} \rangle_?\ :=\ [\![\, e \,]\!]_{\mathbf{tm}};$$

Programs are translated.

Example. Translation of `$from_clause`.

```
sequential symbol
  {|$from_clause|} : (El Bool → El Bool);

rule {|$from_clause|} (or $F1 $F2) |->
  {|eo::ite|} [Bool]
    ({|eo::is_eq|} [Bool] $F2 false)
    $F1 (or $F1 $F2)

with {|$from_clause|} $F1 |-> $F1;
```

Rule declarations are translated.

Example. Translation of `$from_clause`.

```
sequential symbol
  cnf_implies_pos_aux : (El Bool → El Bool);

rule cnf_implies_pos_aux (=> $F1 $F2)
  |-> or (not (=> $F1 $F2))
         (or (not $F1) (or $F2 false));

constant symbol cnf_implies_pos : Π (x0 : El Bool),
  El (Proof (cnf_implies_pos_aux x0));
```

Proof scripts are translated:

Example. Translation of `$from_clause`.

```
constant symbol Z : Set;
constant symbol input : El Bool;
constant symbol reg : El Bool;
constant symbol nf : El Z;
constant symbol flash : El Z;
constant symbol circuit : El Bool;
symbol {|@t1|} : El Bool   not input;
symbol {|@t2|} : El Bool   not reg;
symbol {|@t3|} : El Bool   and input (and {|@t2|} true);
constant symbol {|@p1|} : El (Proof circuit);
constant symbol {|@p2|} : El (Proof (= nf flash));
constant symbol {|@p3|} : El (Proof (not (or {|@t3|} (or {|@t1|} (or reg false)))));
symbol {|@p4|} : El (Proof (not {|@t3|}))   not_or_elim [or {|@t3|} (or {|@t1|} (or reg false)
symbol {|@p5|} : El (Proof {|@t2|})   not_or_elim [or {|@t3|} (or {|@t1|} (or reg false))] {|@
symbol {|@p6|} : El (Proof (not {|@t1|}))   not_or_elim [or {|@t3|} (or {|@t1|} (or reg false)
symbol {|@p7|} : El (Proof input)   not_not_elim [input] {|@p6|};
symbol {|@p8_aux|} : El (Proof (and input (and {|@t2|} true)))   and_cons {|@p7|} (and_cons {|
symbol {|@p8|} : El (Proof {|@t3|})   and_intro [and input (and {|@t2|} true)] {|@p8_aux|};
symbol {|@p9|} : El (Proof false)   contra [{|@t3|}] {|@p8|} {|@p4|};
```

Results & Future Work

Carve out the portion of CPC supporting QFUF.

- Rodin SMT-LIB benchmark, 30 unsat problems.

Carve out the portion of CPC supporting QFUF.

- Rodin SMT-LIB benchmark, 30 unsat problems.
- Run cvc5 with `--proof-format=cpc`, dump proofs.

Carve out the portion of CPC supporting QFUF.

- Rodin SMT-LIB benchmark, 30 unsat problems.
- Run `cvc5` with `--proof-format=cpc`, dump proofs.
- Check which CPC rules were used, calculate dependencies.

Carve out the portion of CPC supporting QFUF.

- Rodin SMT-LIB benchmark, 30 unsat problems.
- Run `cvc5` with `--proof-format=cpc`, dump proofs.
- Check which CPC rules were used, calculate dependencies.
- Make some minor modifications, call this fork CPC-mini.

Translate CPC-mini to LambdaPi using `eo2lp`.

Translate all of our Rodin proofs.

Lots of potential for future work:

- 🌍 Support full CPC: arithmetic, strings, bit-vectors, etc.

Lots of potential for future work:

- 🌍 Support full CPC: arithmetic, strings, bit-vectors, etc.
- 📈 Scale up to bigger proofs.

Lots of potential for future work:

- 🌍 Support full CPC: arithmetic, strings, bit-vectors, etc.
- 📈 Scale up to bigger proofs.
- 🧹 Tidy translation: perform elaboration in LambdaPi?

Lots of potential for future work:

- 🌍 Support full CPC: arithmetic, strings, bit-vectors, etc.
- 📈 Scale up to bigger proofs.
- 🧹 Tidy translation: perform elaboration in LambdaPi?
- 🇧🇷 Do all of this in Brazil, Nov 2025?