Automatically And Efficiently Illustrating Polynomial Equalities in Agda

Donnacha Oisín Kidney

January 11, 2019

Abstract

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings and semirings in the programming language Agda [16]. We use Agda's reflection machinery to provide a simple interface to the solver, and demonstrate a novel use of the constructed relations.

Contents

1 Introduction

Truly formal proofs of even basic mathematical identities are notoriously tedious and verbose. Perhaps the canonical example is Russell and Whitehead's proof that $1 + 1 = 2$, which finally arrives on page 379 of Principia Mathematica [20].

More modern systems have greatly simplified the underlying formalisms, but they still often suffer from a degree of explicitness that makes even elementary identities daunting. Dependently-typed programming languages like Agda [16] and Coq [18] are examples of such systems: used in the naïve way, equivalence proofs require the programmer to specify every individual step ("here we rely on the commutativity of +, followed by the associativity of × on its right-hand-side", and so on).

Coq and Agda are not just programming languages in name, though: they are fully-fledged and powerful, capable of producing useful software, including automated computer-algebra systems. Unlike most CASs, those written in Coq or Agda come with added guarantees of correctness in their operation. Furthermore, these systems can be used to automate the construction of identity proofs which would otherwise be too tedious to do by hand.

1.1 Related Work

The state-of-the-art solver for polynomial equalities (over commutative rings) was originally presented in [8], and is used in Coq's `ring` solver. This work improved on the already existing solver [5] in both efficiency and flexibility. In both the old and improved solvers, a reflexive technique is used to automate the construction of the proof obligation (as described in [1]).

Agda [16] is a dependently-typed programming language based on Martin-Löf's Intuitionistic Type Theory [11]. Its standard library [7] currently contains a ring solver which is similar in flexibility to Coq's `ring`, but doesn't support the reflection-based interface, and is less efficient due to its use of a dense (rather than sparse) internal data structure.

In [17], an implementation of an automated solver for the dependently-typed language Idris [2] is described. It uses type-safe reflection to provide a simple and elegant interface, and its internal solver algorithm uses a correct-by-construction approach. The solver is defined over *non*commutative rings, however, meaning that it is more general (can work with more types) but less powerful (meaning it can prove fewer identities). It does not use a sparse representation.

Reflection and metaprogramming are relatively recent additions to Agda, but form an important part of the interfaces to automated proof procedures. Reflection in dependent types in general is explored in [4], and specific to Agda in [19].

The progress of various formalization efforts is charted in [21]. DoCon [14] is a notable Agda library in this regard: its implementation and goal is described in [13]. [3] describes the manipulation of polynomials in both Haskell and Agda.

Finally, the study of *didactic* computer algebra systems is explored in [10].

1.2 Contributions

An New, Efficient Ring Solver We provide an implementation of a polynomial solver which uses the same optimizations described in [8] in the programming language Agda.

Techniques For Efficient Verification We demonstrate several techniques to thread verification and proof logic through algorithms *without* changing complexity class. These techniques are of general use in functional languages with type systems powerful enough to express invariants.

We also demonstrate a use of the Algebra of Programming approach in Agda [15].

A Simple Reflection-Based Interface We use Agda's reflection machinery to provide the following interface to the solver:

```
lemma : ∀ x y →
  (x + y) ^ 2 ≈ x ^ 2 + y ^ 2 + 2 * x * y
lemma = solve NatRing
```

It imposes minimal overhead on the user: only the Ring implementation is required, with no need for user implementations of quoting. Despite this, it is generic over any type which implements ring.

A Didactic Computer-Algebra System As a result of the flexibility of the solver, the equivalence relation it constructs can be instantiated into a number of different forms (not just equality, for instance). While this has been exploited in Agda before to generate isomorphisms over containers, we use it here to construct didactic (or "step-by-step") solutions.

2 Explaining The Reflexive Technique With Monoids

Before jumping into commutative rings, we will first illustrate a general technique for automatically constructing equivalence proofs over a simpler algebra—*monoids*.

Definition 2.1 (Monoids)**.** A monoid is a set equipped with a binary operation, $\bullet$, and a distinguished element ϵ, such that the following equations hold:

$$
\begin{aligned}
x \bullet (y \bullet z) &= (x \bullet y) \bullet z && \text{(Associativity)} \\
x \bullet \epsilon &= x && \text{(Left Identity)} \\
\epsilon \bullet x &= x && \text{(Right Identity)}
\end{aligned}
$$

Addition and multiplication (with 0 and 1 being the respective identity elements) are perhaps the most obvious instances of the algebra. In computer science, monoids have proved a useful abstraction for formalizing concurrency (in a sense, an associative operator is one which can be evaluated in any order).

2.1 A "Trivial" Identity

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

Figure 1: A Simple Identity Of Monoids

As a running example for this section, we will use the identity in figure 1. To a human, the fact that the identity holds may well be obvious: • is associative, so we can scrub out all the parentheses, and ε is the identity element, so scrub it out too. After that, both sides are equal, so voilà!

Unfortunately, our compiler isn't nearly that clever. As alluded to before, we need to painstakingly specify every intermediate step, justifying every move:

```
1  ident w x y z =
2    begin
3      ((w • ε) • (x • y)) • z
4    ≈⟨ assoc (w • ε) (x • y) z ⟩
5      (w • ε) • ((x • y) • z)
6    ≈⟨ identityʳ w ⟨ •-cong ⟩ assoc x y z ⟩
7      w • (x • (y • z))
8    ≈⟨ sym (assoc w x (y • z)) ⟩
9      (w • x) • (y • z)
10   ∎
```

The syntax is designed to mimic that of a handwritten proof: line 3 is the expression on the left-hand side of ≈ in the type, and line 9 the right-hand-side. In between, the expression is repeatedly rewritten into equivalent forms, with justification provided inside the angle brackets. For instance, to translate the expression from the form on line 3 to that on line 5, the associative property of • is used on line 4.

One trick worth pointing out is on line 6: the •-cong lifts two equalities to either side of a •. In other words, given a proof that $x_1 \approx x_2$, and $y_1 \approx y_2$, it will provide a proof that $x_1 \bullet y_1 \approx x_2 \bullet y_2$. This function needs to be explicitly provided by the user, as we only require ≈ to be an equivalence relation (not just propositional equality). In other words, we don't require it to be substitutive.

2.2 ASTs for the Language of Monoids

The first hurdle for automatically constructing proofs comes from the fact that the identity is opaque: it's hidden behind a lambda. We can't scrutinize or pattern-match on its contents. Our first step, then, is to define an AST for these expressions which we *can* pattern-match on:

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

We have constructors for both monoid operations, and a way to refer to variables. These are referred to by their de Bruijn indices (the type itself is indexed by the number of variables it contains). Here is how we would represent the left-hand-side of the identity in figure 1:

```
((0 ⊕ e) ⊕ (1 ⊕ 2)) ⊕ 3
```

To get *back* to the original expression, we can write an “evaluator”:

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

This performs no normalization, and as such its refult is *definitionally* equal to the original expression[1]:

```
definitional
  : ∀ {w x y z}
  → (w ∙ x) ∙ (y ∙ z)
    ≈ ⟦ (0 ⊕ 1) ⊕ (2 ⊕ 3) ⟧
      (w ∷ x ∷ y ∷ z ∷ [])
definitional = refl
```

We’ve thoroughly set the table now, but we still don’t have a solver. What’s missing is another evaluation function: one that normalizes.

2.3 Canonical Forms

In both the monoid and ring solver, we will make use of the *canonical forms* of expressions in each algebra. Like the AST we defined above, these canonical forms represent expressions in the algebra, however *unlike* the AST, they definitionally obey the laws of the algebra.

For monoids, the canonical form is *lists*.

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

ε here is simply the empty list, and ∙ is concatenation:

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

Similarly to the previous AST, it has variables and is indexed by the number of variables it contains. Its evaluation will be recognizable to functional programmers as the foldr function:

```
_μ_ : ∀ {i} → List i → Vec Carrier i → Carrier
xs μ ρ = foldr (λ x xs → lookup x ρ ∙ xs) ε xs
```

And finally (as promised) the opening identity is *definitionally* true when written in this language:

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

Now, to “evaluate” a monoid expression in a *normalized* way, we simply first convert to the language of lists:

```
norm : ∀ {i} → Expr i → List i
norm (x ⊕ y) = norm x ++ norm y
norm e       = []
norm (ν x)   = η x
```

Or, combining both steps into one:

```
⟦_⇓⟧ : ∀ {i}
      → Expr i
      → Vec Carrier i
      → Carrier
⟦ x ⇓⟧ ρ = norm x μ ρ
```

[1] The type of the unnormalized expression has changed slightly: instead of being a curried function of n arguments, it’s now a function which takes a vector of length n. The final solver has an extra translation step for going between these two representations, but it’s a little fiddly, and not directly relevant to what we’re doing here, so we’ve glossed over it. We refer the interested reader to the Relation.Binary.Reflection module of Agda’s standard library [7] for an implementation.

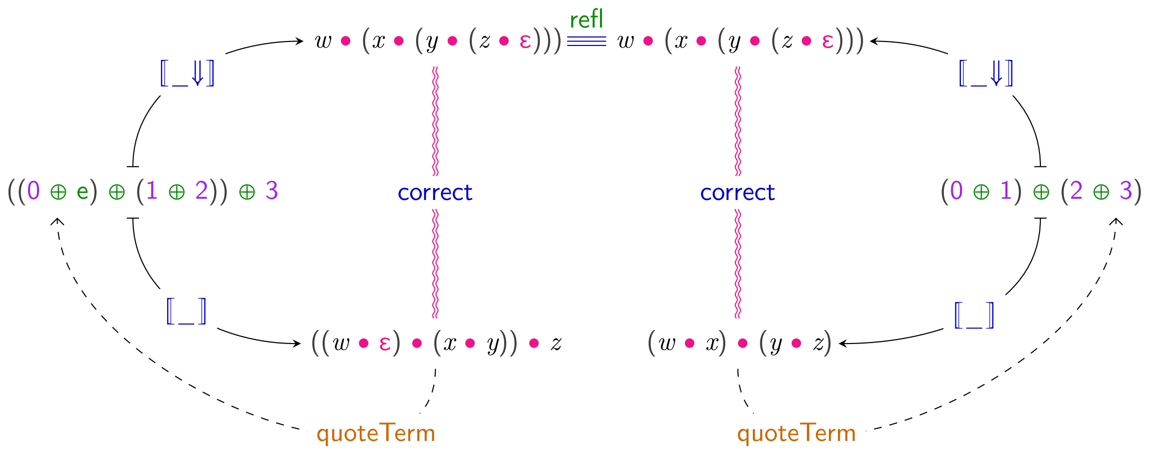

Figure 2: The Reflexive Proof Process

2.4 Homomorphism

Now we have a concrete way to link the normalized and non-normalized forms of the expressions. A diagram of the strategy for constructing our proof is in figure 2. The goal is to construct a proof of equivalence between the two expressions at the bottom: to do this, we first construct the AST which represents the two expressions (for now, we'll assume the user constructs this AST themselves. Later we'll see how too construct it automatically from the provided expressions). Then, we can evaluate it into either the normalized form, or the unnormalized form. Since the normalized forms are syntactically equal, all we need is refl to prove their equality. The only missing part now is correct, which is the task of this section.

Taking the non-normalizing interpreter as a template, the three cases are as follows[2]:

$$[\![\, x \oplus y \,]\!]\, \rho \approx [\![\, x \oplus y \Downarrow]\!]\, \rho \tag{1}$$

$$[\![\, \mathsf{e} \,]\!]\, \rho \approx [\![\, \mathsf{e} \Downarrow]\!]\, \rho \tag{2}$$

$$[\![\, \nu\, i \,]\!]\, \rho \approx [\![\, \nu\, i \Downarrow]\!]\, \rho \tag{3}$$

Proving each of these cases in turn finally verifies the correctness of our list language.

```
++-hom : ∀ {i} (x y : List i)
       → (ρ : Vec Carrier i)
       → (x ++ y) μ ρ ≈ x μ ρ ∙ y μ ρ
++-hom [] y ρ = sym (identityˡ _)
++-hom (x :: xs) y ρ =
  begin
    lookup x ρ ∙ (xs ++ y) μ ρ
  ≈⟨ refl ⟨ ∙-cong ⟩ ++-hom xs y ρ ⟩
    lookup x ρ ∙ (xs μ ρ ∙ y μ ρ)
  ≈⟨ sym (assoc _ _ _) ⟩
    lookup x ρ ∙ xs μ ρ ∙ y μ ρ
  ∎
```

```
correct : ∀ {i}
        → (x : Expr i)
        → (ρ : Vec Carrier i)
        → ⟦ x ⇓⟧ ρ ≈ ⟦ x ⟧ ρ
correct (x ⊕ y) ρ =
  begin
    (norm x ++ norm y) μ ρ
  ≈⟨ ++-hom (norm x) (norm y) ρ ⟩
    norm x μ ρ ∙ norm y μ ρ
  ≈⟨ correct x ρ ⟨ ∙-cong ⟩ correct y ρ ⟩
```

[2] Equations 1 and 2 comprise a monoid homomorphism.

```
    ⟦ x ⟧ ρ ∙ ⟦ y ⟧ ρ
  ■
correct e ρ = refl
correct (ν x) ρ = identityʳ _
```

2.5 Usage

Combining all of the components above, with some plumbing provided by the Relation.Binary.Reflection module, we can finally automate the solving of the original identity in figure 1:

```
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
```

2.5.1 Reflection

While the procedure is now automated, the interface isn't ideal: users have to write the identity they want to prove *and* the AST representing the identity. Removing this step is the job of reflection (section 4): in figure 2 it's represented by the path labeled quoteTerm.

3 A Polynomial Solver

We now know the components required for an automatic solver for some algebra: a canonical form, a concrete representation of expressions, and a proof of correctness (homomorphism). We now turn our focus to polynomials.

3.1 Choice of Algebra

So far, we've assumed the solver is defined over commutative rings. That wasn't the only algebra available to us when writing a solver, though: we've demonstrated techniques using monoids in the previous section, and indeed [17] uses *non*commutative rings as its algebra. Here, we will justify our[3]choice (and admit to a minor lie).

Because we want to solve arithmetic equations, we will need the basic operations of addition, multiplication, subtraction, and exponentiation (to a power in ℕ). This is only half of the story, though: along with those operations we will need to specify the laws or equations that they obey (commutativity, associativity, etc.). Here we need balance: the more equations specified, the more equalities the solver can prove, but the fewer types the solver will be available for.

The elephant in the room here is ℕ: perhaps the most used numeric type in Agda, it doesn't have an additive inverse. So that our solver will still function with it as a carrier type, we don't require

$$x - x = 0$$

to hold. This lets us lawfully define negation as the identity function for ℕ.

A potential worry is that because we don't require $x - x = 0$ axiomatically, it won't be provable in our system. This is not so: as is pointed out in [8],as long as $1 - 1$ reduces to 0 in the coefficient set, the solver will verify the identity.

3.2 Horner Normal Form

The canonical representation of polynomials is a list of coefficients, least significant first ("Horner Normal Form"). Our initial attempt at encoding this representation will begin like so:

```
open import Algebra

module HornerNormalForm
  {c} (coeff : RawRing c) where
```

The entire module is parameterized by the choice of coefficient. This coefficient should support the ring operations, but it is "raw", i.e. it doesn't prove the ring laws. The operations[4]on the polynomial itself are defined in figure 3.

[3] "Our" choice here is the same choice as in [8].

[4] Symbols chosen for operators use the following mnemonic:

1. Operators preceded with "ℕ." are defined over ℕ; e.g. ℕ.+, ℕ.∗.

```
Poly : Set c
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) []
```

Figure 3: Simple Operations on Dense Horner Normal Form

Finally, evaluation of the polynomial uses "Horner's rule" to minimize multiplications:

```
⟦_⟧ : Poly → Carrier → Carrier
⟦ x ⟧ ρ = foldr (λ y ys → y + ρ * ys) 0# x
```

3.3 Eliminating Redundancy

As it stands, the above representation has two problems:

Redundancy The representation suffers from the problem of trailing zeroes. In other words, the polynomial $2x$ could be represented by any of the following:

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

This is a problem for a solver: the whole *point* is that equivalent expressions are represented the same way.

Inefficiency Expressions will tend to have large gaps, full only of zeroes. Something like x^5 will be represented as a list with 6 elements, only the last one being of interest. Since addition is linear in the length of the list, and multiplication quadratic, this is a major concern.

In [8], the problem is addressed primarily from the efficiency perspective: they add a field for the "power index". For our case, we'll just store a list of pairs, where the second element of the pair is the power index[5].

As an example, the polynomial:

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

Will be represented as:

$$(3, 0), (2, 1), (4, 2), (2, 1)$$

Or, mathematically:

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

Definition 3.1 (Dense and Sparse Encodings)**.** In situations like this, where inductive types have large "gaps" of zero-like terms between interesting (non-zero-like) terms, the encoding which uses an index to represent the distance to the next interesting term will be called *sparse*, and the encoding which simply stores the zero term will be called *dense*.

3.3.1 Uniqueness

While this form solves our efficiency problem, we still have redundant representations of the same polynomials. In [8], care is taken to ensure all operations include a normalizing step, but this is not verified: in other words, it is not proven that the polynomials are always in normal form.

[2] Plain operators, like + and $*$, are defined over the coefficients.

[3] Boxed operators, like ⊞ and ⊠, are defined over polynomials.

[5] In [8], the expression $(c, i) :: P$ represents $P \times X^i + c$. We found that $X^i \times (c + X \times P)$ is a more natural translation, and it's what we use here. A power index of i in this representation is equivalent to a power index of $i + 1$ in [8].

Expressing that a polynomial is in normal form turns out to be as simple as disallowing zeroes: without them, there can be no trailing zeroes, and all gaps must be represented by power indices. To check for zero, we require the user supply a decidable predicate on the coefficients. This changes the module declaration like so:

```
open import Algebra

module EliminatingRedundancy
  {c ℓ}
  (coeffs : RawRing c)
  (Zero : Pred (RawRing.Carrier coeffs) ℓ)
  (zero? : Decidable Zero) where

open RawRing coeffs
```

Importantly, we don't require that the user provides a decidable proof of *equivalence*, rather just a decidable proof of some predicate which can later be translated into an equivalence with zero. Functionally, this means the user could supply a predicate which is always false, or a predicate which is only *weakly* decidable.

And now we have a definition for sparse polynomials:

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

infixl 6 _Δ_
record PowInd {c} (C : Set c) : Set c where
  constructor _Δ_
  field
    coeff : C
    power : ℕ
open PowInd

Poly : Set (c ⊔ ℓ)
Poly = List (PowInd Coeff)
```

The proof of nonzero is marked irrelevant (preceded with a dot) to avoid computing it at runtime.

We can wrap up the implementation with a cleaner interface by providing a normalizing version of _::_:

```
infixr 8 _⍍_
_⍍_ : Poly → ℕ → Poly
[] ⍍ i = []
(x Δ j :: xs) ⍍ i = x Δ (j ℕ.+ i) :: xs

infixr 5 _::↓_
_::↓_ : PowInd Carrier → Poly → Poly
x Δ i ::↓ xs with zero? x
... | yes p = xs ⍍ suc i
... | no ¬p = _≠0 x {¬p} Δ i :: xs
```

3.4 A Sparse Encoding for Multiple Variables

So far, the polynomials have been (suspiciously) single-variable. Luckily, there's a natural technique to support multiple variables: for a polynomial with n variables, it has coefficients of $n - 1$ variables. In types:

```
record Coeff n : Set (c ⊔ ℓ) where
  inductive
  constructor _≠0
  field
    coeff : Poly n
    .{coeff≠0} : ¬Zero coeff

record PowInd {c} (C : Set c) : Set c where
  inductive
  constructor _Δ_
  field
    coeff : C
    power : ℕ

Poly : ℕ → Set (c ⊔ ℓ)
Poly zero = Lift ℓ Carrier
Poly (suc n) = List (PowInd (Coeff n))

¬Zero : ∀ {n} → Poly n → Set ℓ
¬Zero {zero} (lift lower) = ¬ Zero lower
¬Zero {suc n} [] = Lift _ ⊥
¬Zero {suc n} (x :: xs) = Lift _ ⊤
```

However, this encoding is again a dense one. In a polynomial of n variables, addressing the n^{th} variable needlessly requires $n-1$ layers of nesting. Alternatively, a constant expression in this polynomial is hidden behind n layers of nesting.

In contrast to the previous sparse encoding, though, the size of the gap is type-relevant. Because of this, the gap will have to be lifted into an index (figure 4).

This "type-relevant" gap will present some problems later on, but we will leave the definition as it is for now.

3.5 Efficiency in Indexed Types

3.5.1 Call-Pattern Specialization

While both sparse encodings provide a more space-efficient representation, the computational efficiency has yet to be realized. Starting with the sparse monomial, we'll look at the addition function to start off. In the dense encoding (figure 3), we needed to line up corresponding coefficients to add together. For this encoding, the "corresponding" coefficients are slightly harder to find. In order to line things p correctly, we'll need to compare the gap indices. This, however, presents our first problem:

```
_⊞_ : Poly → Poly → Poly
[]  ⊞ ys = ys
xs ⊞ []  = xs
(x Δ i :: xs) ⊞ (y Δ j :: ys) with compare i j
... | less      i k = x Δ i :: xs ⊞ (y Δ k :: ys)
... | greater j k = y Δ j :: (x Δ k :: xs) ⊞ ys
... | equal   i = coeff x + coeff y Δ i ::↓ xs ⊞ ys
```

The above definition won't pass the termination checker. While it does indeed terminate, it isn't structurally decreasing in its arguments. To argue our case that the function does terminate, we have to reveal this fact to the compiler using a well-known optimization for functional languages called "call-pattern specialization" [9].

Principle 3.1 (To make termination obvious, perform call-pattern specialization)**.** Unpack any constructors into function arguments as soon as possible, and eliminate any redundant pattern matches in

```
infixl 6 _Δ_
record PowInd {c} (C : Set c) : Set c where
  inductive
  constructor _Δ_
  field
    coeff : C
    pow : ℕ

mutual
  infixl 6 _Π_
  data Poly : ℕ → Set (c ⊔ ℓ) where
    _Π_ : ∀ {j}
          → FlatPoly j
          → ∀ i
          → Poly (suc (i ℕ.+ j))

  data FlatPoly : ℕ → Set (c ⊔ ℓ) where
    K : Carrier → FlatPoly zero
    Σ : ∀ {n}
      → (xs : Coeffs n)
      → .{xn : Norm xs}
      → FlatPoly (suc n)

  Coeffs : ℕ → Set (c ⊔ ℓ)
  Coeffs = List ∘ PowInd ∘ NonZero

  infixl 6 _≠0
  record NonZero (i : ℕ) : Set (c ⊔ ℓ) where
    inductive
    constructor _≠0
    field
      poly : Poly i
      .{poly≠0} : ¬ ZeroPoly poly
```

Figure 4: A Sparse Multivariate Polynomial

the offending functions. Happily, this transformation both makes termination more obvious *and* improves performance.

This transformation is performed automatically by GHC as an optimization: perhaps a similar transformation could be performed by Agda's termination checker to reveal more terminating programs.

For our case, the principle applied can be seen in figure 5.

3.5.2 Built-In Functions

The second optimization we might rely on involves the call to compare. This is a classic "leftist" function: it returns an *indexed* data type (figure 6). The compare function itself is $\mathcal{O}(\min(n, m))$:

```
compare : ∀ m n → Ordering m n
compare zero    zero    = equal   zero
compare (suc m) zero    = greater zero m
compare zero    (suc n) = less    zero n
compare (suc m) (suc n) with compare m n
... | less    m k = less    (suc m) k
... | equal   m   = equal   (suc m)
... | greater n k = greater (suc n) k
```

The implementation of compare may raise suspicion with regards to efficiency: if this encoding of polynomials improves time complexity by skipping the gaps, don't we lose all of that when we encode the gaps as Peano numbers?

The answer is a tentative no. Firstly, since we are comparing gaps, the complexity can be no larger than that of the dense implementation. Secondly, the operations we're most concerned about are those on the underlying coefficient; and, indeed, this sparse encoding does reduce the number of those significantly. Thirdly, if a fast implementation of compare is really and truly demanded, there are tricks we can employ.

Agda has a number of built-in functions on the natural numbers: when applied to closed terms, these call to an implementation on Haskell's `Integer` type, rather than the unary implementation. For our uses, the functions of interest are -, +, <, and ==. The comparison functions provide booleans rather than evidence, but we can prove they correspond to the evidence-providing versions:

```
lt-hom : ∀ n m
       → ((n < m) ≡ true)
       → m ≡ suc (n + (m - n - 1))
lt-hom zero    zero    ()
lt-hom zero    (suc m) _   = refl
lt-hom (suc n) zero    ()
lt-hom (suc n) (suc m) n<m =
  cong suc (lt-hom n m n<m)

eq-hom : ∀ n m
       → ((n == m) ≡ true)
       → n ≡ m
eq-hom zero    zero    _   = refl
eq-hom zero    (suc m) ()
eq-hom (suc n) zero    ()
eq-hom (suc n) (suc m) n≡m =
  cong suc (eq-hom n m n≡m)

gt-hom : ∀ n m
       → ((n < m) ≡ false)
       → ((n == m) ≡ false)
       → n ≡ suc (m + (n - m - 1))
gt-hom zero    zero    n<m ()
gt-hom zero    (suc m) ()  n≡m
gt-hom (suc n) zero    n<m n≡m = refl
gt-hom (suc n) (suc m) n<m n≡m =
  cong suc (gt-hom n m n<m n≡m)
```

Combined with judicious use of erase and inspect, we get the implementation which can be seen in figure 7.

3.5.3 Unification

Now we return to the type-relevant *injection* indices. We encoded the information in figure 4. It encodes "less than" in the same way that the ordering type did (figure 6), so it may seem (initially) like a perfect fit. However, we run into issues when it comes to performing the comparison-like operations above. Because it's an indexed type, pattern matching on it will force unification of the index with whatever type variable it was bound to. This is problematic because the index is defined by a function: pattern match on a pair of Polys and you're asking Agda to unify $i_1 + j_1$

```
infixl 6 _⊞_
_⊞_ : Poly → Poly → Poly
[] ⊞ ys = ys
(x :: xs) ⊞ ys = ⟨ x :: xs ⊞ ys ⟩
  where
  ⟨_::_⊞_⟩ : PowInd Coeff → Poly → Poly → Poly
  _⊢⟨_::_⊞_::_⟩ : ∀ {i j} → Ordering i j → Coeff → Poly → Coeff → Poly → Poly

  less    i k ⊢⟨ x :: xs ⊞ y :: ys ⟩ = x Δ i :: ⟨ y Δ k :: ys ⊞ xs ⟩
  greater j k ⊢⟨ x :: xs ⊞ y :: ys ⟩ = y Δ j :: ⟨ x Δ k :: xs ⊞ ys ⟩
  equal     k ⊢⟨ x :: xs ⊞ y :: ys ⟩ = coeff x + coeff y Δ k ::↓ xs ⊞ ys

  ⟨ x :: xs ⊞ [] ⟩ = x :: xs
  ⟨ x Δ i :: xs ⊞ y Δ j :: ys ⟩ = compare i j ⊢⟨ x :: xs ⊞ y :: ys ⟩
```

Figure 5: Termination by Call-Pattern Specialization

```
data Ordering : ℕ → ℕ → Set where
  less    : ∀ m k → Ordering m (suc (m + k))
  equal   : ∀ m   → Ordering m m
  greater : ∀ m k → Ordering (suc (m + k)) m
```

Figure 6: The Ordering Indexed Type

and $i_2 + j_2$, a task it will likely find too difficult. How do we avoid this?

Principle 3.2 (Don't touch the green slime!)**.** When combining prescriptive and descriptive indices, ensure both are in constructor form. Exclude defined functions which yield difficult unification problems [12].

We'll have to take another route.

3.5.4 Hanging Indices

First, we'll redefine our polynomial like so:

```
record Poly (i : ℕ) : Set (a ⊔ ℓ) where
  inductive
  constructor _Π_
  field
    {j} : ℕ
    flat : FlatPoly j
    j≤i  : j ≤ i
```

The type is now parameterized, rather than indexed: our pattern-matching woes have been solved. Also, the gap is now implicit; instead, we store a proof that the nested polynomial has no more variables then the outer.

The definition for this proof has important performance implications, as the proof will need to mesh with whatever comparison function we use for the injection indices. As a jumping-off point, we'll look at the three definitions of ≤ in the Agda standard library [7].

Option 1: The Standard Way The most commonly used definition of ≤ is as follows:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n}
      → (m≤n : m ≤ n)
      → suc m ≤ suc n
```

Trying to proceed with this type will yield a nasty performance bug, though: the inductive structure of the type gives us no real information about the underlying "gap", so we're forced

```
compare : ∀ n m → Ordering n m
compare n m with n < m | inspect (n <_) m
... | true  | [ n<m ] rewrite erase (lt-hom n m n<m)        = less n (m − n − 1)
... | false | [ n≮m ] with n == m | inspect (n ==_) m
... | true  | [ n≡m ] rewrite erase (eq-hom n m n≡m)        = equal m
... | false | [ n≢m ] rewrite erase (gt-hom n m n≮m n≢m) = greater m (n − m − 1)
```

Figure 7: Fast comparison function using built-in functions on the natural numbers

to compare the actual size of the nested polynomials. To see why this is a problem, consider the following sequence of nestings:

$$(5 \leq 6), (4 \leq 5), (3 \leq 4), (1 \leq 3), (0 \leq 1)$$

The outer polynomial has 6 variables, but it has a gap to its inner polynomial of 5, and so on. The comparisons will be made on 5, 4, 3, 1, and 0. Like repeatedly taking the length of the tail of a list, this is quadratic. There must be a better way.

Option 2: With Propositional Equality Once you realize we need to be comparing the gaps and not the tails, another encoding of ≤ in Data.Nat seems the best option:

```
record _≤_ (m n : ℕ) : Set where
  constructor less-than-or-equal
  field
    {k} : ℕ
    proof : m + k ≡ n
```

It stores the gap *right there*: in k!

Unfortunately, though, we're still stuck. While you can indeed run your comparison on k, you're not left with much information about the rest. Say, for instance, you find out that two respective ks are equal. What about the ms? Of course, you *can* show that they must be equal as well, but it requires a proof. Similarly in the less-than or greater-than cases: each time, you need to show that the information about k corresponds to information about m. Again, all of this can be done, but it all requires propositional proofs, which are messy, and slow. Erasure is an option, but I'm not sure of the correctness of that approach.

Option 3 What we really want is to *run* the comparison function on the gap, but get the result on the tail. Turns out we can do exactly that with the following:

```
data _≤_ (m : ℕ) : ℕ → Set where
  m≤m : m ≤ m
  ≤-s   : ∀ {n}
         → (m≤n : m ≤ n)
         → m ≤ suc n
```

This is the structure we will choose.

What's important about our chosen type is that, ignoring the indices, its inductive structure mimics that of the actual Peano encoding of the gaps previously. In other words, m≤m appears wherever zero would have previously, and ≤-s where there was suc. This gives us another principle:

Principle 3.3 (To add more type information, to a type or function, keep the *structure* of the old type, while *hanging* new information off of it)**.** The three options above each present avenues to possible solutions to our "gap" problem, but they should have been ignored. Instead, we should have taken the previous untyped solution, and seen where in the inductive cases of the types used extra type information could have been stored. With this approach, the efficiency of the already-written algorithms is maintained. This practice can be somewhat automated using *ornaments* [6].

This is not yet enough to fully write our comparison function, though. Looking back to the previous definition of Ordering, we see that it contains +, something we'll have to figure out an equivalent for in terms of ≤. It turns out that equivalence is *transitivity*:

```
≤-trans : ∀ {x y z} → x ≤ y → y ≤ z → x ≤ z
≤-trans x≤y m≤m = x≤y
≤-trans x≤y (≤-s y≤z) = ≤-s (≤-trans x≤y y≤z)
```

And with that, we have enough to define our comparison function:

```
data ≤-Ordering {n : ℕ} : ∀ {i j}
                        → (i≤n : i ≤ n)
                        → (j≤n : j ≤ n)
                        → Set
  where
  ≤-lt  : ∀ {i j-1}
        → (i≤j-1 : i ≤ j-1)
        → (j≤n : suc j-1 ≤ n)
        → ≤-Ordering (≤-trans (≤-s i≤j-1) j≤n)
                     j≤n
  ≤-gt  : ∀ {i-1 j}
        → (i≤n : suc i-1 ≤ n)
        → (j≤i-1 : j ≤ i-1)
        → ≤-Ordering i≤n
                     (≤-trans (≤-s j≤i-1) i≤n)
  ≤-eq  : ∀ {i}
        → (i≤n : i ≤ n)
        → ≤-Ordering i≤n
                     i≤n

≤-compare : ∀ {i j n}
          → (x : i ≤ n)
          → (y : j ≤ n)
          → ≤-Ordering x y
≤-compare m≤m     m≤m     = ≤-eq m≤m
≤-compare m≤m     (≤-s y) = ≤-gt m≤m y
≤-compare (≤-s x) m≤m     = ≤-lt x m≤m
≤-compare (≤-s x) (≤-s y)
  with ≤-compare x y
... | ≤-lt i≤j-1 _     = ≤-lt i≤j-1 (≤-s y)
... | ≤-gt _ j≤i-1     = ≤-gt (≤-s x) j≤i-1
... | ≤-eq _           = ≤-eq (≤-s x)
```

3.6 Abstraction and Folds for Simpler Proofs

3.7 Proving Higher-Order Termination With Well-Founded Recursion

4 Reflection

5 Setoid Applications

5.1 Isomorphisms

5.2 Didactic Solutions

6 The Correct-By-Construction Approach

References

[1] S. Boutin, "Using reflection to build efficient and certified decision procedures," in *Theoretical Aspects of Computer Software*, ser. Lecture Notes in Computer Science, M. Abadi and T. Ito, Eds. Springer Berlin Heidelberg, 1997, pp. 515–529.

[2] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 05, pp. 552–593, Sep. 2013. [Online]. Available: http://journals.cambridge.org/article_S095679681300018X

[3] C.-M. Cheng, R.-L. Hsu, and S.-C. Mu, "Functional Pearl: Folding Polynomials of Polynomials," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science. Springer, Cham, May 2018, pp. 68–83. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-90686-7_5

[4] D. R. Christiansen, "Practical Reflection and Metaprogramming for Dependent Types," Ph.D. dissertation, IT University of Copenhagen, Nov. 2015. [On-

line]. Available: http://davidchristiansen.dk/david-christiansen-phd.pdf

[5] T. Coq Development Team, *The Coq Proof Assistant Reference Manual, Version 7.2*, 2002. [Online]. Available: http://coq.inria.fr

[6] P.-E. Dagand, "The essence of ornaments," *Journal of Functional Programming*, vol. 27, 2017/ed. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/essence-of-ornaments/4D2DF6F4FE23599C8C1FEA6C921A3748

[7] N. A. Danielsson, "The Agda standard library," Jun. 2018. [Online]. Available: https://agda.github.io/agda-stdlib/README.html

[8] B. Grégoire and A. Mahboubi, "Proving Equalities in a Commutative Ring Done Right in Coq," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, vol. 3603. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 98–113. [Online]. Available: http://link.springer.com/10.1007/11541868_7

[9] S. P. Jones, "Call-pattern specialisation for haskell programs." ACM Press, 2007, p. 327. [Online]. Available: https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions-2/

[10] D. Lioubartsev, "Constructing a Computer Algebra System Capable of Generating Pedagogical Step-by-Step Solutions," Ph.D. dissertation, KTH Royal Institue of Technology, Stockholm, Sweden, 2016. [Online]. Available: http://www.diva-portal.se/smash/get/diva2:945222/FULLTEXT01.pdf

[11] P. Martin-Löf, *Intuitionistic Type Theory*, Padua, Jun. 1980. [Online]. Available: http://www.cse.chalmers.se/~peterd/papers/MartinL%00f6f1984.pdf

[12] C. McBride, "A Polynomial Testing Principle," Jul. 2018. [Online]. Available: https://twitter.com/pigworker/status/1013535783234473984

[13] S. D. Meshveliani, "Dependent Types for an Adequate Programming of Algebra," Program Systems Institute of Russian Academy of sciences, Pereslavl-Zalessky, Russia, Tech. Rep., 2013. [Online]. Available: http://ceur-ws.org/Vol-1010/paper-05.pdf

[14] ——, "DoCon-A a Provable Algebraic Domain Constructor," Pereslavl - Zalessky, Apr. 2018. [Online]. Available: http://www.botik.ru/pub/local/Mechveliani/docon-A/2.02/

[15] S.-C. Mu, H.-S. Ko, and P. Jansson, "Algebra of programming in Agda: Dependent types for relational program derivation," *Journal of Functional Programming*, vol. 19, no. 5, pp. 545–579, Sep. 2009. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/algebra-of-programming-in-agda-dependent-types-for-relational-ACA0C08F29621A892FB0C0B745254D15

[16] U. Norell and J. Chapman, "Dependently Typed Programming in Agda," p. 41, 2008.

[17] F. Slama and E. Brady, "Automatically Proving Equivalence by Type-Safe Reflection," in *Intelligent Computer Mathematics*, H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, Eds. Cham: Springer International Publishing, 2017, vol. 10383, pp. 40–55. [Online]. Available: http://link.springer.com/10.1007/978-3-319-62075-6_4

[18] T. C. D. Team, "The Coq Proof Assistant, version 8.8.0," Apr. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1219885

[19] P. D. van der Walt, "Reflection in Agda," Master's Thesis, Universiteit of Utrecht, Oct. 2012. [Online]. Available: https://dspace.library.uu.nl/handle/1874/256628

[20] A. N. Whitehead and B. Russell, *Principia Mathematica. Vol. I*, 1910. [Online]. Available: https://zbmath.org/?q=an%3A41.0083.02

[21] F. Wiedijk, "Formalizing 100 Theorems," Oct. 2018. [Online]. Available: http://www.cs.ru.nl/~freek/100/