System Y — Solid Foundations for SMT Proofs

Hans-Jörg Schurr, J. Garrett Morris, Christa Jenkins
Andrew Reynolds, Cesare Tinelli

EuroProofNet — WG2 Closing Workshop
Institut Pascal, Orsay, France (remote)
September 11, 2025

SMT Proofs: Everyone for Themselves?

- SMT proofs are in use!
  - **cvc5 in Isabelle**
  - veriT in Isabelle
  - Internal proof checkers
  - External proof checkers
  - **Translation to Dedukti**

SMT Proofs: Everyone for Themselves?

- SMT proofs are in use!
  - **cvc5 in Isabelle**
  - veriT in Isabelle
  - Internal proof checkers
  - External proof checkers
  - **Translation to Dedukti**
- However!
  - This is done for each solver individually!
  - There is no hope for an SMT DRAT.

SMT Proofs: Everyone for Themselves?

- SMT proofs are in use!
    - **cvc5 in Isabelle**
    - veriT in Isabelle
    - Internal proof checkers
    - External proof checkers
    - **Translation to Dedukti**
- However!
    - This is done for each solver individually!
    - There is no hope for an SMT DRAT.

Alethe

+ Looks like SMT-LIB
+ Used! (cvc5, veriT, Isabelle, Carcara)
+ Well documented...
- ... in English
- Generality is questionable

SMT Proofs: Everyone for Themselves?

- SMT proofs are in use!
  - **cvc5 in Isabelle**
  - veriT in Isabelle
  - Internal proof checkers
  - External proof checkers
  - **Translation to Dedukti**
- However!
  - This is done for each solver individually!
  - There is no hope for an SMT DRAT.

Alethe

+ Looks like SMT-LIB
+ Used! (cvc5, veriT, Isabelle, Carcara)
+ Well documented…
- … in English
- Generality is questionable

LFSC

+ High performance checker
+ Declarative
- Hard to read
- Side conditions from another world
- Limited theories

Eunoia: Inspired by Alethe and LFSC

Goal 1
Look like SMT-LIB and Alethe.

By SMT people for SMT people!

Eunoia: Inspired by Alethe and LFSC

Goal 1
Look like SMT-LIB and Alethe.

Goal 2
Provide a declarative language to specify proof rules for all SMT-LIB logics.
(that includes Bit Vectors).

By SMT people for SMT people!

Eunoia: Inspired by Alethe and LFSC

Goal 1
Look like SMT-LIB and Alethe.

Goal 2
Provide a declarative language to specify proof rules for all SMT-LIB logics.
(that includes Bit Vectors).

Goal 3
Allow fast checking of proofs against specified rules.

By SMT people for SMT people!

Eunoia: Inspired by Alethe and LFSC

Goal 1
Look like SMT-LIB and Alethe.

Goal 2
Provide a declarative language to specify proof rules for all SMT-LIB logics.
(that includes Bit Vectors).

Goal 3
Allow fast checking of proofs against specified rules.

Non Goal
Rules can be specified freely. It is not necessary to prove them correct.
Not Curry-Howard correspondence based.

By SMT people for SMT people!

Eunoia: Example 1

```
(assume a1 (and true (= a b)))
(assume a2 (= b c))
(step s1 (= a b)          :rule andE  :premises (a1) :args (2))
(step s2 (= a c)          :rule trans :premises (s1 a2))
(step s3 (= (f a) (f c)) :rule cong  :premises (s2) :args (f))
```

Eunoia: Example 1 (Under The Hood)

$$\Gamma \vdash (\text{cong}\ f\ (\text{trans}$$
$$(\text{andE}\ 2\ (\text{assume}\ (\top \wedge (a = b)))$$
$$(\text{assume}\ (a = b))))) : \text{Proof}\ (f\ a = f\ b)$$

Eunoia: Example 1 (The Rules)

```
(declare-rule trans ((T Type) (a T) (b T) (c T))
   :premises ((= a b) (= b c))
   :conclusion (= a c)
)
(declare-rule cong ((T Type) (S Type) (a T) (b T) (f (-> S T)))
   :premises ((= a b))
   :args (f)
   :conclusion (= (f a) (f c))
)
```

Eunoia: Example 1 (The Rules)

```
(program select ((a Bool) (b Bool) (i Int))
    :signature (Int Bool) Bool
    (
        ((select 1 (and a b)) a)
        ((select 2 (and a b)) b)
    )
)
(declare-rule andE ((a Bool) (b Bool) (i Int))
   :premises ((and a b))
   :args (i)
   :conclusion (select i (and a b))
)
```

Eunoia: Example 1 (The Rules, Abstractly)

$$\Gamma \vdash \mathrm{trans} : \mathrm{Proof}\ a = b \rightarrow \mathrm{Proof}\ b = c \rightarrow \mathrm{Proof}\ a = c$$

$$\Gamma \vdash \mathrm{cong} : (f : T \rightarrow S) \rightarrow \mathrm{Proof}\ a = b \rightarrow \mathrm{Proof}\ (f\ a) = (f\ b)$$

$$\Gamma \vdash \mathrm{andE} : (i : \mathrm{Int}) \rightarrow \mathrm{Proof}\ a \wedge b \rightarrow \mathrm{Proof}\ (\mathrm{select}\ i\ (a \wedge b)$$

Eunoia: Example 2 (Recursion)

```
(program selectLast ((a Bool) (b Bool))
    :signature (Bool) Bool
    (
        ((selectLast (and a b)) (selectLast b))
        ((selectLast a)                      a )
    )
)
(declare-rule andLast ((a Bool))
   :premises (a)
   :conclusion (selectLast a)
)
```

So what do we have?

Eunoia is

- a dependently typed programming language,
- that mixes data and computation freely,
- that allows divergent computations.
- Computations can throw exceptions
  - that can be cached!
  - i.e., we have observable effects.
- More features I will discuss.
- Some features I will not discuss.

So what do we have?

Eunoia is

- a dependently typed programming language,
- that mixes data and computation freely,
- that allows divergent computations.
- Computations can throw exceptions
    - that can be cached!
    - i.e., we have observable effects.
- More features I will discuss.
- Some features I will not discuss.

Oh, my...
how does that all work?

Let's look at Ethos!

Ethos: a Proof Checker, Not a Type Checker

Ethos checking model (roughly):

1. Check only that (constants, programs, rules) signature is well-formed.
2. Iterate over proof steps.
   - Observe that all terms have concrete type!

   2.1 Instantiate variables in types.

   2.2 Recurse into type constraints.

   2.3 Perform computations.
   - Divergence, exception: proof reject.

Upsides

- Correct!
- Fast.
- Easy to implement.
- Easy to extend.

Downsides

- Bugs in rules can be missed.
- Unexpected.
- Wasted work (e.g., function composition).
- Breaks **Goal 2**

Ethos: Goal 2 Problem

Goal 2
Provide a declarative language to specify proof rules for all SMT-LIB logics.

```
; bvsub, bvadd : BitVec n -> BitVec n -> BitVec n
(declare-rule bv-sub-eliminate
  ((n Int) (m Int) (x (BitVec n)) (y (BitVec m)))
 :args (x y)
 :conclusion (= (bvsub x y) (bvadd x (bvneg y)))
)
```

I claim: not well typed. Not a soundness issue: Ethos accepts only valid uses.

However, Ethos cannot detect that this is a bad specification.

So what do we have?

Eunoia is

- a dependently typed programming language,
- that mixes data and computation freely,
- that allows divergent computations.
- Computations can throw exceptions
  - that can be cached!
  - i.e., we have observable effects.
- More features I will discuss.
- Some features I will not discuss.

~~Oh, my...~~
~~how does that all work?~~

~~**Let's look at Ethos!**~~

So what do we have?

Eunoia is

- a dependently typed programming language,
- that mixes data and computation freely,
- that allows divergent computations.
- Computations can throw exceptions
  - that can be cached!
  - i.e., we have observable effects.
- More features I will discuss.
- Some features I will not discuss.

J. Garrett Morris solving a problem.

μEunoia : System Ƴ + Core Eunoia

System Ƴ: *Decidable* Dependent Type Theory with explicit evaluation evidence.

μEunoia is not a subset of Eunoia.

μEunoia checking model (abstractly):

1. Write your signature in μEunoia (auto translation is future work).
2. Typecheck your signature.
3. Run modified Ethos on an **Eunoia** proof.
    - Divergence, exceptions: reject proof
    - Otherwise: output proof with evaluation evidence (μEunoia proof)
4. Typecheck your μEunoia proof.

Rule Sketches (I am sorry for this slide.)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash_c M : A}$$

$$\frac{\Gamma \vdash A : \mathscr{T} \quad \Gamma, A \vdash_c M : B}{\Gamma \vdash \lambda\, M : \Pi\, A\, B}$$

$$\frac{\Gamma \vdash_c M : \mathscr{M}A \quad \Gamma \vdash M \twoheadrightarrow [n]\ \mathrm{return}\ V}{\Gamma \vdash \langle n \rangle : V \leftarrow M}$$

$$\frac{\Gamma \vdash A : \mathscr{T}}{\Gamma \vdash \mathscr{M}A : \mathscr{T}}$$

$$\frac{\Gamma \vdash A : \mathscr{T} \quad \Gamma, A \vdash_c B : \mathscr{T}}{\Gamma \vdash \Pi\, A\, B : \mathscr{T}}$$

$$\frac{\Gamma \vdash M : \mathrm{nlet}\ A\ B \quad \Gamma \vdash N : C \leftarrow A}{\Gamma \vdash [M]\ N : B\ [C]}$$

$$\frac{\Gamma \vdash M : A}{\mathrm{return}\ M : \mathscr{M}A}$$

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash_c M : \mathscr{M}A}{\Gamma \vdash V \leftarrow M : \mathscr{T}}$$

$$\frac{\Gamma \vdash_c M : \mathrm{nlet}\ A\ B \quad \Gamma \vdash N : C \leftarrow A}{\Gamma \vdash_c [M]\ N : B\ [C]}$$

$$\frac{\Gamma \vdash_c M : \mathscr{M}A \quad \Gamma \vdash_c B : \mathscr{T} \quad \Gamma, A \vdash_c N : \mathrm{wk}\ B}{\Gamma \vdash_c \mathrm{nlet}\ M\ N : B}$$

$$\frac{\Gamma \vdash_c M : \mathscr{M}A \quad \Gamma, A \vdash_c B : \mathscr{T} \quad \Gamma, A \vdash_c N : B}{\Gamma \vdash_c \mathrm{dlet}\ M\ N : \mathrm{nlet}\ M\ B}$$

Example (Slightly Different Syntax)

$$\text{zeros} : \Pi(n : \mathbb{Z}).\ \text{Vec Int}\ n$$

$$\begin{aligned}
\text{moreZeroes} &: (n : \mathbb{Z}) \to (m : \mathbb{Z}) \to \quad \text{let}\ p = \text{add}\ \mathbb{Z}\,\mathbb{Z}\,\mathbb{Z}\ n\ m\ \langle 1\rangle\ \text{in}\ \mathit{Vec}\ \mathbb{Z}\ p \\
\text{moreZeroes}\ n\ m &= \quad \text{dlet}\ p = \text{add}\ \mathbb{Z}\,\mathbb{Z}\,\mathbb{Z}\ n\ m\ \langle 1\rangle\ \text{in zeros}\ p
\end{aligned}$$

$$\begin{aligned}
\text{theZeros} &: \text{Vec}\ \mathbb{Z}\ 12 \\
\text{theZeros} &= [\text{moreZeros}\ 9\ 3]\ \langle 4\rangle
\end{aligned}$$

The Project

- Goal: add enough to be “Eunoia”
- Deep Embedding in Agda!
- Substantial: $>$ 11 000 lines

What We Support

- Signatures
- Literals
- Overriding literal typing
- Non-linear matching
- Builtins
- Exceptions
- Special variable scoping
    - Declaration-wide scopes
    - “Quote” Arrow

Eunoia Variables: Declaration-Scoped

```
(declare-parameterized-const ubv_to_int
                ((m Int :implicit))
        (-> (BitVec m) Int))
(program fromBvAdd ((n Int) (bv (BitVec n)))
  :signature (Int (BitVec n)) Int
  ( ((fromInt n bv) (+ n (ubv_to_int bv)) )
)
```

"Quote" Arrow

$$\mathrm{ex} : [n + m : \mathbb{Z}] \mapsto \mathrm{BitVec}\ n$$

$$\mathrm{ex}\ (1 + 2) : \mathrm{BitVec}\ 1$$

- Every declaration has *n* variables.
  - `Vec Bool` n to mark assigned, free, bound variables
  - `Vec Term` n for typing, substitution.

Eunoia Variables: Declaration-Scoped

```
(declare-parameterized-const ubv_to_int
                ((m Int :implicit))
        (-> (BitVec m) Int))
(program fromBvAdd ((n Int) (bv (BitVec n)))
  :signature (Int (BitVec n)) Int
  ( ((fromInt n bv) (+ n (ubv_to_int bv)) )
)
```

"Quote" Arrow

$$\text{ex} : [n + m : \mathbb{Z}] \mapsto \text{BitVec } n$$

$$\text{ex } (1 + 2) : \text{BitVec } 1$$

- Every declaration has *n* variables.
  - `Vec Bool n` to mark assigned, free, bound variables
  - `Vec Term n` for typing, substitution.
- Kills De Bruijn indices 😞

Eunoia Variables: Declaration-Scoped

```
(declare-parameterized-const ubv_to_int
                ((m Int :implicit))
        (-> (BitVec m) Int))
(program fromBvAdd ((n Int) (bv (BitVec n)))
  :signature (Int (BitVec n)) Int
  ( ((fromInt n bv) (+ n (ubv_to_int bv)) )
)
```

“Quote” Arrow

$$\text{ex} : [n + m : \mathbb{Z}] \mapsto \text{BitVec } n$$

$$\text{ex } (1 + 2) : \text{BitVec } 1$$

- Every declaration has *n* variables.
  - `Vec Bool n` to mark assigned, free, bound variables
  - `Vec Term n` for typing, substitution.
- Kills De Bruijn indices 😞
- Matching with vectors that track free/bound variables 😎
- Spine-local type inference to assign types in applications.

Eunoia Variables: Declaration-Scoped

```
(declare-parameterized-const ubv_to_int
                ((m Int :implicit))
        (-> (BitVec m) Int))
(program fromBvAdd ((n Int) (bv (BitVec n)))
  :signature (Int (BitVec n)) Int
  ( ((fromInt n bv) (+ n (ubv_to_int bv)) )
)
```

"Quote" Arrow

$$\text{ex} : [n + m : \mathbb{Z}] \mapsto \text{BitVec } n$$

$$\text{ex } (1 + 2) : \text{BitVec } 1$$

- Every declaration has *n* variables.
  - `Vec Bool` n to mark assigned, free, bound variables
  - `Vec Term` n for typing, substitution.
- Kills De Bruijn indices 😞
- Matching with vectors that track free/bound variables 😎
- Spine-local type inference to assign types in applications.
- Big problem: `dlet` leaks variables to outer context!
  - Solution: Program calls must transfer variables into the caller context.

μEunoia

The Project

- Goal: add enough to be "Eunoia"
- Deep Embedding in Agda!
- Substantial: > 11 000 lines

What We Support

- Signatures
- Literals
- Overriding literal typing
- Non-linear matching
- Builtins
- Exceptions
- Special variable scoping
    - Declaration-wide scopes
    - "Quote" Arrow

Status

100%	Language, Evaluation, Typing
100%	Unicity
100%	Decidability
100%	Progress
75%	Preservation
10%	Soundness Case Study

Thank You!