An efficient and flexible evidence-providing solver for polynomial equalities in Agda

Donnacha Oisín Kidney

November 27, 2018

Talking about Mathematics in a Programming Language

Formalised and Mechanized Mathematics

Programming is Proving

A Polynomial Solver

Formalised and Mechanized Mathematics

Why?

Why?

Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem.

***Scientific American*, 237(4):108–121, 1977**

Why?

Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem.

***Scientific American*, 237(4):108–121, 1977**

Did contain bugs!

Formalised mathematics is an attempt to find a core set of axioms and consistent rules from which all mathematical truths can be derived.

Hilbert's program: "dispose of the foundational questions in mathematics once and for all."

But didn't Hilbert's program fail?

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.

Principia Mathematica. Vol. I.

1910 p. 379

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Gödel showed that universal
formal systems are incomplete

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Gödel showed that universal
formal systems are incomplete

Church Proved the
Entscheidungsproblem is
unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

Church Proved the Entscheidungsproblem is unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems!

Church Proved the Entscheidungsproblem is unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems!

Church Proved the Entscheidungsproblem is unsolvable!

We don't automate everything

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

So Where Are We Now?

So Where Are We Now?

- 93% of the “Top 100” Theorems

  Freek Wiedijk. Formalizing 100 Theorems, October 2018

So Where Are We Now?

- 93% of the "Top 100" Theorems

  Freek Wiedijk. Formalizing 100 Theorems, October 2018

- Metamath

  Norman Megill. *Metamath: A Computer Language for Pure Mathematics*.

  Lulu Press, Morrisville, 2007.

  OCLC: 924789462

So Where Are We Now?

- 93% of the "Top 100" Theorems

  Freek Wiedijk. Formalizing 100 Theorems, October 2018
- Metamath

  Norman Megill. *Metamath: A Computer Language for Pure Mathematics*.
  Lulu Press, Morrisville, 2007.
  OCLC: 924789462
- Coq, Agda, etc.

  Georges Gonthier. Formal Proof—The Four-Color Theorem.
  ***Notices of the AMS*, 55(11):12, 2008**

What Does our System Look Like?

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

Law of the Excluded Middle $\times$

$$p \vee \neg p$$

Proof By Contradiction $\times$

$$\neg\neg p \rightarrow p$$

Principle of Explosion $\checkmark$

$$\neg p \wedge p \rightarrow q$$

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

Law of the Excluded Middle $\times$

$$p \vee \neg p$$

Proof By Contradiction $\times$

$$\neg\neg p \to p$$

Principle of Explosion $\checkmark$

$$\neg p \wedge p \to q$$

Partially Automated

While our system can't solve arbitrary problems, we can write certain solvers for specific domains—that's the purpose of this project.

It's similar to the goal of machine learning and AI today, in this sense. While we probably can't build something to solve *everything*, we can build a system that assists us in solving "everything".

Programming is Proving

Per Martin-Löf. *Intuitionistic Type Theory*.

Padua, June 1980

Why Would a Programmer Want to Use this Language?

Why Would a Programmer Want to Use this Language?

- *Prove* things about code

```
assert(list(reversed([1,2,3])) == [3,2,1])
```

vs

$$\mathsf{reverse\text{-}involution} : \forall\ xs \to \mathsf{reverse}\ (\mathsf{reverse}\ xs) \equiv xs$$

Why Would a Programmer Want to Use this Language?

- *Prove* things about code
- Use ideas and concepts from maths—why reinvent them?

Why Would a Programmer Want to Use this Language?

- *Prove* things about code
- Use ideas and concepts from maths—why reinvent them?
- Provide coherent *justification* for language features

The Curry-Howard Correspondence

Philip Wadler. Propositions As Types.

***Commun. ACM*, 58(12):75–84, November 2015**

The Curry-Howard Correspondence

$$
\begin{array}{ccc}
\textit{Type} & \Longleftrightarrow & \textit{Proposition} \\
\downarrow & & \downarrow \\
\textit{Program} & \Longleftrightarrow & \textit{Proof}
\end{array}
$$

Philip Wadler. Propositions As Types.

***Commun. ACM*, 58(12):75–84, November 2015**

Proofs are Programs

Proofs are Programs

Types/Propositions are *sets*

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Proofs are Programs

Types/Propositions are *sets*

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Inhabited by *proofs*

Bool	Proposition
true, false	Proof

Implication

Implication

$A \rightarrow B$

Implication

$A \rightarrow B$

A implies B

Implication

$A \rightarrow B$

A implies B

Constructivist/Intuitionistic

Booleans?

Booleans?

```
data ⊥ : Set where
```

Contradiction

Booleans?

```
data ⊥ : Set where
```

Contradiction

```
law-of-non-contradiction : ∀ {a} {A : Set a} → ¬ A → A → ⊥
law-of-non-contradiction f x = f x
```

Booleans?

```
data ⊥ : Set where
```

Contradiction

```
law-of-non-contradiction : ∀ {a} {A : Set a} → ¬ A → A → ⊥
law-of-non-contradiction f x = f x
```

```
not-false : ¬ ⊥
not-false ()
```

Booleans?

```
data ⊥ : Set where
```

Contradiction

```
data ⊤ : Set where
  tt : ⊤
```

Tautology

Turing Completeness

Turing Completeness

The dual to termination is *productivity*

Turing Completeness

The dual to termination is *productivity*

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

Turing Completeness

The dual to termination is *productivity*

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail  : Stream A
```

You can write terminating and non-terminating programs: *you just have to say so*

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*.

PhD Thesis, PhD thesis, Université Paris VII, 1972

Russell’s Paradox

“The Set of all Sets which do not contain themselves”

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l’arithmétique d’ordre supérieur*.

PhD Thesis, PhD thesis, Université Paris VII, 1972

$$
\begin{aligned}
&\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{not}\ \mathsf{true} \ = \mathsf{false}\\
&\mathsf{not}\ \mathsf{false} = \mathsf{true}
\end{aligned}
$$

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*.

PhD Thesis, PhD thesis, Université Paris VII, 1972

$$
\begin{aligned}
\mathsf{not} &: \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{not}\ \mathsf{true} &= \mathsf{false} \\
\mathsf{not}\ \mathsf{false} &= \mathsf{true}
\end{aligned}
$$

$$
\begin{aligned}
\neg\_ &: \mathsf{Set} \to \mathsf{Set} \\
\neg\, A &= A \to \bot
\end{aligned}
$$

Function Extensionality

```
postulate function-extensionality
            : {A B : Set} {f g : A → B}
            → (∀ x → f x ≡ g x)
            → f ≡ g
```

A Polynomial Solver

Monoids

Monoids

Monoid

A monoid is a set equipped with a binary operation, $\bullet$, and a distinguished element ϵ, such that the following equations hold:

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z \qquad \text{(Associativity)}$$

$$x \bullet \epsilon = x \qquad \text{(Left Identity)}$$

$$\epsilon \bullet x = x \qquad \text{(Right Identity)}$$

A Boring Proof

$$\mathsf{ident} : \forall\ w\ x\ y\ z$$
$$\to ((w \bullet \varepsilon) \bullet (x \bullet y)) \bullet z \approx (w \bullet x) \bullet (y \bullet z)$$

A Boring Proof

```
ident : ∀ w x y z
  → ((w • ε) • (x • y)) • z ≈ (w • x) • (y • z)


ident w x y z =
  begin
    ((w • ε) • (x • y)) • z
  ≈⟨ assoc (w • ε) (x • y) z ⟩
    (w • ε) • ((x • y) • z)
  ≈⟨ identityʳ w ⟨ •-cong ⟩ assoc x y z ⟩
    w • (x • (y • z))
  ≈⟨ sym (assoc w x (y • z)) ⟩
    (w • x) • (y • z)
  ∎
```

Automation

Automation

Goals

Automation

Goals

Decidable Should be total, and terminating.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

External solvers (Z3, etc) We don't trust them!

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

External solvers (Z3, etc) We don't trust them!

Canonical forms Our approach.

Canonical Forms

```
infixr 5 _::_
data List (i : ℕ) : Set where
  [] : List i
  _::_ : Fin i → List i → List i
```

Canonical Forms

```
infixr 5 _∷_
data List (i : ℕ) : Set where
  [] : List i
  _∷_ : Fin i → List i → List i


infixr 5 _++_
_++_ : ∀ {i} → List i → List i → List i
[] ++ ys = ys
(x ∷ xs) ++ ys = x ∷ xs ++ ys
```

Canonical Forms

```
infixr 5 _∷_
data List (i : ℕ) : Set where
  [] : List i
  _∷_ : Fin i → List i → List i


infixr 5 _++_
_++_ : ∀ {i} → List i → List i → List i
[] ++ ys = ys
(x ∷ xs) ++ ys = x ∷ xs ++ ys


_μ_ : ∀ {i} → List i → Vec Carrier i → Carrier
[] μ ρ = ε
(x ∷ xs) μ ρ = lookup x ρ • xs μ ρ
```

```
obvious
  : (List 4 ∋
  ((η # 0 ++ []) ++ (η # 1 ++ η # 2)) ++ η # 3)
  ≡ (η # 0 ++ η # 1) ++ (η # 2 ++ η # 3)
obvious = ≡.refl
```

Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  ν_  : Fin i → Expr i
```

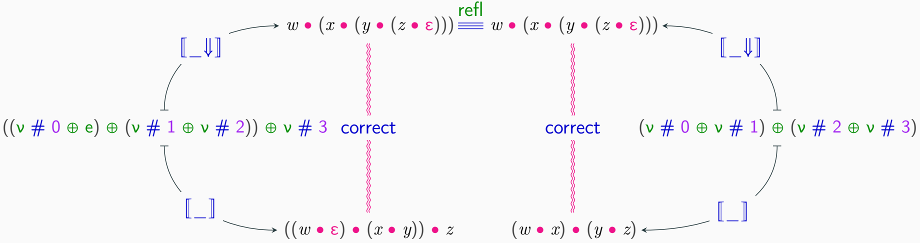
Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  ν_  : Fin i → Expr i
```

Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  ν_  : Fin i → Expr i
```

```
⟦_⟧ : ∀ {i} → Expr i → Vec Carrier i → Carrier
⟦ x ⊕ y ⟧ ρ = ⟦ x ⟧ ρ • ⟦ y ⟧ ρ
⟦ e ⟧ ρ     = ε
⟦ ν i ⟧ ρ   = lookup i ρ
```


refl
w • (x • (y • (z • ε)))
w • (x • (y • (z • ε)))
⟦_⇓⟧
⟦_⇓⟧
((ν # 0 ⊕ e) ⊕ (ν # 1 ⊕ ν # 2)) ⊕ ν # 3
correct
correct
(ν # 0 ⊕ ν # 1) ⊕ (ν # 2 ⊕ ν # 3)
⟦_⟧
⟦_⟧
((w • ε) • (x • y)) • z
(w • x) • (y • z)

Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq.

In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg

Canonical Form

```
Poly : Set ℓ
Poly = List Carrier

_⊞_ : Poly → Poly → Poly
[] ⊞ ys = ys
(x ∷ xs) ⊞ [] = x ∷ xs
(x ∷ xs) ⊞ (y ∷ ys) = x + y ∷ xs ⊞ ys

_⊠_ : Poly → Poly → Poly
_⊠_ [] _ = []
_⊠_ (x ∷ xs) =
  foldr (λ y ys → x * y ∷ map (_* y) xs ⊞ ys) []
```

Horner's Rule

$$
\begin{aligned}
p(x) &= a_0x^0 + a_1x^1 + a_2x^2 + ...a_nx^n \\
&= a_0 + x(a_1 + x(a_2 + x(...a_n + x(0))))
\end{aligned}
$$

Horner's Rule

$$
\begin{aligned}
p(x) &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \ldots a_n x^n \\
&= a_0 + x(a_1 + x(a_2 + x(\ldots a_n + x(0))))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{Poly} \rightarrow \mathsf{Carrier} \rightarrow \mathsf{Carrier} \\
&[\![\ x\ ]\!]\ \rho = \mathsf{foldr}\ (\lambda\ y\ ys \rightarrow y + \rho * ys)\ 0\#\ x
\end{aligned}
$$

Problems

Problems

Redundancy

$$\begin{aligned} 2x &= 0, 2 \\ & 0, 2, 0 \\ & 0, 2, 0, 0 \\ & 0, 2, 0, 0, 0, 0, 0 \end{aligned}$$

Problems

Redundancy

$$
\begin{aligned}
2x = {} & 0, 2 \\
& 0, 2, 0 \\
& 0, 2, 0, 0 \\
& 0, 2, 0, 0, 0, 0, 0
\end{aligned}
$$

Inefficiency

A Sparse Encoding

$3 + 2x^2 + 4x^5 + 2x^7$

A Sparse Encoding

$$3 + 2x^2 + 4x^5 + 2x^7 = x^0(3 + xx^1(2 + xx^2 * (4 + xx^1(2 + x0))))$$

```
[(3,0),(2,1),(4,2),(2,1)]
```

```
infixl 6 _≠0
record Coeff : Set (a ⊔ ℓ) where
  inductive
  constructor _≠0
  field
    coeff : Carrier
    .{coeff≠0} : ¬ Zero coeff
open Coeff

Poly : Set (a ⊔ ℓ)
Poly = List (Coeff × ℕ)
```

Termination

$$\begin{aligned}
&\mathrm{fib} : \mathbb{N} \to \mathbb{N} \\
&\mathrm{fib}\ 0 = 0 \\
&\mathrm{fib}\ 1 = 1 \\
&\mathrm{fib}\ (1{+}\ (1{+}\ n)) = \mathrm{fib}\ (1{+}\ n) + \mathrm{fib}\ n
\end{aligned}$$

$$\begin{aligned}
&\mathrm{fib} : \mathbb{N} \to \mathbb{N} \\
&\mathrm{fib}\ 0 = 0 \\
&\mathrm{fib}\ 1 = 1 \\
&\mathrm{fib}\ n = \mathrm{fib}\ (n - 1) + \mathrm{fib}\ (n - 2)
\end{aligned}$$

Well-Founded Recursion

Well-Founded Relation

Contains no infinite descending chains.

"Less than" on $\mathbb{N}$

$$1 < 2 < 4 < 8$$

Bengt Nordström. Terminating general recursion.

***BIT*, 28(3):605–619, September 1987**

Is this consistent?

Well-Founded Recursion

```
data Acc {A : Set} (_R_ : A → A → Set) (x : A) : Set where
  acc : (∀ y → y R x → Acc _R_ y) → Acc _R_ x

data _<_ (m : ℕ) : ℕ → Set where
  0<1 : m < suc m
  m<s : ∀ {n} → m < n → m < suc n

<-wellFounded : ∀ m → Acc _<_ m
<-wellFounded = acc ∘ go
  where
  go : ∀ m n → n < m → Acc _<_ n
  go zero n ()
  go (suc m) .m 0<1 = <-wellFounded m
  go (suc m) n (m<s n<m) = go m n n<m
```

Homomorphism

Richard Bird and Oege de Moor. *Algebra of Programming*.

Prentice-Hall international series in computer science. Prentice Hall, London ; New York, 1997

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation.

***Journal of Functional Programming*, 19(5):545–579, September 2009**

Interface

```
ident′ : ∀ w x y z
       → ((w • ε) • (x • y)) • z
       ≈ (w • x) • (y • z)
ident′ = solve 4
  ( λ w x y z
    → ((w ⊕ e) ⊕ (x ⊕ y)) ⊕ z
    ⊜ (w ⊕ x) ⊕ (y ⊕ z))
  refl
```

$$
\begin{tikzcd}
 & w \bullet (x \bullet (y \bullet (z \bullet \varepsilon))) \arrow[r, equal, "\mathsf{refl}"] \arrow[dd, squiggly, dash, "\mathsf{correct}"] & w \bullet (x \bullet (y \bullet (z \bullet \varepsilon))) \arrow[dd, squiggly, dash, "\mathsf{correct}"] & \\
((\nu \mathbin{\#} 0 \oplus e) \oplus (\nu \mathbin{\#} 1 \oplus \nu \mathbin{\#} 2)) \oplus \nu \mathbin{\#} 3 \arrow[ur, bend left, mapsto, "\llbracket \_ \Downarrow \rrbracket"] \arrow[dr, bend right, mapsto, "\llbracket \_ \rrbracket"'] & & & (\nu \mathbin{\#} 0 \oplus \nu \mathbin{\#} 1) \oplus (\nu \mathbin{\#} 2 \oplus \nu \mathbin{\#} 3) \arrow[ul, bend right, mapsto, "\llbracket \_ \Downarrow \rrbracket"'] \arrow[dl, bend left, mapsto, "\llbracket \_ \rrbracket"] \\
 & ((w \bullet \varepsilon) \bullet (x \bullet y)) \bullet z & (w \bullet x) \bullet (y \bullet z) &
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & w \bullet (x \bullet (y \bullet (z \bullet \varepsilon))) \arrow[r, equal, "\mathsf{refl}"] \arrow[dd, squiggly, dash, "\mathsf{correct}"] & w \bullet (x \bullet (y \bullet (z \bullet \varepsilon))) \arrow[dd, squiggly, dash, "\mathsf{correct}"] & \\
((\nu \mathbin{\#} 0 \oplus e) \oplus (\nu \mathbin{\#} 1 \oplus \nu \mathbin{\#} 2)) \oplus \nu \mathbin{\#} 3 \arrow[ur, mapsto, bend left, "[\![\_\Downarrow]\!]"] \arrow[dr, mapsto, bend right, "[\![\_]\!]"'] & & & (\nu \mathbin{\#} 0 \oplus \nu \mathbin{\#} 1) \oplus (\nu \mathbin{\#} 2 \oplus \nu \mathbin{\#} 3) \arrow[ul, mapsto, bend right, "[\![\_\Downarrow]\!]"'] \arrow[dl, mapsto, bend left, "[\![\_]\!]"] \\
 & ((w \bullet \varepsilon) \bullet (x \bullet y)) \bullet z \arrow[ul, dashed, bend left=60, "\mathsf{quoteTerm}"] & (w \bullet x) \bullet (y \bullet z) \arrow[ur, dashed, bend right=60, "\mathsf{quoteTerm}"'] &
\end{tikzcd}
$$

The Finished Solver

```
lemma : ∀ x y
      → x + y * 1 + 3 ≈ 2 + 1 + x + y
lemma = solve NatRing
```

Setoids

Setoids

Pedagogical Solutions

```
data Traced {A : Set} (x : A) : A → Set where
  refl    : Traced x x
  ⟨_⟩≡_ : ∀ {y z}
          → (reason : String)
          → Traced y z
          → Traced x z
```

Setoids

Isomorphisms

```agda
record _⇌_ (x y : Set) : Set where
  field ↼ : x → y; ⇀ : y → x
open _⇌_

sym : ∀ {x y} → x ⇌ y → y ⇌ x
sym x⇌y .↼ x = x⇌y .⇀ x
sym x⇌y .⇀ y = x⇌y .↼ y

trans : ∀ {x y z} → x ⇌ y → y ⇌ z → x ⇌ z
trans x⇌y y⇌z .↼ x = y⇌z .↼ (x⇌y .↼ x)
trans x⇌y y⇌z .⇀ z = x⇌y .⇀ (y⇌z .⇀ z)

refl : ∀ {x} → x ⇌ x
refl .↼ x = x; refl .⇀ x = x
```

The Correct-by-Construction Approach

The Correct-by-Construction Approach

Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq.

In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg

```
Inductive Pol (C:Set) : Set :=
  | Pc : C -> Pol C
  | Pinj : positive -> Pol C -> Pol C
  | PX : Pol C -> positive -> Pol C -> Pol C.
```

The Correct-by-Construction Approach

Franck Slama and Edwin Brady. Automatically Proving Equivalence by Type-Safe Reflection.

In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, volume 10383, pages 40–55. Springer International Publishing, Cham, 2017

The Correct-by-Construction Approach

```
data Poly : Carrier → Set (a ⊔ ℓ) where
  ⟦⟧ : Poly 0#
  ⟦_∷_⟧
    : ∀ x {xs}
    → Poly xs
    → Poly (λ ρ → x Coeff.+ ρ Coeff.* xs ρ)

infixr 0 _⇐_
record Expr (expr : Carrier) : Set (a ⊔ ℓ) where
  constructor _⇐_
  field
    {norm} : Carrier
    poly   : Poly norm
    proof  : expr ≋ norm
```

Questions?