An Efficient and Flexible Evidence-Providing Polynomial Solver in Agda

D Oisín Kidney

October 10, 2018

Abstract

We present a new implementation of a ring solver in the programming language Agda [19]. The new implementation is significantly more efficient than the version in the standard library [6], bringing it in line with Coq's `ring` tactic [22], [9].

We demonstrate techniques for constructing proofs based on the theory of lists, show how Agda's reflection system can be used to provide a safe and simple interface to the solver, and compare the "correct by construction" approach to that of auxiliary proofs.

We also show that, as a by-product of proving equivalences rather than equalities, the prover can be used to provide artifacts other than equational proofs, including step-by-step solutions, and isomorphisms.

Contents

```
record Monoid c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _•_
  infix 4 _≈_
  field
    Carrier   : Set c
    _≈_       : Rel Carrier ℓ
    _•_       : Op₂ Carrier
    ε         : Carrier
    isMonoid  : IsMonoid _≈_ _•_ ε
```

Figure 1: The definition of Monoid in the Agda Standard Library [6]

1 Introduction

Dependently typed programming languages allow programmers and mathematicians alike to write proofs which can be executed. For programmers, this often means being able to formally verify the properties of their programs; for mathematicians, it provides a system for writing machine-checked (rather than hand-checked) proofs.

Naïve usage of these systems can be tedious: the typechecker is often over-zealous in its rigor, demanding justification for every minute step in a proof, no matter how obvious or trivial it may seem to a human. For algebraic proofs, this kind of thing usually consists of long chains of rewrites, of the style "apply commutativity of +, then associativity of +, then at this position apply distributivity of $*$ over +" and so on, when really the programmer wants to say "rearrange the expression into this form, checking it's correct".

However, since our proof assistant is also a programming language, we can automate this process by writing a verified program to compute these proofs for us.

2 A Case Study in Monoids

Before describing the ring solver, first we will explain the technique of writing a solver in Agda in the simpler setting of 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).

Monoids can be represented in Agda in a straightforward way, as a record (see Figure 1). Immediately it should be noted that we're no longer talking about a monoid over a set, but rather one over a setoid. In other words, rather than using propositional equality (indicated by the ≡ symbol), we will use a user-supplied equivalence relation (≈ in Figure 1) in our proofs.

2.1 Equivalence Proofs

Propositions are stated in type signatures in dependently typed languages. Figure 2 is an example of such a proposition. 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!

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

Figure 2: Example Identity

Unfortunately, to convince the compiler we need to specify every instance of associativity and identity, rewriting the left-hand-side repeatedly until it matches the right:

```
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 hand-written 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.

Because we're not using propositional equality, some familiar tools are unavailable, like Agda's rewrite mechanism, or function congruence (this is why we have to explicitly specify the congruence we're using on line 6). The purpose of this particular hair shirt is flexibility: users can still use the solver even if their type only satisfies the monoid laws modulo some equivalence relation (perhaps they are have an implementation of finite, mergeable sets as balanced trees, and want to treat two sets as equivalent if their elements are equal, even if their internal structures are not). Beyond flexibility, we get some other interesting applications, which are explored in section 9.

Despite the pleasant syntax, the proof is mechanical, and it's clear that similar proofs would become tedious with more variables or more complex algebras (like rings). To avoid the tedium, then, we automate the procedure.

2.2 Canonical Forms

Automation of equality proofs like the one above can be accomplished by first rewriting both sides of the equation into a canonical form. Not every algebra has a canonical form: monoids do, though, and it's the simple list.

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

We're going to treat this type like an AST for a simple "language of lists". This language supports two functions: the empty list, and concatenation.

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

The type itself parameterized by the number of variables it contains. Users can refer to variables by their index:

```
infix 9 η_
η_ : ∀ {i} → Fin i → List i
η x = x :: []
```

And we can interpret this language with values for each variable supplied in a vector:

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

Compare this language to the language of monoid expressions that Figure 2 uses: both have identity elements and a binary operator, and both refer to variables. Our language of lists, however, has one significant advantage: the monoid operations don't depend on the contents of the lists, only the structure. In other words, an expression in the language of lists

will reduce to a flat list even if it has elements which are abstract variables. As a result, the identity from Figure 2 is *definitionally* true when written in the language of lists:

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

2.3 Extracting Evidence

While this is beginning to look like a solver, it's still not entirely clear how we're going to join up the pieces. The first step is to get a concrete representation of expressions which we can manipulate and pattern-match on (Figure 3). It has constructors for

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

Figure 3: The AST for monoid expressions

each of the monoid operations (⊕ and e are • and ε, respectively), and it's indexed by the number of variables it contains, which are constructed with ν.

We can convert from this AST to an unnormalized expression like so[1]:

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

Because this function performs no normalization or transformation, the output is definitionally equal to its equivalent expression. This means that we can discharge the proof obligation with refl:

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

The AST might look ugly, but it gives us a link between the expressions we had in Figure 2 and a concrete representation. The next link is between the AST and the normalized expression. We can convert from the AST to a normal form like so:

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

And since we already defined how to "evaluate" the list normal form, we can combine both steps into one:

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

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 4. 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 the next section.

[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 [6] for an implementation.

equivalent diagram for correct by construction (section 10)

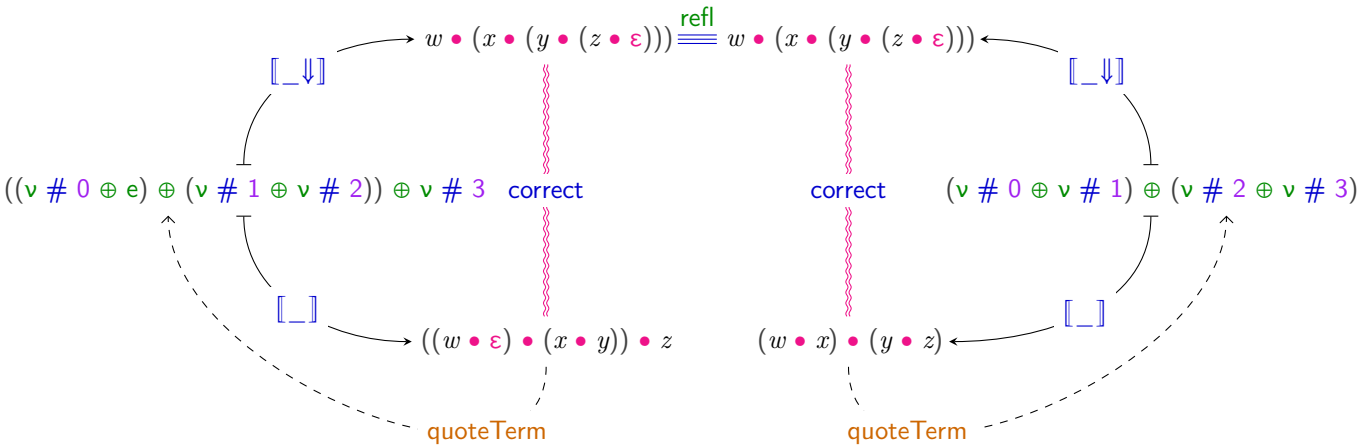


Figure 4: The Reflexive Proof Process

2.4 Homomorphism

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

$$\llbracket x \oplus y \rrbracket \rho \approx \llbracket x \oplus y \Downarrow \rrbracket \rho \tag{1}$$

$$\llbracket \mathsf{e} \rrbracket \rho \approx \llbracket \mathsf{e} \Downarrow \rrbracket \rho \tag{2}$$

$$\llbracket \nu i \rrbracket \rho \approx \llbracket \nu i \Downarrow \rrbracket \rho \tag{3}$$

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
  ■
```

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

```
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 2:

```
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
```

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. We now turn our focus to polynomials.

Prior work in this area includes [21], [15], [24], [5], and [20], but perhaps the state-of-the-art (at least in terms of efficiency) is Coq's `ring` tactic [22], which is based on an implementation described in [9].

3.1 Choice of Algebra

The choice of algebra has been glossed over thus far, but it is an important design decision: choose one with too many laws, and the solver becomes unusable for several types; too few, and we may miss out on normalization opportunities.

The algebra defined in [9] is that of an *almost-ring*. It's very close in definition too a ring, but it discards the requirement that negation is an inverse ($x + (-x) = 0$). Instead, it merely requires that negation distribute over addition and multiplication appropriately. This allows the solver to be used with non-negative types, like $\mathbb{N}$, where negation is simply the identity function.

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 long as $1 - 1$ reduces to 0 in the coefficient set, the solver will verify the identity.

4 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 Dense {ℓ} (coeff : RawRing ℓ) where
  open RawRing coeff
```

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 on the polynomial itself are defined like so[3]:

```
Poly : Set ℓ
Poly = List Carrier

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

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

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
```

4.1 Sparse Horner Normal Form

As it stands, the above representation has two problems:

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

1. Operators preceded with "ℕ." are defined over ℕ; e.g. ℕ.+, ℕ.∗.
2. Plain operators, like + and ∗, are defined over the coefficients.
3. Boxed operators, like ⊞ and ⊠, are defined over polynomials.
4. Operators which are boxed on one side are defined over polynomials on the corresponding side, and the coefficient on the other; e.g. ⋉, ⋊.

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 [9], 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[4].

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 4.1. 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*.

[4] In [9], 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 [9].

4.1.1 Uniqueness

While this form solves our efficiency problem, we still have redundant representations of the same polynomials. In [9], 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.

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:

```
module Sparse
  {a ℓ}
  (coeffs : RawRing a)
  (Zero : Pred (RawRing.Carrier coeffs) ℓ)
  (zero? : Decidable Zero)
  where
  open RawRing coeffs
```

Finally, we can define a sparse encoding of Horner Normal Form:

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

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

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 , j) :: xs) △ i = (x , j ℕ.+ i) :: xs
```

```
infixr 5 _::↓_
_::↓_ : Carrier × ℕ → Poly → Poly
(x , i) ::↓ xs with zero? x
... | yes p = xs △ suc i
... | no ¬p = (_≠0 x {¬p} , i) :: xs
```

4.1.2 Comparison

Our addition and multiplication functions will need to properly deal with the new sparse formulation. First things first, we'll need a way to match the power indices. We can use a function from [13] to do so.

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

This is a classic example of a "leftist" function: after pattern matching on one of the constructors of Ordering, it gives you information on type variables to the *left* of the pattern. In other words, when you run the function on some variables, the result of the function will give you information on its arguments.

4.1.3 Efficiency

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. Combined with judicious use of erase, we get the following:

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

compare : (n m : ℕ) → Ordering n m
compare n m with n < m | inspect (_<_ n) m
... | true  | [ n<m ]
  rewrite erase (less-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)
```

4.1.4 Termination

Unfortunately, using compare in the most obvious way won't pass the termination checker.

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

Agda needs to be able to see that one of the numbers returned by compare always reduces in size: however, since the difference is immediately packed up in a list in the recursive call, it's buried too deeply in constructors for the termination checker to see it.

Principle 4.1. To make termination obvious, perform call pattern specialization. The solution is twofold: unpack any constructors into function arguments as soon as possible, and eliminate any redundant pattern matches in the offending functions. Taken together, these form an transformation known as "call pattern specialization" [11][5]. Happily, this transformation both makes termination more obvious *and* improves performance.

```
mutual
  infixl 6 _⊞_
  _⊞_ : Poly → Poly → Poly
  [] ⊞ ys = ys
  ((x , i) :: xs) ⊞ ys = ⊞-zip-r x i xs ys

  ⊞-zip-r : Coeff → ℕ → Poly → Poly → Poly
  ⊞-zip-r x i xs [] = (x , i) :: xs
  ⊞-zip-r x i xs ((y , j) :: ys) =
    ⊞-zip (compare i j) x xs y ys

  ⊞-zip : ∀ {p q}
        → Ordering p q
        → Coeff
        → Poly
        → Coeff
        → Poly
        → Poly
  ⊞-zip (less i k) x xs y ys =
    (x , i) :: ⊞-zip-r y k ys xs
  ⊞-zip (greater j k) x xs y ys =
    (y , j) :: ⊞-zip-r x k xs ys
  ⊞-zip (equal i) x xs y ys =
    (coeff x + coeff y , i) ::↓ (xs ⊞ ys)
```

Figure 5: Terminating Addition, Using Call-Pattern Specialization

Every helper function in the mutual block matches on exactly one argument, eliminating redundancy. This also simplifies some of the homomorphism proofs later on.

5 Binary

Before continuing with polynomials, we'll take a short detour to look at binary numbers. These have a number of uses in dependently typed programming: as

You can probably get the foldr version of these functions to pass the termination checker if you use a sized type. [1] [2]

[5] 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.

well as being a more efficient alternative to Peano numbers, their structure informs that of many data structures, such as binomial heaps, and as such they're used in proofs about those structures.

Similarly to polynomials, though, the naïve representation suffers from redundancy in the form of trailing zeroes. There are a number of ways to overcome this (see [14] and [7], for example); yet another is the repurposing of our sparse polynomial from above.

```
Bin : Set
Bin = List ℕ
```

We don't need to store any coefficients, because 1 is the only permitted coefficient. Effectively, all we store is the distance to another 1.

Addition (elided here for brevity) is linear in the number of bits, as expected, and multiplication takes full advantage of the sparse representation:

```
pow : ℕ → Bin → Bin
pow i [] = []
pow i (x :: xs) = (x ℕ.+ i) :: xs

infixl 7 _*_
_*_ : Bin → Bin → Bin
_*_ [] _ = []
_*_ (x :: xs) =
  pow x ∘ foldr (λ y ys → y :: xs + ys) []
```

6 Multivariate

Up until now our polynomial has been an expression in just one variable. For it to be truly useful, though, we'd like to be able to extend it to many: luckily there's a well-known isomorphism we can use to extend our earlier implementation. For a polynomial with two variables, we will represent it as a polynomial whose coefficients are themselves polynomials of one variable. For three, it'll be a polynomial whose coefficients are polynomials in two variables, and so on. Generally speaking, a multivariate polynomial is one where its coefficients are polynomials with one fewer variable [5].

Before jumping to implement this, though, we should notice an opportunity for optimization (also pointed out in [9]). Just like how the monomials had gaps of zeroes between non-zero terms, this type will have gaps of constant polynomials between non-constant polynomials. In other words, in a expression of n variables, information about the last variable will be hidden behind n layers of nesting. If those n layers don't contain any information (i.e. they're all constant), we want to skip all that nesting with an *injection* index (like the power index).

6.1 Sparse Nesting

It's immediately clear that removing the gaps from the nesting will be more difficult than it was for the exponents: the Poly type is *indexed* by the number of variables it contains, so any manipulation of the injection index will have to correspond carefully to the Poly's index.

Our first approach might mimic the structure of Ordering, with an indexed type:

```
data Poly : ℕ → Set (a ⊔ ℓ) where
  _Π_ : ∀ i {j}
      → FlatPoly j
      → Poly (suc (i ℕ.+ j))
```

Where FlatPoly is effectively the gappy type we had earlier. If you actually tried to use this type, though, you'd run into issues: 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 $i_1 + j_1$ and $i_2 + j_2$, a task it will likely find too difficult. How do we avoid this? "Don't touch the green slime!" [12]:

> When combining prescriptive and descriptive indices, ensure both are in constructor form. Exclude defined functions which yield difficult unification problems.

We'll have to take another route.

6.1.1 Inequalities

First, we'll define our polynomial like so:

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

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. Next, FlatPoly:

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

We're back to an indexed type here, so you may be concerned about similar unification problems like the ones we had above: not to worry, though, as these indices are in constructor form, which prove much easier to unify than functions.

Also new here is the Norm function. It serves the same purpose that Zero did in the monomial, ensuring that there are no constant polynomials which could be replaced by incrementing the injection index. Its definition is as follows:

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

The rest of types are similar to what they were before:

```
infixl 6 _Δ_
record CoeffExp (i : ℕ) : Set (a ⊔ ℓ) where
  inductive
  constructor _Δ_
  field
    coeff : Coeff i
    pow : ℕ

Coeffs : ℕ → Set (a ⊔ ℓ)
Coeffs n = List (CoeffExp n)

infixl 6 _≠0
record Coeff (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 ℓ ⊤
Zero (Σ (_ :: _) Π _) = Lift ℓ ⊥
```

Again, similarly to the sparse exponent encoding, we provide a smart constructor which ensures normalization.

6.1.2 Choosing an Inequality

Conspicuously missing above is a definition for ≤. This choice has important performance implications, and exploring the various design avenues turned out to be more difficult than anticipated. As a jumping-off point, we'll look at the three definitions of ≤ in the Agda standard library [6].

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: first, remember the functions defined on the sparse exponent encoding above. When dealing with two polynomials, they'll have to line them up, comparing the respective gaps to find where two coefficients coincide (see ⊞-zip in figure 5). We'll have to do

the same with this version of sparseness: however, since we're no longer storing the gaps, the comparison will be on the size of the nested polynomial. 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:

```
infix 4 _≤_
data _≤_ (m : ℕ) : ℕ → Set where
  m≤m : m ≤ m
  ≤-s  : ∀ {n}
       → (m≤n : m ≤ n)
       → m ≤ suc n
```

While this structure stores the inequality by induction on the gap.

As long as the comparison uses the inductive structure of the final definition of ≤ above, it will have the correct complexity, while providing the comparison result about the correct type variables. With a slightly different definition of Ordering, we get the following:

```
data Ordering : ℕ → ℕ → Set where
  less : ∀ {n m}
       → n ≤ m
       → Ordering n (suc m)
  greater : ∀ {n m}
          → m ≤ n
          → Ordering (suc n) m
  equal : ∀ {n}
        → Ordering n n

≤-compare : ∀ {i j n}
          → (i≤n : i ≤ n)
          → (j≤n : j ≤ n)
          → Ordering i j
≤-compare m≤m       m≤m       = equal
≤-compare m≤m       (≤-s m≤n) = greater m≤n
≤-compare (≤-s m≤n) m≤m       = less m≤n
≤-compare (≤-s i≤n) (≤-s j≤n) =
  ≤-compare i≤n j≤n
```

A few things to note here:

- The ≤-compare function is one of those reassuring ones which Agda can completely infer.
- This function looks somewhat similar to the normal definition of compare and as a result, the "matching" logic for degree and number of variables began too look similar.

6.1.3 Axiom K

With this approach, we can indeed write all of the functions we need, but we will run into trouble trying to prove homomorphism.

The issue revolves around the fact that $\leq$ is *irrelevant*. In other words, if you have two proofs of (say) $n \leq m$, then those two proofs must also be propositionally equal. When we're trying to prove that two polynomials evaluate to the same thing, we'd like to rely on this property.

Unfortunately, though, proving this without axiom K is difficult. However, we should notice that we didn't need to prove anything like this when dealing with the sparse exponent encoding: why do we have to now?

6.1.4 Indexed Ordering

The issue lies in our newer definition of Ordering: whereas the standard definition (on $\mathbb{N}$) provided information about the arguments to the comparison function, the new definition provides only information about their *indices*. When pattern matching on the result of the comparison before, we got all the equality information we needed about the $\leq$ proofs.

To solve the problem, we'll more closely mimic the old comparison, and provide a result on the $\leq$ proofs directly. We should first look for an equivalent to addition. This turns out to be transitivity:

```
infixl 6 _⋈_
_⋈_ : ∀ {x y z} → x ≤ y → y ≤ z → x ≤ z
xs ⋈ m≤m = xs
xs ⋈ (≤-s ys) = ≤-s (xs ⋈ ys)
```

With this defined, the Ordering type is obvious:

```
data Ordering {n : ℕ} : ∀ {i j}
                      → (i≤n : i ≤ n)
                      → (j≤n : j ≤ n)
                      → Set
                      where
  _<_ : ∀ {i j-1}
      → (i≤j-1 : i ≤ j-1)
      → (j≤n : suc j-1 ≤ n)
      → Ordering (≤-s i≤j-1 ⋈ j≤n) j≤n
  _>_ : ∀ {i-1 j}
      → (i≤n : suc i-1 ≤ n)
      → (j≤i-1 : j ≤ i-1)
      → Ordering i≤n (≤-s j≤i-1 ⋈ i≤n)
  eq : ∀ {i} → (i≤n : i ≤ n) → Ordering i≤n i≤n
```

6.2 Operations

After adding the injection index optimization, we use a normalizing version of Π, like we did for the power indices (Figure 7). Operations now perform two "match-up" operations: one for the injection indices and one for the power indices (Figure 8).

Maybe unify the two match-up functions?

6.2.1 Termination, Again

Unfortunately, by introducing gaps into the encoding of nested polynomials, we have obscured the fact that the recursive call into the nested polynomial is safe (i.e. it strictly decreases in size). As a result, our functions are no longer obviously terminating:

```
{-# TERMINATING #-}
mutual
  ⊟_ : ∀ {n} → Poly n → Poly n
  ⊟ (K x Π i≤n) = K (- x) Π i≤n
  ⊟ (Σ xs Π i≤n) =
    foldr ⊟-cons [] xs Π↓ i≤n

  ⊟-cons : ∀ {n}
    → CoeffExp n
    → Coeffs n
    → Coeffs n
  ⊟-cons (x ≠0 Δ i) xs =
    ⊟ x ^ i ::↓ xs
```

There are a number of ways to deal with the problem. First, we can remove the use of a higher-order function:

```
mutual
  ⊟_ : ∀ {n} → Poly n → Poly n
  ⊟ (K x Π i≤n) = K (- x) Π i≤n
  ⊟ (Σ xs Π i≤n) = ⊟-cons xs Π↓ i≤n

  ⊟-cons : ∀ {n}
```

```
        → Coeffs n
        → Coeffs n
⊟-cons [] = []
⊟-cons (x ≠0 △ i :: xs) =
  ⊟ x ^ i ::↓ ⊟-cons xs
```

Agda can track termination through mutual functions: what we've done here is exposed the fact that the argument passed to ⊟ was embedded in a constructor of the original type recursed on, meaning it's strictly smaller. Calling foldr breaks this link, and therefore doesn't pass the termination checker.

However, we would like to use higher-order functions, especially in the more complex functions like ⊞ and ⊠. There are a number of techniques to solve the problem, which we'll go through here.

Sized Types The Agda compiler recently included a notion of "size" for its types [1]. To demonstrate this, we'll first start with a standard definition of a rose tree:

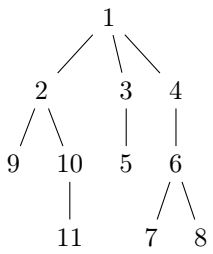
```
infixr 6 _&_
data Tree {a} (A : Set a) : Set a where
  _&_ : A → List (Tree A) → Tree A
```

Level-wise enumeration (Figure 6) is an example of a function with a non-obvious termination pattern:

```
{-# TERMINATING #-}
levels : ∀ {a} {A : Set a}
  → Tree A
  → List (List A)
levels {A = A} tr = f tr []
  where
  f : Tree A → List (List A) → List (List A)
  f (x & xs) []       = (x :: []) :: foldr f [] xs
  f (x & xs) (q :: qs) = (x :: q) :: foldr f qs xs
```

Again, the problem could be solved by avoiding the higher-order foldr:

```
levels : ∀ {a} {A : Set a}
  → Tree A
  → List (List A)
levels {A = A} (t & tr) = (t :: []) :: f tr []
  where
```



$$[[1],[2,3,4],[9,10,5,6],[11,7,8]]$$

Figure 6: Level-Order Traversal of a Tree

```
  f : List (Tree A)
    → List (List A)
    → List (List A)
  f []           qs       = qs
  f (x & xs :: xxs) []       = (x :: []) :: f xs []
  f (x & xs :: xxs) (q :: qs) = (x :: q) :: f xs qs
```

But now we have a far inferior function, and we can't reuse code about lists.

A solution is to add a *size* parameter to the rose tree:

```
infixr 6 _&_
data Tree {a} (A : Set a)
      : {i : Size} → Set a where
  _&_ : ∀ {i}
        → A
        → List (Tree A {i})
        → Tree A {↑ i}
```

The size parameter functions like ℕ might for termination: if we track it in the recursive calls, it's evidence for decreasing size.

```
levels : ∀ {a} {A : Set a}
       → Tree A
       → List (List A)
levels {A = A} tr = f tr []
  where
  f : ∀ {i}
    → Tree A {i}
    → List (List A)
```

```
  → List (List A)
f (x & xs) []       = (x :: []) :: foldr f [] xs
f (x & xs) (q :: qs) = (x :: q) :: foldr f qs xs
```

You'll notice that the top-level function doesn't specify the size it takes: it defaults to ∞, only being used where you need it for termination.

Unfortunately for us, this approach won't work: remember that we normalize the nesting of polynomials (Figure 7). This means we may remove one layer of nesting on certain polynomials, which would mean a change in the size parameter itself. This means we can't use the nice signature that the level-wise traversal has above: we'd need something a little more involved. What we *do* know is that the internal poly always has a *smaller* size than the external, but here we must stop, recognizing that we're about to duplicate the logic we already encoded with the $\leq$ proofs. Luckily for us, there *is* a way to use these to give us what we want.

Well-Founded Recursion 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 ℓ}
         {A : Set a}
         (_<_ : Rel A ℓ)
         (x : A) : Set (a ⊔ ℓ) 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!

```
open import Induction.Nat
open import Induction.WellFounded

⌊_⌋ : ℕ → Set
⌊_⌋ = Acc ℕ._<′_

⌊↓⌋ : ∀ {n} → ⌊ n ⌋
⌊↓⌋ {n} = <′-wellFounded n

mutual
  ⊟-step : ∀ {n} → ⌊ n ⌋ → Poly n → Poly n
  ⊟-step _ (K x Π i≤n) = K (- x) Π i≤n
  ⊟-step (acc wf) (Σ xs Π i≤n) =
    foldr (⊟-cons (wf _ i≤n)) [] xs Π↓ i≤n

  ⊟-cons : ∀ {n}
         → ⌊ n ⌋
         → CoeffExp n
         → Coeffs n
         → Coeffs n
  ⊟-cons ac (x ≠0 Δ i) xs =
    ⊟-step ac x ^ i ::↓ xs

⊟_ : ∀ {n} → Poly n → Poly n
⊟_ = ⊟-step ⌊↓⌋
```

6.3 Semantics

When it comes to "interpreting" the polynomials, we don't need to use the same ring as we did for the coefficients: all we need is a homomorphism between the coefficients and the target. This allows the user to

```
_Π↑_ : ∀ {n m} → Poly n → (suc n ≤ m) → Poly m
(xs Π i≤n) Π↑ n≤m = xs Π (≤-s i≤n ⋈ n≤m)

infixr 4 _Π↓_
_Π↓_ : ∀ {i n} → Coeffs i → suc i ≤ n → Poly n
[]                    Π↓ i≤n = K 0#                    Π  z≤n
(x ≠0 Δ zero :: [])   Π↓ i≤n = x                       Π↑ i≤n
(x₁ Δ zero   :: x₂ :: xs) Π↓ i≤n = Σ (x₁ Δ zero :: x₂ :: xs) Π  i≤n
(x  Δ suc j  :: xs)   Π↓ i≤n = Σ (x Δ suc j :: xs)   Π  i≤n
```

Figure 7: Normalizing Nesting

supply (potentially) a more efficient implementation for the coefficients, and then use it to prove things about some slower target type. The declaration for the semantics module, then, looks like this:

```
module Semantics
  {r₁ r₂ r₃ r₄}
  (coeffs : RawRing r₁)
  (Zero : Pred (RawRing.Carrier coeffs) r₂)
  (zero? : Decidable Zero)
  (ring : AlmostCommutativeRing r₃ r₄)
  (morphism :
    coeffs -Raw-AlmostCommutative⟶ ring)
  where

  open AlmostCommutativeRing ring
  open SparseNesting coeffs Zero zero?
  open _-Raw-AlmostCommutative⟶_
    morphism
    using ()
    renaming (⟦_⟧ to ⟦_⟧ᵣ)
```

Finally, for the interpretation functions, we see again evidence that our choice of inequality was the right one: it naturally works with the vector input, dropping elements in linear time.

```
  infixr 8 _^_
  _^_ : Carrier → ℕ → Carrier
  x ^ zero = 1#
  x ^ suc n = x * x ^ n

  drop : ∀ {i n}
       → i ≤ n
       → Vec Carrier n
       → Vec Carrier i
  drop m≤m P = P
  drop (≤-s si≤n) (_ :: P) = drop si≤n P

  vec-uncons : ∀ {n}
             → Vec Carrier (suc n)
             → Carrier × Vec Carrier n
  vec-uncons (x :: xs) = x , xs

  drop-1 : ∀ {i n}
         → suc i ≤ n
         → Vec Carrier n
         → Carrier × Vec Carrier i
  drop-1 si≤n xs = vec-uncons (drop si≤n xs)

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

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

7 Writing The Proofs

The proofs are long (roughly 1000 lines), albeit mechanical. There are some techniques that made the size manageable.

7.1 Equational Reasoning Techniques

7.1.1 Operators

Because congruence has to be explicitly stated in the reasoning tools, quite often you'll find yourself writing long chains of +-cong or *-cong just to focus into one expression buried in a larger one. To that end, we can use the following operators to make the "focusing" code shooter and easier to read:

```
infixr 1 ≪+_ +≫_ ≪*_ *≫_
≪+_ : ∀ {x₁ x₂ y}
      → x₁ ≈ x₂ → x₁ + y ≈ x₂ + y
≪+ prf = +-cong prf refl

+≫_ : ∀ {x y₁ y₂}
      → y₁ ≈ y₂ → x + y₁ ≈ x + y₂
+≫_ = +-cong refl

≪*_ : ∀ {x₁ x₂ y}
      → x₁ ≈ x₂ → x₁ * y ≈ x₂ * y
≪* prf = *-cong prf refl

*≫_ : ∀ {x y₁ y₂}
      → y₁ ≈ y₂ → x * y₁ ≈ x * y₂
*≫_ = *-cong refl
```

Their fixty means that they can be chained without parentheses. Something like: "≪+ *≫ *≫ ≪*" means "go to the left of +, then to the right of ∗, right again, then to the left of ∗". In other words, it can take a proof of $x \approx y$, and put it in a larger context:

```
example
  : ∀ {a b c d x y}
  → x ≈ y
  → (a * (b * (x * c))) + d ≈
          (a * (b * (y * c))) + d
example prf = ≪+ *≫ *≫ ≪* prf
```

7.1.2 Examples

Expand on the proofs. Maybe include an example proof?

7.2 The Algebra of Programming and List Homomorphisms

The algebra of programming [3] has been adapted to the dependently typed setting [16]. Along with Horner's rule [8], we use it here in the construction of proofs. First, a relational fold:

This actually works, but needs to be expanded upon.

```
foldR : ∀ {a b r}
        {A : Set a}
        {B : Set b}
      → (_R_ : B → List A → Set r)
      → {f : A → B → B}
      → {b : B}
      → (∀ y {ys zs}
          → ys R zs
          → f y ys R (y :: zs))
      → b R []
      → ∀ xs
      → foldr f b xs R xs
foldR _ f b [] = b
foldR P f b (x :: xs) = f x (foldR P f b xs)
```

8 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:

```
ident-infer : ∀ w x y z → _
ident-infer = solve 4
  ( λ w x y z
    → ((w ⊕ e) ⊕ (x ⊕ y)) ⊕ z
    ⊜ (w ⊕ x) ⊕ (y ⊕ z))
  refl
```

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: remember that one of the steps the solver does is curry the expressions, meaning that they all will take one argument. If a variable is referred to anywhere in the expression other than one of those arguments, it's index may be incorrect after currying. We'll see later how to deal with this. will we?

8.1 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 in Figure 3. 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 8.1. 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 ⟩:: []
```

8.2 Matching on the Reflected Expression

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

8.2.1 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 8.2. 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.

8.2.2 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 8.3. 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.

8.2.3 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 [10]), 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 8.4. 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!

8.2.4 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

Don't typecheck

9 Setoid Applications

I mentioned that the notion of equality we were using was more general than propositional, and that we could use it more flexibly in different contexts.

9.1 Traced

One "equivalence relation" is simply a labeled path: a list of rewrite rules or identities, repeatedly applied until the left-hand-side has been changed to the right. Print out the labels when done, and you have a step-by-step computer algebra system à la Wolfram Alpha. The definition of this type is straightforward:

```
infix 4 _≡⋯≡_
infixr 5 _≡⟨_⟩_
data _≡⋯≡_ : A → A → Set a where
  [refl] : ∀ {x} → x ≡⋯≡ x
  _≡⟨_⟩_ : ∀ {x} y {z}
           → String
           → y ≡⋯≡ z
           → x ≡⋯≡ z
  cong₁ : ∀ {x y z} {f : A → A}
          → String
          → x ≡⋯≡ y
          → f y ≡⋯≡ z
          → f x ≡⋯≡ z
  cong₂ : ∀ {x₁ x₂ y₁ y₂ z} {f : A → A → A}
          → String
          → x₁ ≡⋯≡ x₂
          → y₁ ≡⋯≡ y₂
          → f x₂ y₂ ≡⋯≡ z
          → f x₁ y₁ ≡⋯≡ z
```

And it does indeed implement the expected properties of an equivalence relation:

```
trans-≡⋯≡ : ∀ {x y z}
            → x ≡⋯≡ y
            → y ≡⋯≡ z
            → x ≡⋯≡ z
trans-≡⋯≡ [refl] ys = ys
trans-≡⋯≡ (y ≡⟨ x₁ ⟩ xs) ys =
  y ≡⟨ x₁ ⟩ (trans-≡⋯≡ xs ys)
trans-≡⋯≡ (cong₁ e x≡y fy≡z) ys =
  cong₁ e x≡y (trans-≡⋯≡ fy≡z ys)
trans-≡⋯≡ (cong₂ e x y fxy≡z) ys =
  cong₂ e x y (trans-≡⋯≡ fxy≡z ys)

cong : ∀ {x y}
       → (f : A → A)
       → x ≡⋯≡ y
       → f x ≡⋯≡ f y
cong f xs = cong₁ "cong" xs [refl]

sym-≡⋯≡ : ∀ {x y} → x ≡⋯≡ y → y ≡⋯≡ x
sym-≡⋯≡ {x} {y} = go [refl]
  where
    go : ∀ {z}
         → z ≡⋯≡ x
         → z ≡⋯≡ y
         → y ≡⋯≡ x
    go xs [refl] = xs
    go xs (y ≡⟨ y? ⟩ ys) =
      go (_ ≡⟨ y? ⟩ xs) ys
    go xs (cong₁ e ys zs) =
      go (cong₁ e ys (_ ≡⟨ e ⟩ xs)) zs
    go xs (cong₂ e xp yp zp) =
      go (cong₂ e xp yp (_ ≡⟨ e ⟩ xs)) zp
```

9.2 Isomorphisms

9.3 Counterexamples

Expand on the traced version, maybe clean it up? Also provide some examples.

10 The Correct-by-Construction Approach

The Agda and Coq communities exhibit something of a cultural difference when it comes to proving

Use the proof to translate between types. Check out Conor McBride's work on containers for this.

Possible to provide counterexamples if a

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.

[9], 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.

Expand on this section.

Performing the same task in a correct-by-construction way is explored in [21] (in Idris [4]). Here we provide a similar implementation:

```
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

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] A. Abel, "MiniAgda: Integrating sized and dependent types," in *PAR, Volume 43 of EPTCS*, 2010, pp. 14–28.

[2] G. Allais, "Three Tricks to make Termination Obvious," Nov. 2016. [On-

line]. Available: https://gallais.github.io/blog/termination-tricks.html

[3] R. Bird and O. de Moor, *Algebra of Programming*, ser. Prentice-Hall international series in computer science. London ; New York: Prentice Hall, 1997.

[4] 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

[5] 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

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

[7] M. Escardo, "Libraries for Bin," Jul. 2018. [Online]. Available: https://lists.chalmers.se/pipermail/agda/2018/010379.html

[8] J. Gibbons, "Horner's Rule," May 2011. [Online]. Available: https://patternsinfp.wordpress.com/2011/05/05/horners-rule/

[9] 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, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Hurd, and T. Melham, Eds., vol. 3603. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 98–113. [Online]. Available: http://link.springer.com/10.1007/11541868_7

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

[11] 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/

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

[13] C. McBride and J. McKinna, "The View from the Left," *J. Funct. Program.*, vol. 14, no. 1, pp. 69–111, Jan. 2004. [Online]. Available: http://strictlypositive.org/vfl.pdf

[14] S. Meshveliani, "Binary-4 – a Pure Binary Natural Number Arithmetic library for Agda," 21-Aug-2018. [Online]. Available: http://www.botik.ru/pub/local/Mechveliani/binNat/

[15] 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

[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] D. M. Russinoff, "Polynomial Terms and Sparse Horner Normal Form," Tech. Rep., Jul. 2017. [Online]. Available: http://www.russinoff.com/papers/shnf.pdf

[21] 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

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

[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] U. Zalakain, "Evidence-providing problem solvers in Agda," Submitted for the Degree of B.Sc. in Computer Science, University of Strathclyde, Strathclyde, 2017. [Online]. Available: https://umazalakain.info/static/report.pdf

A Extra Code Examples

```
mutual
  infixl 7 _⊠_
  _⊠_ : ∀ {n}
      → Poly n
      → Poly n
      → Poly n
  (xs Π i≤n) ⊠ (ys Π j≤n) =
    ⊠-match (i≤n cmp j≤n) xs ys

  ⊠-inj : ∀ {i k}
      → i ≤ k
      → FlatPoly i
      → Coeffs k
      → Coeffs k
  ⊠-inj _ _ [] = []
  ⊠-inj i≤k x (y Π j≤k ≠0 Δ p :: ys) =
    ⊠-match (i≤k cmp j≤k) x y ^ p ::↓
      ⊠-inj i≤k x ys

  ⊠-match : ∀ {i j n}
          → {i≤n : i ≤ n}
          → {j≤n : j ≤ n}
          → Ordering i≤n j≤n
          → FlatPoly i
          → FlatPoly j
          → Poly n
  ⊠-match (eq i&j≤n)    (K x)  (K y)  = K (x * y)          Π  i&j≤n
  ⊠-match (eq i&j≤n)    (Σ xs) (Σ ys) = ⊠-coeffs xs ys     Π↓ i&j≤n
  ⊠-match (i≤j-1 < j≤n) xs     (Σ ys) = ⊠-inj i≤j-1 xs ys Π↓ j≤n
  ⊠-match (i≤n > j≤i-1) (Σ xs) ys     = ⊠-inj j≤i-1 ys xs Π↓ i≤n

  ⊠-coeffs : ∀ {n}
           → Coeffs n
           → Coeffs n
           → Coeffs n
  ⊠-coeffs _ [] = []
  ⊠-coeffs xs (y ≠0 Δ j :: ys) = ⊠-step y ys xs △ j

  ⊠-step : ∀ {n}
         → Poly n
         → Coeffs n
         → Coeffs n
         → Coeffs n
  ⊠-step y ys [] = []
  ⊠-step y ys (x Π j≤n ≠0 Δ i :: xs) =
    (x Π j≤n) ⊠ y ^ i ::↓
      ⊞-coeffs (⊠-inj j≤n x ys) (⊠-step y ys xs)
```


Figure 8: Full Implementation of Multiplication on the Final Encoding of Poly