

An Efficient and Flexible Evidence-Providing Polynomial Solver in Agda

D Oisín Kidney

September 13, 2018

Abstract

We present a new implementation of a ring solver in the programming language Agda [11]. The efficiency is improved over the version included in the standard library [2] by including optimizations described in [4], among others.

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 generated proofs can be instantiated into a number of exotic settings, including:

1. Pretty-printing step-by-step solutions.
2. Providing human-readable counterexamples when a proof fails.
3. Constructing isomorphisms between types represented as polynomials.

Contents

1	Introduction	1
2	A Case Study in Monoids	2
2.1	Equivalence Proofs	2
2.2	Canonical Forms	3
2.3	Homomorphism	4
2.4	Usage	5
2.5	Reflection	5

3	A Polynomial Solver	5
4	Horner Normal Form	5
4.1	Sparse Horner Normal Form	6
4.1.1	Uniqueness	6
4.1.2	Comparison	7
4.1.3	Efficiency	7
4.1.4	Termination	8
5	Binary	9
6	Multivariate	9
6.1	Sparse Nesting	9
6.1.1	Inequalities	10
6.1.2	Choosing an Inequality	10
6.1.3	Indexed Ordering	11
7	Writing The Proofs	12
7.1	Equational Reasoning Techniques	12
7.2	The Algebra of Programming and List Homomorphisms	12
8	Setoid Applications	12
8.1	Traced	12
8.2	Isomorphisms	12
8.3	Counterexamples	12

9	The Correct-by-Construction Approach	13
----------	---	-----------

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 of machine-checked verification not available to handwritten 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”.

Luckily, since our proof assistant is also a programming language, we can provide the desired capability, by writing a function which converts expressions into a canonical form, and a proof that the conversion preserves the semantics of the expression. This can then be used to automatically construct equivalence proofs for equivalent expressions.

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 case of monoids.

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

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

Addition and multiplication (with 0 and 1 being the respective identity elements) are perhaps the most obvious candidates, as well as boolean conjunction and disjunction. In computer science, monoids have proved a useful abstraction for formalizing concurrency (associativity can be thought of as a kind of “order-independence”).

Monoids can be represented in Agda in a straightforward way, as a record (see Figure 1). Immediately

```
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 [2]

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 \equiv symbol), we will use a user-supplied equivalence relation (\approx 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.

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

Figure 2: Example Identity

To a human, the fact that the identity holds may well be obvious: \bullet is associative, so 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, 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 \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.

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 too 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 8.

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). Luckily, we can 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, which can be seen when we rewrite the identity in Figure 2 in it:

```

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

```

The equivalence of each side is obvious to Agda, and so we don't need to write it out as we did before.

Proving that the language of lists and the language of monoid expressions¹ are equivalent is the task of the next section.

¹For simplicity's sake, from now on when we talk about

2.3 Homomorphism

Our **obvious** proof has the form:

$$\text{lhs}_{\text{list}} = \text{rhs}_{\text{list}} \quad (1)$$

What we want, though, is the following:

$$\text{lhs}_{\text{mon}} = \text{rhs}_{\text{mon}} \quad (2)$$

Equation 1 can be used to build equation 2, if we supply two extra proofs:

$$\text{lhs}_{\text{mon}} \stackrel{a}{=} \text{lhs}_{\text{list}} = \text{rhs}_{\text{list}} \stackrel{b}{=} \text{rhs}_{\text{mon}} \quad (3)$$

The proofs labeled *a* and *b* are the homomorphism proofs, and it's what we tackle next.

First, we'll define a concrete AST for the monoid language:

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

It has constructors for each of the monoid operations (\oplus and **e** are \bullet and ϵ , respectively), and it's indexed by the number of variables it contains, which are constructed with **v**. Similarly to the list language, we can interpret it with a vector of values for the variables.

```
[_] : ∀ {i} → Expr i → Vec Carrier i → Carrier
[x ⊕ y] ρ = [x] ρ • [y] ρ
[e] ρ     = ε
[v i] ρ   = lookup i ρ
```

This interpreter will result in an unnormalized expression like the one in Figure 2; in other words, the outer sides of Equation 3. To get the normalized expressions, we'll first need to convert a monoid AST to a list AST:

the “language of monoid expressions”, we won't be referring to curried functions of *n* arguments, rather we'll deal with functions which take a vector of length *n*, that refer to each variable by position. Translating between those two forms is not directly relevant to what we're doing here: we refer the interested reader to the Relation.Binary.Reflection module of Agda's standard library [2].

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

Combining that with the fact that we know how to interpret the list AST, we get a normalizing interpreter for the monoid AST:

```
[_↓] : ∀ {i}
  → Expr i
  → Vec Carrier i
  → Carrier
[x ↓] ρ = norm x μ ρ
```

With this, we can see that the proofs needed by *a* and *b* are actually proofs of equivalence of these two interpreter functions.

Taking the non-normalizing interpreter as a template, the three cases are as follows²:

$$[x \oplus y] \rho \approx [x \oplus y \downarrow] \rho \quad (4)$$

$$[e] \rho \approx [e \downarrow] \rho \quad (5)$$

$$[v i] \rho \approx [v i \downarrow] \rho \quad (6)$$

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 (identity! _)
+-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)
```

²Equations 4 and 5 comprise a monoid homomorphism.

```

      → [ 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 (v x) ρ = identity' _

```

2.4 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 • e) • (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 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.

Fill in reflection section

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 [13], [9], [15], [1], and [12], but perhaps the state-of-the-art (at least in terms of efficiency) is Coq's `ring` tactic [14], which is based on an implementation described in [4].

That implementation has a number of optimizations which dramatically improve the complexity of evaluation, but it also includes a careful choice of algebra which allows for maximum reuse. 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 [4] is that of an *almost-ring*. This is a ring-like algebra, which 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. Also, because the implementation uses coefficients in the underlying ring, we lose no opportunities for normalization, as identities like $x + (-x) = 0$ will indeed compute.

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 so³:

```

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
[] ⊠ ys = []
(x :: xs) ⊠ [] = []
(x :: xs) ⊠ (y :: ys) =
  x * y :: (map (x * _) ys ⊕ (xs ⊠ (y :: ys)))

```

4.1 Sparse Horner Normal Form

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:

³Symbols chosen for operators use the following mnemonic:

1. Operators preceded with “N.” are defined over \mathbb{N} ; e.g. $\mathbb{N}+$, $\mathbb{N}*$.
2. Plain operators, like $+$ and $*$, are defined over the coefficients.
3. Boxed operators, like \boxplus and \boxtimes , 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. \ltimes , \rtimes .

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

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 [4], 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 index⁴.

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:

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

4.1.1 Uniqueness

While this form solves our efficiency problem, we still have redundant representations of the same polynomials. In [4], 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.

⁴In [4], 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 [4].

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 _Δ_
_Δ_ : Poly → ℕ → Poly
[] Δ i = []
((x , j) :: xs) Δ i = (x , j ℕ.+ i) :: xs

infixr 5 _::↓_
_::↓_ : Carrier × ℕ → Poly → Poly
(x , i) ::↓ xs with zero? x
... | yes p = xs Δ 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 gapless formulation. First things first, we'll need a way to match the power indices. We can use a function from [7] 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, we cannot yet define addition and multiplication. Using `compare` above 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.

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 optimization known as “call pattern specialization” [5]: it’s performed automatically in GHC, here we’re doing it manually. Perhaps a similar transformation could be automatically applied before termination checking in Agda’s compiler.

Until then, the structurally terminating function is defined like so:

```
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)

```

Ever helper function in the mutual block matches on exactly one argument, eliminating redundancy. Happily, this makes the function more efficient, as well as more obviously terminating.

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 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 [8] and [3], 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. A multivariate polynomial is one where its coefficients are polynomials with one fewer variable [1].

Before going any further, though, we should notice that this type is dense with regards to nesting the same way that the original monomial type was dense with regards to exponentiation. Every polynomial with n variables will be represented by n nested polynomials, regardless of how many of the variables in the expression are non-constant.

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 that number will have to carefully prove its correctness.

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. Pattern matching on a pair of `Polys` won't work, as Agda cannot (usually) unify user-defined functions. How do we avoid this? "Don't touch the green slime!" [6]:

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 gap is now implicit; instead, we store a proof that the nested polynomial has no more variables than the outer. Next, the rest of the types are similar to what they were before:

```
data FlatPoly : ℕ → Set (a ⊔ ℓ) where
  K : Carrier → FlatPoly 0
  Σ : ∀ {n}
    → (xs : Coeffs n)
    → {xn : Norm xs}
    → FlatPoly (suc n)

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
```

New here is the `Norm` function, in `FlatPoly`. Like `Zero` in `Coeff`, it proves that there really are no gaps (here in the nesting, rather than exponentiation, though). Its definition is as follows:

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

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

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 \leq .

Option 1: The Standard Way The most commonly used definition of \leq 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
```

For our purposes, though, this type is dangerous: it actually *increases* the complexity from the dense encoding. To understand why, remember the addition function above with the gapless exponent encoding. For it to work, we needed to

compare the gaps, and proceed based on that. We'll need to do a similar comparison on variable counts for this gapless encoding. However, we don't store the *gaps* now, we store the number of variables in the nested polynomial. 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 \leq 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 k s are equal. What about the m s? 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. That structure can be used to write a comparison function which was linear in the size of the gap (even though it compares the length of the tail).

6.1.3 Indexed Ordering

Now that the inequality is an inductive type, which mimics a Peano number stored in the gap, the parallels with the sparse exponent encoding should be even more clear. To write a comparison function, then, 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
```

Explain more the path from this to the final version. Intermediate comparison function and axiom K, especially

7 Writing The Proofs

The proofs are long (roughly 1000 lines), albeit mechanical.

7.1 Equational Reasoning Techniques

7.2 The Algebra of Programming and List Homomorphisms

8 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.

8.1 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
```

$$\begin{aligned} &\rightarrow f \ x_2 \ y_2 \equiv \dots \equiv z \\ &\rightarrow f \ x_1 \ y_1 \equiv \dots \equiv z \end{aligned}$$

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
```

8.2 Isomorphisms

8.3 Counterexamples

Expand on the proofs. Operators used, etc.

I haven't actually been able to apply the “algebra of programming” [10] stuff in the proofs themselves yet. This section may well be removed if it turns out I can't manage it, but for now it seems like a promising avenue that might get us some cleaner, more interesting proofs. Especially since so many of the functions are written as folds on lists.

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

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

9 The Correct-by-Construction Approach

Correct-by-construction is another approach [13].

```

infixr 0 [] <= _ [ _ :: ( _ ) ] <= _
data Poly (expr : Carrier) : Set (a ⊔ ℓ) where
  [] <= _ : expr ≍ 0#
           → Poly expr
  [ _ :: ( _ ) ] <= _
    : ∀ x xs
      → Poly xs
      → expr ≍ (λ ρ → x Coeff.+ ρ Coeff.* xs ρ)
      → Poly expr

```

References

- [1] 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
- [2] N. A. Danielsson, “The Agda standard library,” Jun. 2018. [Online]. Available: <https://agda.github.io/agda-stdlib/README.html>
- [3] M. Escardo, “Libraries for Bin,” Jul. 2018. [Online]. Available: <https://lists.chalmers.se/pipermail/agda/2018/010379.html>
- [4] 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
- [5] 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/>
- [6] C. McBride, “A Polynomial Testing Principle,” Jul. 2018. [Online]. Available: <https://twitter.com/pigworker/status/1013535783234473984>
- [7] 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>
- [8] 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/>
- [9] 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>
- [10] 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>
- [11] U. Norell and J. Chapman, “Dependently Typed Programming in Agda,” p. 41, 2008.
- [12] D. M. Russinoff, “Polynomial Terms and Sparse Horner Normal Form,” Tech. Rep., Jul. 2017. [Online]. Available: <http://www.russinoff.com/papers/shnf.pdf>

$$\begin{aligned}
& _ \boxplus _ : \forall \{x\ y\} \rightarrow \text{Poly } x \rightarrow \text{Poly } y \rightarrow \text{Poly } (x + y) \\
& ([_] \leftarrow xp) \boxplus ([_] \leftarrow yp) = [_] \leftarrow xp \langle +\text{-cong} \rangle yp \langle \text{trans} \rangle +\text{-identity}^l _ \\
& ([_] \leftarrow xp) \boxplus ([_] y :: ys \langle ys' \rangle) \leftarrow yp = [_] y :: ys \langle ys' \rangle \leftarrow xp \langle +\text{-cong} \rangle yp \langle \text{trans} \rangle +\text{-identity}^l _ \\
& ([_] x :: xs \langle xs' \rangle) \leftarrow xp \boxplus ([_] \leftarrow yp) = [_] x :: xs \langle xs' \rangle \leftarrow xp \langle +\text{-cong} \rangle yp \langle \text{trans} \rangle +\text{-identity}^r _ \\
& ([_] x :: xs \langle xs' \rangle) \leftarrow xp \boxplus ([_] y :: ys \langle ys' \rangle) \leftarrow yp = [_] x \text{Coeff.} + y :: xs + ys \langle xs' \boxplus ys' \rangle \leftarrow \\
& \quad xp \langle +\text{-cong} \rangle yp \langle \text{trans} \rangle \lambda \rho \rightarrow +\text{-distrib} _ _ _ _ \rho
\end{aligned}$$

- [13] 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
- [14] T. C. D. Team, “The Coq Proof Assistant, version 8.8.0,” Apr. 2018. [Online]. Available: <https://doi.org/10.5281/zenodo.1219885>
- [15] 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>