Automatically And Efficiently Illustrating Polynomial Equalities in Agda

Donnacha Oisín Kidney

January 13, 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 [19]. 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 [24].

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 [19] and Coq [21] 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 [19] is a dependently-typed programming language based on Martin-Löf's Intuitionistic Type Theory [12]. 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 [20], 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 [23].

The progress of various formalization efforts is charted in [25]. DoCon [15] is a notable Agda library in this regard: its implementation and goal is described in [14]. [3] describes the manipulation of polynomials in both Haskell and Agda.

Finally, the study of *didactic* computer algebra systems is explored in [11].

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 [16].

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:

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z \qquad \text{(Associativity)}$$
$$x \bullet \epsilon = x \qquad \text{(Left Identity)}$$
$$\epsilon \bullet x = x \qquad \text{(Right Identity)}$$

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: $\bullet$ 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 $\approx$ 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 $\bullet$ is used on line 4.

One trick worth pointing out is on line 6: the ∙-cong lifts two equalities to either side of a $\bullet$. 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 $\approx$ 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.

[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.

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
```

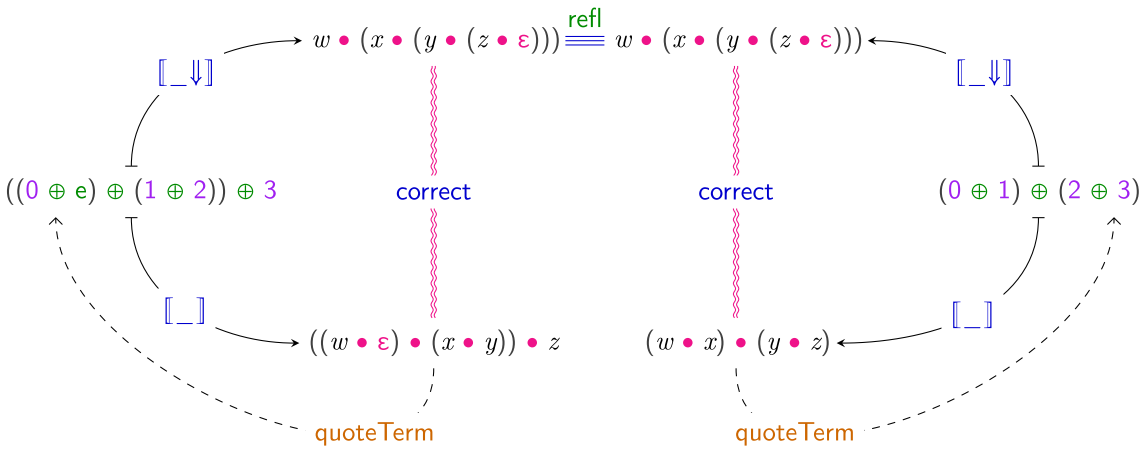

Figure 2: The Reflexive Proof Process

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]:

$$⟦\ x ⊕ y\ ⟧\ \rho \approx ⟦\ x ⊕ y ⇓⟧\ \rho \tag{1}$$

$$⟦\ \mathsf{e}\ ⟧\ \rho \approx ⟦\ \mathsf{e} ⇓⟧\ \rho \tag{2}$$

$$⟦\ \nu\ i\ ⟧\ \rho \approx ⟦\ \nu\ i ⇓⟧\ \rho \tag{3}$$

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

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 ⟩
```

```
    ⟦ 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 [20] 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:

$$0, 2$$
$$0, 2, 0$$
$$0, 2, 0, 0$$
$$0, 2, 0, 0, 0, 0, 0$$

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" [10].

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 [13].

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

At this point, following our many optimizations, the task of proving homomorphism for these implementations is more than a little daunting. However, we can ease the burden somewhat by leveraging the foldr function, in a similar way to [16].

The goal is to modularize the proofs by independently proving the correctness of each optimization. To do this, we will strive to write the arithmetic operations as higher-order functions, which operate over the "unoptimized" version of the polynomials (the dense versions), and have an intermediate function convert to and from the dense encoding. As a case study, we'll work with the negation function. Our initial definition (on the type defined in figure 8) is as follows:

```
⊟_ : ∀ {n} → Poly n → Poly n
⊟ (K x  Π i≤n) = K (- x) Π i≤n
⊟ (Σ xs Π i≤n) = go xs Π↓ i≤n
  where
  go : ∀ {n} → Coeffs n → Coeffs n
  go [] = []
  go (x ≠0 Δ i :: xs) = ⊟ x Δ i ::↓ go xs
```

Immediately we can recognize two functions which are good candidates for separated homomorphism proofs: the two normalizing functions. These functions will be used in every arithmetic operation, so it stands to reason that a separate proof of their correctness should save us some repetition (the semantics of the polynomials are defined in figure 9). These lemmas are as follows:

```
Π↓-hom
  : ∀ {n m}
  → (xs : Coeffs n)
  → (sn≤m : suc n ≤′ m)
  → ∀ ρ
  → ⟦ xs Π↓ sn≤m ⟧ ρ
     ≈ Σ⟦ xs ⟧ (drop-1 sn≤m ρ)

::↓-hom
  : ∀ {n}
  → (x : Poly n)
  → ∀ i xs ρ ρs
  → Σ⟦ x Δ i ∷↓ xs ⟧ (ρ , ρs)
      ≈ ((x , xs) ⟦∷⟧ (ρ , ρs)) * ρ ^ i
```

Next, the go helper function is an obvious candidate for foldr[6].

.

```
⊟_ : ∀ {n} → Poly n → Poly n
⊟ (K x  Π i≤n) = K (- x) Π i≤n
⊟ (Σ xs Π i≤n) = foldr go [] xs Π↓ i≤n
  where
  go = λ { (x ≠0 Δ i) xs → ⊟ x Δ i ∷↓ xs }
```

Continuing in this vein, we are able to isolate the component of this function which is different from the other operations, and therefore confine our proofs to that difference. In this particular case, we use the following lemma:

```
poly-mapR
  : ∀ {n} ρ ρs
  → ([f] : Poly n → Poly n)
  → (f : Carrier → Carrier)
  → (∀ x y → f x * y ≈ f (x * y))
  → (∀ x y → f (x + y) ≈ f x + f y)
  → (∀ y → ⟦ [f] y ⟧ ρs ≈ f (⟦ y ⟧ ρs))
  → (f 0# ≈ 0#)
  → ∀ xs
  → Σ⟦ poly-map [f] xs ⟧ (ρ , ρs)
      ≈ f (Σ⟦ xs ⟧ (ρ , ρs))
```

In this way, we demonstrate a concrete and practical use of [16].

3.7 Proving Higher-Order Termination With Well-Founded Recursion

Unfortunately, by using a higher-order function, we've obscured the fact that the negation operation obviously terminates.

Among the bag of tricks available to Agda programmers to prove termination is what's known as *well-founded recursion* [17]. It works by providing a relation which describes some strictly decreasing finite chain: $<$ on $\mathbb{N}$, for instance. It's strictly decreasing (the first argument always gets smaller, in contrast to, say, $\leq$), and it's finite, because it must end at 0.

It's a powerful tool, which can be used to prove complex termination patterns: multiple relations can be combined lexicographically, for instance, if the recursive function decreases on different arguments in different settings.

In Agda, well-founded recursion is specified with the following type:

```
data Acc {a r}
         {A : Set a}
         (_<_ : A → A → Set r)
         (x : A) : Set (a ⊔ r) where
  acc
    : (∀ y → y < x → Acc _<_ y)
    → Acc _<_ x
```

This type is a way to construct arguments to functions which are strictly decreasing in size. Simply add it as an extra parameter to the dangerous functions, pattern match on acc, and you've provided evidence that an argument is strictly decreasing.

One of the warts of well-founded recursion is that usually the programmer has to separately construct the relation they're interested in. As well as being complex, it can be computationally expensive to do so. Usually the compiler can elide the calls, recognizing that the argument isn't used, but the optimization can't be guaranteed.

Luckily, in our case, the relation is already lying around, in the injection index. So we can just use that!

```
⊟′ : ∀ {n} → Acc _<′_ n → Poly n → Poly n
⊟′ _  (K x Π i≤n) = K (- x) Π i≤n
⊟′ (acc wf) (Σ xs Π i≤n) = foldr go [] xs Π↓ i≤n
  where
  go = λ { (x ≠0 Δ i) xs
          → ⊟′ (wf _ i≤n) x Δ i ∷↓ xs }

⊟_ : ∀ {n} → Poly n → Poly n
⊟_ = ⊟′ (<′-wellFounded _)
```

[6]Using foldr will actually yield some termination problems, which we will have to deal with in the next section

4 Reflection

One annoyance of the automated solver is that we have to write the expression we want to solve twice: once in the type signature, and again in the argument supplied to solve. Agda can infer the type signature, but we would prefer to write the expression in the type signature, and have it infer the argument to solve, as the expression in the type signature is the desired equality, and the argument to solve is something of an implementation detail.

This inference can be accomplished using Agda's reflection mechanisms [23].

Reflection in Agda allows a program to inspect and modify its own code. Here, it will allow us to build the AST representation of an expression from a stated goal in the program, meaning that proofs become as simple as the following:

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

There are three basic components that we'll use for the reflection machinery:

Term The representation of Agda's AST, retrievable via quoteTerm.

Name The representation of identifiers, retrievable via quote.

TC The type-checker monad, which includes scoping and environment information, can raise type errors, unify variables, or provide fresh names. Computations in the TC monad can be run with unquote.

While quote, quoteTerm, and unquote provide all the functionality we need, they're somewhat low-level, so instead we will define *macros*. Macros (in Agda) are essentially syntactic sugar for the above keywords. They're defined by first declaring a macro block, and then defining a function within it which has the return type:

```
Term → TC ⊤
```

The rest of the arguments can be treated normally like any other function, or, if they have the type Term or Name, they're quoted before being passed in. The final argument to the function is the hole representing where the macro was called: to "return" a value you unify it with that hole. As an example, here's a macro to count the number of occurrences of some identifier in an expression:

```
natTerm : ℕ → Term
natTerm zero = con (quote ℕ.zero) []
natTerm (suc i) =
  con
    (quote suc)
    (arg (arg-info visible relevant)
    (natTerm i) :: [])

mutual
  occOf : Name → Term → ℕ
  occOf n (var _ args) = occsOf n args
  occOf n (con c args) = occsOf n args
  occOf n (def f args) with n ≟-Name f
  ... | yes p = suc (occsOf n args)
  ... | no ¬p = occsOf n args
  occOf n (lam v (abs s x)) = occOf n x
  occOf n (pat-lam cs args) = occsOf n args
  occOf n (pi a (abs s x)) = occOf n x
  occOf n _ = 0

  occsOf : Name → List (Arg Term) → ℕ
  occsOf n [] = 0
  occsOf n (arg i x :: xs) =
    occOf n x + occsOf n xs

macro
  occurencesOf : Name
               → Term
               → Term
               → TC ⊤
  occurencesOf nm xs hole =
    unify hole (natTerm (occOf nm xs))

occPlus :
  occurencesOf _+_ (λ x y → 2 + 1 + x + y)
  ≡ 3
occPlus = refl
```

Some of the core characteristics of working with the reflected AST are clear here. Firstly, it's verbose. The natTerm function, for instance, simply gets the syntactic representation of a natural number. Unfortunately, we can't necessarily just call quoteTerm: the returned AST includes all kinds of information about context and environment which can clash with the environment where the macro is instantiated. A large amount of reflection and metaprogramming in Agda unfortunately consists of this kind of boilerplate (currently, at any rate).

Next, it's far less *typed* than it could be. To be clear, it doesn't break type safety: the generated program is still type-checked, but you can generate code with a type error in it without any difficulty. On the other end, the reflected AST doesn't contain as much type information as it could, which is often an annoyance.

Finally, it's fragile. Say we want to solve some expression in ℕ. Converting this to some Expr type will involve, among other things, being able to find functions like + and * in the expression. However, if the user implements AlmostCommutativeRing in the normal way, there'll be two identifiers which refer to each of those functions: one being the original implementation in Data.Nat, and the other being the field in AlmostCommutativeRing. But wait, it gets worse: in actual fact, there may well be *three* identifiers. Remember that the "almost" qualifier in AlmostCommutativeRing refers to the fact that negation isn't required to cancel, allowing us to use types without a notion of negation (like ℕ). With the best of intentions, we may even provide a helper function which takes a Semiring (ring without negation) and converts it into a AlmostCommutativeRing, supplying the identity function for negation. That's where our third identifier comes from: the Semiring type. The third identifier is the field in the Semiring record. If the Semiring record is constructed from other records again (two monoids, say), we get even more identifiers to choose from.

This all makes it difficult to check if a function application is +, because we're only going to look at name equality. The following, for example, is morally the same as the argument to occPlus above, but returns a different argument, because we use an aliased version of +:

```
infixl 6 _plus_
_plus_ : ℕ → ℕ → ℕ
_plus_ = _+_

occWrong :
  occurencesOf _+_
    (λ x y → 2 plus 1 plus x plus y)
  ≡ 0
occWrong = refl
```

So our solver will demand that the user only refer to the functions defined in the record.

Something that isn't visible in the example above the fact that Term uses de Bruijn indices for variables. This means we have to be extra careful about scope.

4.1 Building The AST for Proving

Though Term is itself an AST we could theoretically manipulate and use in the prover, as is demonstrated above it's complex and unwieldy: what we really want is to use a smaller AST for ring expressions, like the one for lists. To that end, we'll need build the Term which will construct it for us.

First, to make things easier on ourselves, we'll define some pattern synonyms:

```
infixr 5 ⟨_⟩∷_ ⟅_⟆∷_
pattern ⟨_⟩∷_ x xs =
  arg (arg-info visible relevant) x ∷ xs
pattern ⟅_⟆∷_ x xs =
  arg (arg-info hidden relevant) x ∷ xs
```

These match visible and hidden arguments, respectively.

Next, we'll need another helper which applies the hidden arguments to the constructors for Expr. There are *three* of these. The first is the universe level, the second is the Carrier type, and the third is the number of variables it's indexed by.

```
infixr 5 ⟅_⋯⟆∷_
⟅_⋯⟆∷_ : ℕ
          → List (Arg Term)
          → List (Arg Term)
```

```
≀ i ⋯∫:: xs =
  ≀ unknown ∫::
  ≀ unknown ∫::
  ≀ natTerm i ∫::
  xs
```

The unknown value translates into using an underscore; i.e. it means we're asking Agda to infer the value in its place. This might seem suboptimal: we can probably figure out the values of those underscores, so shouldn't we try and find them, and supply them instead? In our experience, the answer is no.

Principle 4.1. Don't help the compiler! Supply the *minimal* amount of information possible in the AST to be unquoted, relying on inference as much as possible. The metalanguage is fragile and finicky with regards to scopes and context: the compiler isn't. You're more likely to get an argument wrong if you try and figure it out than the compiler is.

Using this, we can make a function for the AST which will generate a constant expression:

```
constExpr : ℕ → Term → Term
constExpr i x =
  quote K ⟨ con ⟩ ≀ i ⋯∫:: ⟨ x ⟩:: []
```

4.2 Matching on the Reflected Expression

There are three components we want to match on in the reflected expression: ring operators, variables, and constants.

4.2.1 Matching the Ring Operators

We'll do this via name equality. The three names we're interested in are as follows:

```
+′ *′ -′ : Name
+′ = quote AlmostCommutativeRing._+_
*′  = quote AlmostCommutativeRing._*_
-′  = quote AlmostCommutativeRing.-_
```

When we encounter the AST constructor which corresponds to a name reference def, we test for equality on each of these names successively. When (if) we get a match, we call one of the following functions, depending on the operator's arity:

```
getBinOp : ℕ
         → Name
         → List (Arg Term)
         → Term
getBinOp i nm (⟨ x ⟩:: ⟨ y ⟩:: []) =
  nm ⟨ con ⟩
    ≀ i ⋯∫::
    ⟨ toExpr i x ⟩::
    ⟨ toExpr i y ⟩::
    []
getBinOp i nm (x :: xs) = getBinOp i nm xs
getBinOp _ _ _ = unknown

getUnOp : ℕ
        → Name
        → List (Arg Term)
        → Term
getUnOp i nm (⟨ x ⟩:: []) =
  nm ⟨ con ⟩
    ≀ i ⋯∫::
    ⟨ toExpr i x ⟩::
    []
getUnOp i nm (x :: xs) = getUnOp i nm xs
getUnOp _ _ _ = unknown
```

These take a list of arguments, dropping any extra from the front, and package up the relevant ones with the relevant constructor from the AST. It may seem strange that there are "extra" arguments: surely these operators should have the same number of arguments as their arity?

Principle 4.2. Don't assume structure! Details like the order or number of implicit arguments to a function often can't be relied upon: be accommodating in your matching functions, only extracting components you really and truly need. In this case, for instance, the first argument will actually be the AlmostCommutativeRing record, as these functions are actually field accessors. But wait, no it won't—the first argument will actually be the hidden universe level of the carrier type, and the second (also hidden) will be the universe level of the equality relation. Only the third is the record, making the following arguments the "real" arguments to the function. Remember, none of this is typed, so if something changes in the order of arguments, you'll get type errors where you call solve, not where it's implemented.

4.2.2 Matching Variables

This task is actually reasonably simple: we check the de Bruijn index of the variable question, and if it's smaller than the number of variables in the ring expression, we simply leave it as is. We can do this because we're using the interface provided by Relation.Binary.Reflection as an intermediary: it will wrap up the variables in our Expr for us automatically. Which leads us to another observation:

Principle 4.3. Try and implement as much of the logic outside of reflection as possible. The expressive power granted by reflection comes with poor error messages, fragility, and a loss of first-class status. If something can be done without reflection, *do it*, and use reflection as the glue to get from one standard representation to another.

4.2.3 Matching Constants

This task seems the most daunting: with all of the logic we required before just to match expressions, how are we going to match the carrier type? Do we need the user to provide that function? Some kind of matching logic (as in [9]), which would need to recognize every constructor case, and build the corresponding Term?

Maybe we go another direction: we could search the subexpression to see if it contains any free variables, and decide based on that. $\mathcal{O}(n^2)$ time, anyone?

If the reflection API was different, this task could conceivably be made easier: for now, what most people seem to do is automate the process (as in [18]).

It is only by a very lucky coincidence that we can avoid all of this. Notice that all of the other cases are spoken for: in a correctly constructed expression, if no other clause matches, then what's left *has* to be a constant. So we just wrap it up in the constant constructor!

```
1  toExpr : (i : ℕ) → Term → Term
2  toExpr i t@(def f xs) with f ≟-Name +′
3  ... | yes p = getBinOp i (quote _⊕_) xs
4  ... | no _ with f ≟-Name *′
5  ... | yes p = getBinOp i (quote _⊗_) xs
6  ... | no _ with f ≟-Name -′
7  ... | yes p = getUnOp i (quote ⊖_) xs
8  ... | no _ = constExpr i t
9  toExpr i v@(var x args) with suc x ℕ.≤? i
10 ... | yes p = v
11 ... | no ¬p = constExpr i v
12 toExpr i t = constExpr i t
```

You'll notice that in the clauses where we fail to find a match (lines 8, 11, and 12), we assume that what we must have is a constant, so we just wrap it up as if it were one.

But what about incorrectly constructed expressions? What if the user makes a mistake in what they ask us to solve: surely we can't *assume* correctness? Well actually:

Principle 4.4. Ask for forgiveness, not permission. If the input to your macro requires the user to write an expression which conforms to a certain structure, *assume* that they have done so; don't check for it and proceed conditionally. With careful structuring of the macro's output, you can funnel the type error to exactly where the user was incorrect. For instance, in this case, if the user makes a mistake and the subexpression isn't a constant Carrier, the type error they'll get back will be something like "Expected type Carrier, found ...". In other words, exactly the type error we expect!

4.2.4 Building the Solution

The rest of the function is similar to above. Eventually, it will call Relation.Binary.Reflection with the constructed arguments. Because we've been careful not to supply any type information we don't need to, we actually get decent error messages when the solver fails. For instance, if we ask it to solve the following:

$$x + y * 1 + 3 \approx 2 + 1 + x + x$$

It will demonstrate where exactly it fails, with the error message:

$$(y + 0 * y) * 1 \neq x$$

This is because we pass it refl as the proof that the normal forms are equal: if they're not, this is where we'll get an error.

5 Setoid Applications

After constraining ourselves by only demanding a setoid from the user, we now get to reap some rather interesting benefits.

5.1 Isomorphisms

The first, and most obvious, "exotic" setoid is that over a universe of types. The relation is an *isomorphism* between these types. In this way, the solver can now automatically construct functions to convert between equivalent types. This has been explored before in a number of settings.

5.2 Didactic Solutions

One of the most widely-used and successful applications of computer algebra, especially among non-programmers, has been Wolfram Alpha [26]. Perhaps its most used feature is didactic (or pedagogic) solutions to maths problems [22]. For instance, given the input $x^2 + 5x + 6 = 0$, it will give the following output:

$$\begin{aligned} x^2 + 5x + 6 &= 0 \\ (x+2)(x+3) &= 0 \\ x + 2 = 0 \text{ or } x + 3 &= 0 \\ x = -2 \text{ or } x + 2 &= 0 \\ x = -2 \text{ or } x &= -3 \end{aligned}$$

These tools can be invaluable for students learning basic mathematics. Unfortunately, much of the software capable of generating usable solutions is proprietary, and little information is available as to their implementation techniques. [11] is perhaps the best current work on the topic, but even so very little exists in the way of the theoretical basis for pedagogical solutions.

Taking [11] as a jumping-off point, we can see that solutions are treated as *paths*: indeed, A* is the underlying algorithm used.

These paths have edges of rewrite rules—"commutativity of addition", etc. The vertices are equation states, and the start and end points of the path are the left-hand-side and right-hand-side of the equation, respectively. It's clear that we can form from this a relation with symmetry, transitivity and reflexivity: enough to satisfy our solver.

6 The Correct-By-Construction Approach

The Agda and Coq communities exhibit something of a cultural difference when it comes to proving things. Coq users seem to prefer writing simpler, almost non-dependent code and algorithms, to separately prove properties about that code in auxiliary lemmas. Agda users, on the other hand, seem to prefer baking the properties into the definition of the types themselves, and writing the functions in such a way that they prove those properties as they go (the "correct-by-construction" approach).

There are advantages and disadvantages to each approach. The Coq approach, for instance, allows you to reuse the same functions in different settings, verifying different properties about them depending on what's required. In Agda, this is more difficult: you usually need a new type for every invariant you maintain (lists, and then length-indexed lists, and then sorted lists, etc.). On the other hand, the proofs themselves often contain a lot of duplication of the logic in the implementation: in the Agda style, you avoid this duplication, by doing both at once. Also worth noting is that occasionally attempting to write a function that is correct by construction will lead to a much more elegant formulation of the original algorithm, or expose symmetries between the proof and implementation that would have been difficult to see otherwise.

[8], as an example, is very much in the Coq style: the definition of the polynomial type has no type indices, and makes no requirements on its internal structure:

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

The implementation presented here straddles both camps: we verify homomorphism in separate lemmas, but the type itself does carry information: it's indexed by the number of variables it contains, for instance, and it statically ensures it's always in canonical form.

Performing the same task in a correct-by-construction way is explored in [20] (in Idris [2]).

Here we provide a similar implementation, using the following definition:

```
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
```

While this approach reduced the amount of code we needed to write, we found it made optimizations more difficult, as it combined the execution and proof code together.

```
infixr 0 _⟸_
_⟸_ : ∀ {x y} → x ≋ y → Expr y → Expr x
x≋y ⟸ xs ⇐ xp = xs ⇐ x≋y ⟨ trans ⟩ xp

_⊞_ : ∀ {x y}
      → Expr x
      → Expr y
      → Expr (x + y)
(x ⇐ xp) ⊞ (y ⇐ yp) =
  xp ⟨ +-cong ⟩ yp ⟸ x ⊕ y
  where
  _⊕_ : ∀ {x y}
        → Poly x
        → Poly y
        → Expr (x + y)
  ⟦⟧ ⊕ ys = ys ⇐ +-identityˡ _
  ⟦ x ∷ xs ⟧ ⊕ ⟦⟧ = ⟦ x ∷ xs ⟧ ⇐ +-identityʳ _
  ⟦ x ∷ xs ⟧ ⊕ ⟦ y ∷ ys ⟧ with xs ⊕ ys
  ... | zs ⇐ zp = ⟦ x Coeff.+ y ∷ zs ⟧ ⇐
        (λ ρ → +-distrib ρ)
      ⟨ trans ⟩
        (refl ⟨ +-cong ⟩ (refl ⟨ *-cong ⟩ zp))
```

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. [Online]. 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] W. Jedynak, "A simple demonstration of the Agda Reflection API." Sep. 2018. [Online]. Available: https://github.com/wjzz/Agda-reflection-for-semiring-solver

[10] 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/

[11] 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

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

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

[14] 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

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

[16] 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

[17] B. Nordström, "Terminating general recursion," *BIT*, vol. 28, no. 3, pp. 605–619, Sep. 1987. [Online]. Available: http://link.springer.com/10.1007/BF01941137

[18] U. Norell, "Agda-prelude: Programming library for Agda," Aug. 2018. [Online]. Available: https://github.com/UlfNorell/agda-prelude

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

[20] 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

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

[22] The Development Team, "Step-by-Step Math," Dec. 2009. [Online]. Available: http://blog.wolframalpha.com/2009/12/01/step-by-step-math/

[23] 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

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

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

[26] Wolfram Research, Inc., "Wolfram|Alpha," Wolfram Research, Inc., 2019. [Online]. Available: https://www.wolframalpha.com/

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

mutual
  infixl 6 _Π_
  record Poly (n : ℕ) : Set (a ⊔ ℓ) where
    inductive
    constructor _Π_
    field
      {i} : ℕ
      flat : FlatPoly i
      i≤n : i ≤′ n

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

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

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

  Zero : ∀ {n} → Poly n → Set ℓ
  Zero (K x          Π _) = Zero-C x
  Zero (Σ []         Π _) = Lift _ ⊤
  Zero (Σ (_ ∷ _)    Π _) = Lift _ ⊥

  Norm : ∀ {i} → Coeffs i → Set
  Norm []                      = ⊥
  Norm (_ Δ zero   ∷ [])       = ⊥
  Norm (_ Δ zero   ∷ _ ∷ _)    = ⊤
  Norm (_ Δ suc _  ∷ _)        = ⊤
```

Figure 8: Final Definition of Sparse Polynomials

```
_⟦∷⟧_ : ∀ {n}
       → Poly n × Coeffs n
       → Carrier × Vec Carrier n
       → Carrier
(x , xs) ⟦∷⟧ (ρ , ρs) =
  ⟦ x ⟧ ρs + Σ⟦ xs ⟧ (ρ , ρs) * ρ

Σ⟦_⟧ : ∀ {n}
      → Coeffs n
      → Carrier × Vec Carrier n
      → Carrier
Σ⟦ [] ⟧ _ = 0#
Σ⟦ x ≠0 Δ i ∷ xs ⟧ (ρ , ρs) =
  (x , xs) ⟦∷⟧ (ρ , ρs) * ρ ^ i

⟦_⟧ : ∀ {n}
     → Poly n
     → Vec Carrier n
     → Carrier
⟦ K x  Π i≤n ⟧ _ = ⟦ x ⟧ᵣ
⟦ Σ xs Π i≤n ⟧ ρ  = Σ⟦ xs ⟧ (drop-1 i≤n ρ)
```

Figure 9: Semantics of Sparse Polynomials