Automatically And Efficiently Illustrating Polynomial Equalities in Agda

Donnacha Oisín Kidney

January 20, 2019

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

Figure 1: The Solver

Abstract

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings and semirings in the programming language Agda [21]. It is asymptotically faster than Agda's existing solver. We use Agda's reflection machinery to provide a simple interface to the solver, and demonstrate a novel use of the constructed relations: step-by-step solutions.

Contents

```
proof : ∀ x y → x + y * 1 + 3 ≈ 2 + 1 + y + x
proof x y =
  begin
    x + y * 1 + 3
                  ≈⟨ refl ⟨ +-cong ⟩ *-identityʳ y ⟨ +-cong ⟩ refl {x = 3} ⟩
    x + y + 3
                  ≈⟨ +-comm x y ⟨ +-cong ⟩ refl ⟩
    y + x + 3
                  ≈⟨ +-comm (y + x) 3 ⟩
    3 + (y + x)
                  ≈⟨ sym (+-assoc 3 y x) ⟩
    2 + 1 + y + x
  ■
```

Figure 2: A Tedious Proof

1 Introduction

Why don't most mathematicians write their proofs in dependently-typed programming languages? Figure 2 suggests one reason: tedium! If even simple proofs take lines and lines of fussy arithmetic, how will the more complex ones fare?

So, we must automate them. This work presents a program to do exactly that, written in Agda (section 3). Automated solvers of this kind are important components of any dependently-typed language's standard library: ours is asymptotically more efficient than Agda's current implementation.

As well as being boring, though, writing these proofs is *difficult*. The programmer needs to remember the particular syntax and name for each step ("is it +-comm or +-commutative?"), and error messages can be unhelpful.

Our solver strives to be as easy to use as possible (section 4): the high-level interface is a single macro (solve, figure 1), we don't require anything of the user other than an implementation of one of the supported algebras, and effort is made to generate useful error messages.

Finally, when it comes to using computers to assist with mathematics, dependently-typed languages aren't the only game in town. Several computer algebra systems (CASs) have gained widespread popularity among mathematicians and students alike. While these systems don't have the correctness guarantees we get with Agda, they do have many other useful features. We take one of these features ("pedagogical", or step-by-step solutions), formalize it, and demonstrate how our solver can be used to implement it (section 5.2).

We want to make doing mathematics in Agda easier, friendlier, and more fun. This work contributes to that effort in the small (but important) area of polynomial equations.

1.1 Contributions

An New, Efficient Ring Solver (section 3) We provide an implementation of a polynomial solver in the programming language Agda. It improves upon the efficiency of the current solver in Agda's standard library [8] by using the optimizations described in [9]. Like the current solver, it works over a wide range of types and algebras, including rings, "almost"-rings (rings which don't necessarily support negation), and commutative semirings.

Techniques For Efficient Verification We demonstrate several techniques to make algorithms correct-by-construction *without* changing

their complexity class (section 3.5). These techniques can be used in any functional language with a type system powerful enough to express invariants.

Where we require extrinsic proofs, we demonstrate how the "Algebra of Programming" approach [18] can be used to cut down on boilerplate (section 3.6).

A Simple Interface (section 4) Using Agda's reflection machinery, we provide the simple interface shown in figure 1. It imposes minimal overhead on the user: only the Ring implementation is required, with no need for user implementations of quoting. Despite this, it is generic over any type which implements ring. To our knowledge, such an interface does not exist in Agda.

A Pedagogical Solver (section 5.2) We examine "pedagogical solutions", and present a formalism to describe them. We then show how our solver can be used to generate these solutions, and how to present them in a user-friendly way.

All of these contributions, while developed together, are entirely modular. For instance, both the reflection interface *and* the pedagogical solutions work with the current version of the ring solver.

2 The Reflexive Technique

Before describing the specifics of a given solver algorithm, it's important to understand how that algorithm can be applied in a dependently-typed language. How do we go from goal to proof? How do we describe the goal? How do we instantiate the proof?

We use a reflexive technique [2]. Rather than explaining it *and* the algorithm for solving rings all at once, we're first going to illustrate the technique with a simpler algebra: *monoids*.

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

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

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

In this section, we'll write a simple monoid solver. In the next, we will swap out monoids for commutative rings to get the full solver.

2.1 A "Trivial" Identity

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

Figure 3: A Simple Identity Of Monoids

As a running example for this section, we will use the identity in figure 3. To a human, the fact that the identity holds may well be obvious: • is associative, so we can scrub out all the parentheses, and ε is the identity element, so scrub it out too. After that, both sides are equal, so voilà!

Unfortunately, our compiler isn't nearly that clever. As alluded to before, we need to painstakingly specify every intermediate step, justifying every move (figure 4).

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

One trick worth pointing out is on line 6: the •-cong lifts two equalities to either side of a •. In

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

Figure 4: Proof of the identity in figure 3

other words, given proofs of the following:

$$x_1 \approx x_2$$
$$y_1 \approx y_2$$

it will prove:

$$x_1 \bullet y_1 \approx x_2 \bullet y_2$$

This function needs to be explicitly provided by the user, as we only require $\approx$ to be an equivalence relation (rather than, say, requiring that it be true propositional equality). Section 5 explains why this is useful.

2.2 ASTs for the Language of Monoids

The first hurdle for automatically constructing proofs comes from the fact that the identity in figure 3 is opaque: to the compiler, it just looks like a function with four arguments. This means we can't scrutinize or pattern-match on its contents. Our first step, then, is to define an AST (abstract syntax tree) for these expressions which we *can* pattern-match on:

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

This AST can express any expression which comprises of only monoid operations and variables.

$$\bullet \Longrightarrow \oplus$$
$$\varepsilon \Longrightarrow \mathsf{e}$$
$$x \Longrightarrow \nu \text{ (de Bruijn index of } x)$$

Variables are referred to by their de Bruijn indices (the type itself is indexed by the number of variables it contains). Here is how we would represent the left-hand-side of the identity in figure 3:

$$((0 \oplus \mathsf{e}) \oplus (1 \oplus 2)) \oplus 3$$

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

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

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

[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 [8] for an implementation.

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

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

2.3 Free Objects and Normal Forms

In both the monoid and ring solver, we will make use of the normal and canonical forms of expressions in each algebra. In the literature on CASs, the precise meaning of "normal" and "canonical" can vary from writer to writer. When used here, their definitions are as follows.

Definition 2.2 (Normal Forms)**.** The "normal form" of an expression is the *standard* way of representing that expression. For instance, we may say that our normal forms are fully expanded, with any free variables alphabetized where possible. As an example:

$$\begin{gathered}(21+y)(x+y)\\ \Downarrow \\ xy+21x+y^2+21y\end{gathered} \tag{1}$$

We often use ⇓ or ↓ to symbolize "normalization".

Definition 2.3 (Canonical Forms)**.** The canonical form of an expression is a representation of that expression such that any two *equivalent* expressions have the same representation.

A carefully chosen normal form may also be a canonical form, but it's often impractical or impossible to convert an expression to canonical form. Usually, we will instead define some normal form which is often (but not always) canonical.

The basic idea is to convert each side of the equation to their normal forms, and check if those forms are equal.

To convert to the normal form, we'll use a related concept: the free object.

Definition 2.4 (Free Objects)**.** For our purposes, a free object for some algebra is a data structure capable of representing expressions in that algebra. It's an AST, in other words. Crucially, it also must have the property that the laws or equations of the algebra hold *definitionally*. This implies that every free object is also a canonical form.

So now we can accomplish "normalization" by converting an expression to the free object, and then the free object back to an expression. Again, we won't always have a free object for the algebra we're interested in, so often we will settle for something close. Luckily, we do have such an object for monoids: the free monoid is more commonly known as the *list*.

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

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

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

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

```
_⊙_ : ∀ {i} → List i → Vec Carrier i → Carrier
xs ⊙ ρ = foldr (λ x xs → lookup x ρ • xs) ε xs
```

And finally (as promised) the opening identity (figure 3) is *definitionally* true when written in this language:

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

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

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

And then evaluate as before:

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

2.4 Homomorphism

Now we have a concrete way to link the normalized and non-normalized forms of the expressions. A diagram of the strategy for constructing our proof is in figure 5. The goal is to construct a proof of equivalence between the two expressions at the bottom: to do this, we first construct the ASTs which represent the two expressions (for now, we'll assume the user constructs this AST themselves. Later we'll see how to construct it automatically from the provided expressions). In figure 5, these ASTs are on the far left and right. 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.

Taking the non-normalizing interpreter as a template, the three cases correct will have to deal with are as follows[2]:

$$\llbracket\ x \oplus y\ \rrbracket\ \rho \approx \llbracket\ x \oplus y\ \Downarrow\rrbracket\ \rho \tag{2}$$

$$\llbracket\ \mathsf{e}\ \rrbracket\ \rho \approx \llbracket\ \mathsf{e}\ \Downarrow\rrbracket\ \rho \tag{3}$$

$$\llbracket\ \nu\ i\ \rrbracket\ \rho \approx \llbracket\ \nu\ i\ \Downarrow\rrbracket\ \rho \tag{4}$$

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

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

correct : ∀ {i}
        → (x : Expr i)
        → (ρ : Vec Carrier i)
        → ⟦ x ⇓⟧ ρ ≈ ⟦ x ⟧ ρ
correct (x ⊕ y) ρ =
  begin
    (norm x ++ norm y) μ ρ
  ≈⟨ ++-hom (norm x) (norm y) ρ ⟩
    norm x μ ρ • norm y μ ρ
  ≈⟨ correct x ρ ⟨ •-cong ⟩ correct y ρ ⟩
    ⟦ x ⟧ ρ • ⟦ y ⟧ ρ
  ■
correct e ρ = refl
correct (ν x) ρ = identityʳ _
```

2.5 Usage

Combining all of the components above, with some plumbing provided by the Relation.Binary.Reflection module, we can finally automate the solving of the original identity: figure 6.

2.5.1 Reflection

While the procedure is now automated, the interface isn't ideal: users have to write the identity they want to prove *and* the AST representing the identity. Removing this step is the job of reflection (section 4): in figure 5 it's represented by the path labeled quoteTerm.

[2] Equations 2 and 3 comprise a monoid homomorphism.

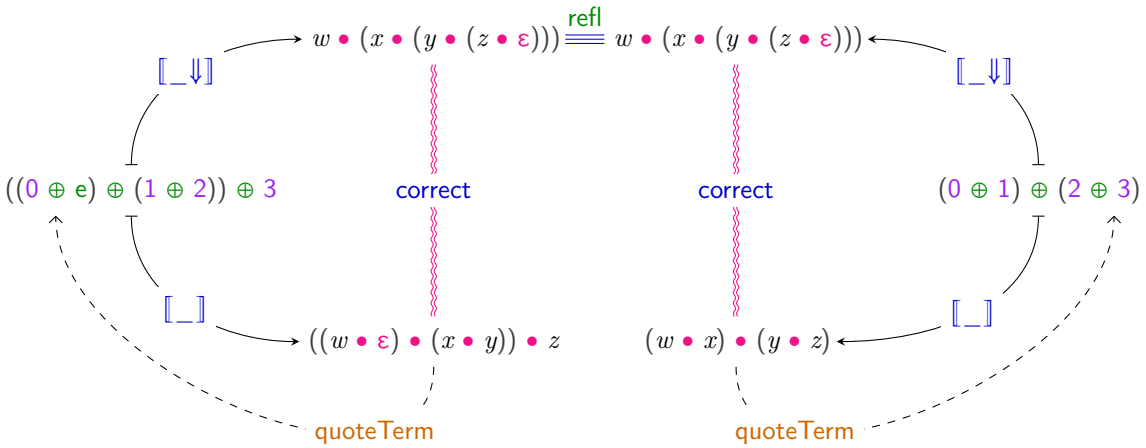

Figure 5: The Reflexive Proof Process

```
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
```

Figure 6: Automated Proof of the Identity in figure 3

3 A Polynomial Solver

With the monoid solver as a template, the components required for the ring solver are as follows: a normal form, a concrete representation of expressions, and a proof of correctness (homomorphism). Before addressing each of these, a small digression.

3.1 Choice of Algebra

So far, we've assumed the solver is defined over commutative rings. That wasn't the only algebra available to us when writing a solver, though: we've demonstrated techniques using monoids in the previous section, and indeed [22] uses *non*commutative rings as its algebra. Here, we will justify our[3]choice (and admit to a minor lie).

Because we want to solve arithmetic equations, we will need the basic operations of addition, multiplication, subtraction, and exponentiation (to a power in $\mathbb{N}$). This is only half of the story, though: along with those operations we will need to specify the laws or equations that they obey (commutativity, associativity, etc.). Here we must strike a balance: with every equation we specify, we gain more opportunities to solve new equations, but we narrow the range of types the solver will work with.

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

$$x - x = 0$$

to hold. This lets us lawfully define negation as the identity function for $\mathbb{N}$.

A potential worry is that because we don't require $x - x = 0$ axiomatically, it won't be provable in our

[3] "Our" choice here is the same choice as in [9].

system. This is not so: as is pointed out in [9],as long as $1 - 1$ reduces to 0 in the coefficient set, the solver will verify the identity.

3.2 Horner Normal Form

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

```
open import Algebra

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

The entire module is parameterized by the choice of coefficient. This coefficient should support the ring operations, but it is "raw", i.e. it doesn't prove the ring laws. The operations[4]on the polynomial itself are defined in figure 7.

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

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

3.3 Eliminating Redundancy

As it stands, the above representation has two problems:

Redundancy We allow trailing zeroes, so 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 redundancy means that we don't truly have a canonical form.

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.

```
Poly : Set c
Poly = List Carrier

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

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

Figure 7: Simple Operations on Dense Horner Normal Form

Since both leading and trailing zeroes present a problem, we will disallow zeroes altogether. Instead, like in [9], we will store a "power index" with every coefficient[5]. This index represents the "distance to the next non-zero coefficient".

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] 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.

[5] 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].

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

Now that we have chosen a normal form (polynomials without 0), can we prove that our implementation always maintains it? In [9], care is taken to ensure all operations include a normalizing step, but this is not verified: here, we do exactly that.

To check for zero, we require the user supply a decidable predicate on the coefficients. This changes the module declaration like so:

```
open import Algebra

module EliminatingRedundancy
  {c}
  (coeffs : RawRing c)
  (Zero? : RawRing.Carrier coeffs → Bool)
  where

open RawRing coeffs
```

Usually, Bool has no place in dependently-typed code: it doesn't come with evidence, and as such is less useful than a proof of decidability on, say, equivalence.

These properties make it the right choice for our use-case, though; especially when it comes to performance. Firstly, the fact that we're not demanding decidable equivalence means that we don't require the user to *refute* the predicate. In other words, we ask them to say (if the predicate returns true) that the thing is definitely zero; we don't, however, ask them to say it's definitely *not* zero if the predicate returns false. This opens the door to potentially more efficient implementations (and a wider array of types, such as those without decidable equality).

Secondly, the Bool means that we won't *use* the proof, except to say whether it exists or not. This allows us to avoid the expensive construction of a proof object.

And now we have a definition for sparse polynomials:

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

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

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

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

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

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

  is-zero? : ∀ x → Dec (T (Zero? x))
  is-zero? x with Zero? x
  ... | false = no (λ z → z)
  ... | true = yes tt

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

3.4 Multiple Variables

So far, the polynomials have been (suspiciously) single-variable. Luckily, there's a natural way to add multiple variables: nesting. For a polynomial with

one variable (x, say), the implementation is as before. For *two* variables (x and y), we will have an outer polynomial in y, whose *coefficients* are polynomials in x. Put inductively, a polynomial with 0 variables is simply a coefficient; a polynomial with n variables is a list of polynomials with $n-1$ variables. In types:

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

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

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

Zero : ∀ {n} → Poly n → Set
Zero {zero} x = T (Zero? x)
Zero {suc n} [] = ⊤
Zero {suc n} (x :: xs) = ⊥
```

3.5 Efficiency in Indexed Types

3.5.1 Call-Pattern Specialization

While both sparse encodings provide a more space-efficient representation, the computational efficiency has yet to be realized. Starting with the sparse monomial, we'll look at the addition function. In the dense encoding (figure 7), we needed to line up corresponding coefficients to add together. For this encoding, the "corresponding" coefficients are slightly harder to find. To do so, we'll need to compare the gap indices. This, however, presents our first problem:

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

It doesn't pass the termination checker! While it does indeed terminate, it isn't structurally decreasing in its arguments. To make it structurally decreasing, and therefore show the compiler it terminates, we'll use a well-known optimization for functional languages called "call-pattern specialization" [11].

Principle 3.1 (To make termination obvious, perform call-pattern specialization)**.** Unpack any constructors into function arguments as soon as possible, and eliminate any redundant pattern matches in the offending functions. Happily, this transformation both makes termination more obvious *and* improves performance.

GHC automatically performs this optimization: perhaps Agda's compiler could do something similar to reveal more terminating programs.

For our case, the principle applied can be seen in figure 8.

3.5.2 Built-In Functions

The second optimization we might rely on involves the call to compare. This is a classic "leftist" function: it returns an *indexed* data type (figure 9). The compare function itself is $\mathcal{O}(\min(n, m))$:

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

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

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

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

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

Figure 8: Termination by Call-Pattern Specialization

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

Figure 9: The Ordering Indexed Type

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

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

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

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

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

Combined with judicious use of erase and inspect, we

get the implementation in figure 10.

3.5.3 Unification

The way we added the capability for multiple variables to our polynomial type actually introduced an opportunity for another sparse encoding.

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

In contrast to the previous sparse encoding, though, the size of the gap is type-relevant. Because of this, the gap will have to be lifted into an index (figure 11).

It encodes "less than" in the same way that the ordering type did (figure 9), so it may seem (initially) like a perfect fit. However, we run into issues when it comes to performing the comparison-like operations above. Because it's an indexed type, pattern matching on it will force unification of the index with whatever type variable it was bound to. This is problematic because the index is defined by a function: pattern match on a pair of Polys and you're asking Agda to unify $i_1 + j_1$ and $i_2 + j_2$, a task it will likely find too difficult. How do we avoid this?

Principle 3.2 (Don't touch the green slime)**.** When combining prescriptive and descriptive indices, ensure both are in constructor form. Exclude defined functions which yield difficult unification problems [15].

We'll have to take another route.

3.5.4 Hanging Indices

First, we'll redefine our polynomial like so:

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

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

The choice of definition for this proof has important performance implications, as the proof will need to mesh with whatever comparison function we use for the injection indices. The Agda standard library [8] gives us 3 options.

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

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

Trying to proceed with this type will yield a nasty performance bug, though: the inductive structure of the type gives us no real information about the underlying "gap", so we're forced to compare the actual size of the nested polynomials. To see why this is a problem, consider the following sequence of nestings:

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

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.

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!

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

Figure 10: Fast comparison function using built-in functions on the natural numbers

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

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

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

This is the structure we will choose.

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

Principle 3.3 (To add more type information, to a type or function, keep the *structure* of the old type, while *hanging* new information off of it)**.** The three options above each present avenues to possible solutions to our "gap" problem, but they should have been ignored. Instead, we should have taken the previous untyped solution, and seen where in the inductive cases of the types used extra type information could have been stored. With this approach, the efficiency of the already-written algorithms is maintained. This practice can be somewhat automated using *ornaments* [7].

This is not yet enough to fully write our comparison function, though. Looking back to the previous definition of Ordering, we see that it contains +; we need an equivalent function on ≤. Remember that the quantity we'll be adding is adjacent gaps: this suggests the equivalent function is *transitivity*:

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

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

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

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

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

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

  Coeffs : ℕ → Set c
  Coeffs = List ∘ PowInd ∘ NonZero

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

Figure 11: A Sparse Multivariate Polynomial

```
      → (i≤n : suc i-1 ≤ n)
      → (j≤i-1 : j ≤ i-1)
      → ≤-Ordering i≤n
                   (≤-trans (≤-s j≤i-1) i≤n)
  ≤-eq : ∀ {i}
      → (i≤n : i ≤ n)
      → ≤-Ordering i≤n
                   i≤n

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

3.6 Abstraction and Folds for Simpler Proofs

At this point, following our many optimizations, the task of proving homomorphism for these implementations is more than a little daunting. However, we can ease the burden somewhat by leveraging the foldr function, in a similar way to [18].

The goal is to modularize the proofs by independently proving the correctness of each optimization. To do this, we will strive to write the arithmetic operations as higher-order functions, which operate over the "unoptimized" version of the polynomials (the dense versions), and have an intermediate function convert to and from the dense encoding. As a case study, we'll work with the negation function. Our initial definition (on the type defined in figure 16) is as follows:

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

```
  go : ∀ {n} → Coeffs n → Coeffs n
  go [] = []
  go (x ≠0 Δ i :: xs) = ⊟ x Δ i ::↓ go xs
```

Immediately we can recognize two functions which are good candidates for separated homomorphism proofs: ::↓ and Π↓. These functions will be used in every arithmetic operation, so it stands to reason that a separate proof of their correctness should save us some repetition. The lemmas are as follows:

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

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

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

.

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

Remembering that _⊠_ also used a fold, we should now construct a way to talk about "folds" abstractly, so that we can share some of the proof code.

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

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

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

In this way, we demonstrate a concrete and practical use of [18].

Expand!

3.7 Proving Higher-Order Termination With Well-Founded Recursion

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

To prove it, we'll use *well-founded recursion* [19]. It works by providing a relation which describes some strictly decreasing finite chain: $<$ on $\mathbb{N}$, for instance. It's strictly decreasing because the first argument always gets smaller (in contrast to, say, $\leq$), and it's finite because it must end at 0.

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

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

Agda's termination checker algorithm comes from foetus [1]: it's relatively simple, and relies on *structural termination*. Take the following function:

$$\mathsf{f}\ x = 1 + \mathsf{f}\ y$$

Agda will determine f to be terminating if and only if y is *structurally smaller* than x. In practice, this

means that y needs to be a subnode of x's recursive structure. For instance (ignoring partiality for now), all of the following are allowed:

```
f (suc y) = f y
f (_ :: y) = f y
```

Crucially, no matter how many arguments f has, if any one of them is structurally decreasing, it will pass the termination checker. This is where Acc comes in: as far as Agda is concerned, whatever's returned by the function stored in Acc is structurally smaller that the outer type it was wrapped in. So you can add it as a parameter to the dangerous functions, and call the wrapped function to generate the new Acc to pass to the recursive call.

Of course, to *call* the function you have to supply it with an argument: a proof that the relation holds. This part is why well-founded recursion is valid: whoever constructed the Acc originally will have had to pass the termination checker. In other words, they will have written a recursive function to generate nested Accs, where the only argument to that function is the relation. In this way, we have moved the obligation of proving termination from the call-site to the construction site. For the relation we use, the construction function for Acc is as follows:

```
mutual
  <′-wellFounded : ∀ n → Acc _<′_ n
  <′-wellFounded n = acc (go n)

  go : ∀ n m → m <′ n → Acc _<′_ m
  go zero    _ ()
  go (suc n) .n ≤′-refl = <′-wellFounded n
  go (suc n) m (≤′-step m<n) = go n m m<n
```

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. Usually the compiler can elide the calls, recognizing that the argument isn't used, but the optimization can't be guaranteed.

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

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

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

A small aside: why do we pattern match on acc in both cases of _⊟′_? Surely the first could be replaced by an underscore?

It could, but then you'd likely see a performance hit in compiled code. If we want Agda to elide the Acc argument altogether, it can help to treat it the same in every clause of a function, so it is especially obvious that it's not used for computation.

3.8 Laziness

Agda is a lazy language (like Haskell), so all of the usual caveats with regards to performance apply. Unfortunately, Agda's strictness primitives are cumbersome to write proofs about: seq x y is not definitionally equal to y, despite having the same semantics. We have to use another primitive which provides a proof of the equality.

As a result, strictness annotations aren't common in Agda code, and the usual practice is to structure functions carefully so that they force the correct arguments. For instance, the order of arguments to _*_ (in figure 15) was only chosen after careful experimentation: other orders were many thousands of times slower to typecheck.

3.9 Benchmarks

As expected, the sparse implementation exhibits a significant speedup in type-checking over the dense implementation[7]. Figure 12a shows time taken to type check a proof that $(x_1+x_2+x_3+x_4+x_5)^d$ is equal to its expanded form. The sparse representation is clearly faster overall, with a factor of 4 speedup for $d = 8$.

Figure 12b demonstrates where some of the speedup might come from. The expression x^d is represented very differently in the two implementations: in the sparse encoding, it's a single-element list containing a tuple of 1 and d; in the dense encoding, however, it's a list of length d, with a single 1 at the end. Since the complexities of arithmetic operations are bounded by the length of the list, this explains the difference in cost of the addition operations.

Figure 12c demonstrates perhaps a more common use-case, with a mix of high and low powers and some constants. The sparse representation's advantage is even more apparent here, with an 8.5-factor speedup at $d = 8$.

The dense solver does exhibit a small lead (roughly 2-3 seconds) on very simple expressions, possibly caused by the overhead of the sparse solver's more complex implementation. 3 seconds is quite small in the context of Agda type checking (the standard library, for instance, takes several minutes to type check), so we feel this slight loss is more than made up for by the several-minute gains. Furthermore, we have not been able to find a case where the dense solver is significantly faster than the sparse. Nonetheless, if a user really wants to use the dense solver, the components described in the following two sections are entirely modular, and can work with any underlying solver which uses the reflexive technique (including the dense solver).

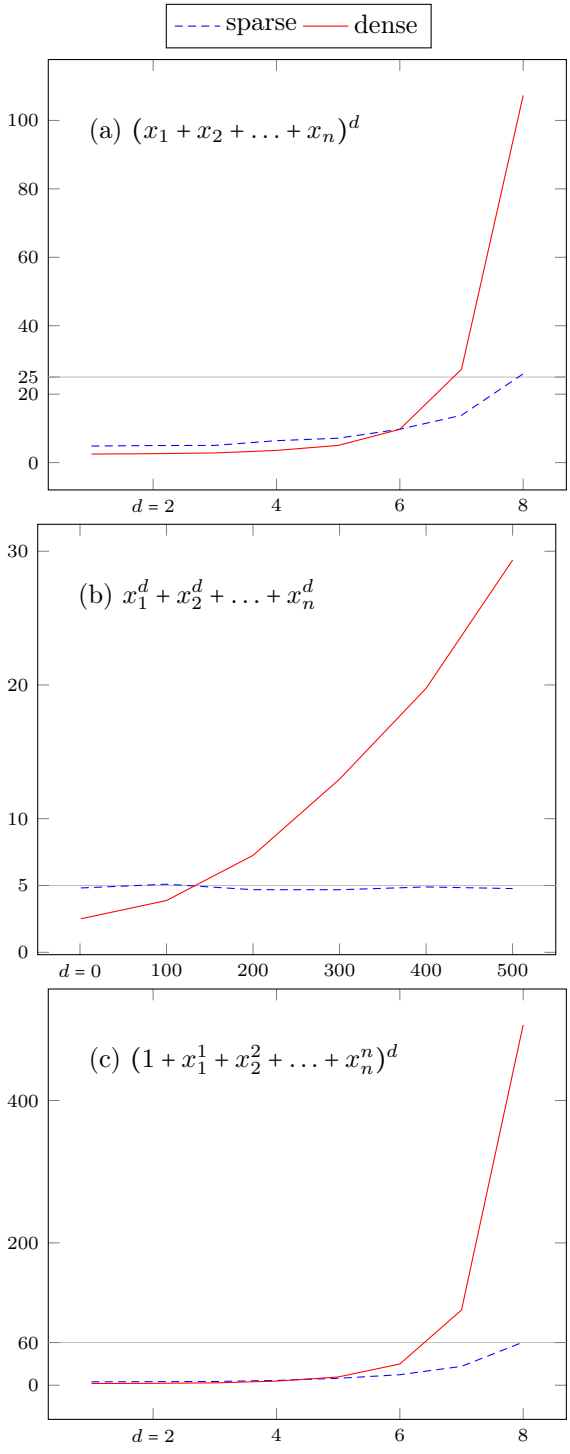

Figure 12: Time (in seconds) to prove each expression is equal to its expanded form ($n = 5$ for each).

[7]These benchmarks were performed on Agda version 2.6-0fa9b13, with the Agda standard library at commit-3bd3334a9552490e396f73f96812105a27e5917b, on a 2016 MacBook Pro, with a 2.9 GHz Intel Core i7 and 16 GB of RAM.

4 Reflection

One annoyance of the automated solver is that we have to write the expression we want to solve twice: once in the type signature, and again in the argument supplied to solve. See figure 6 for an example of this kind of duplication. We can cut down on the writing we need to do by half by omitting the type signature, but for a user, this is the *wrong half*: the syntax and structure of the AST is something of an implementation detail, and is not something we should require the user to learn. Instead, we provide a reflection-based interface, visible in figure 1.

The call to solve first infers the goal type, then constructs the corresponding AST to hand to the solver. Interestingly, the goal type need not be provided explicitly: if a user has a type mismatch between two expressions, just sticking in a call to solve in the appropriate place will often fix the problem. Furthermore, by not explicitly writing the AST in the program's source code, functions become more extensible and resilient to minor changes. As long as the two expressions match up semantically, rather than syntactically, the solver will figure it out.

We build this interface using Agda's reflection mechanisms [24]. Reflection in Agda allows a program to inspect and modify its own code. 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:

```
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. We need a function natTerm, for instance (figure 13),

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

Figure 13: A Function To "quote" ℕ

which simply gets the syntactic representation of a natural number. Unfortunately, we can't necessarily just call quoteTerm: the returned AST includes all kinds of information about context and environment which can clash with the environment where the macro is instantiated. A large amount of reflection and metaprogramming in Agda unfortunately consists of this kind of boilerplate (currently, at any rate).

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

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

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

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

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

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

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

4.1 Building The AST for Proving

Though Term is itself an AST we could theoretically manipulate and use in the prover, as is demonstrated above it's complex and unwieldy: what we really want is to use a smaller AST for ring expressions, like the one for lists.

```
data Expr {ℓ} (A : Set ℓ) (n : ℕ) : Set ℓ where
  K     : A → Expr A n
  I     : Fin n → Expr A n
  _⊕_ : Expr A n → Expr A n → Expr A n
  _⊗_ : Expr A n → Expr A n → Expr A n
  ⊖_    : Expr A n → Expr A n
```

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

Principle 4.1 (Don't help the compiler)**.** Supply the *minimal* amount of information possible in the AST to be unquoted, relying on inference as much as possible. The metalanguage is fragile and finicky with regards to scopes and context: the compiler isn't.

You're likely to get an argument wrong if you guess it; the compiler is more robust and better able to infer implicits than you are.

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

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

4.2 Matching on the Reflected Expression

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

4.2.1 Matching the Ring Operators

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

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

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

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

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

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

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

4.2.2 Matching Variables

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

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

4.2.3 Matching Constants

At first, this task seems the most daunting. Previously, we knew what we were looking for: one of the ring operators, or a variable with a certain de Bruijn index. A constant, though, could be almost anything!

Say the carrier type looks like ℕ: a bunch of nested constructors, in other words. One approach might be to match on those constructors (in Term), as is done in [10]. However, this places a huge burden on the user, requiring them to write complex and error-prone code to match on their type.

But it gets worse! There's no reason to believe that *every* constant in the expression will be a finite number of fully-applied constructors: what if the user refers to a variable? What if the ring type doesn't have any constructors at all (like, for instance, a church numeral)?

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 [20]).

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

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

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

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

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

4.3 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:

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

It will provide the rather helpful error message:

$$x \neq y \text{ of type } \mathbb{N}$$

Again, we're leveraging the typechecker itself rather than reimplementing our own checking logic. It's not perfect: it's followed by several lines of unreadable errors, and other expressions don't yield errors quite as nice. We could possibly improve the output in the future.

5 Setoid Applications

As mentioned previously, the solver does *not* require the user to prove the ring properties up to propositional equality. Instead, we ask them to prove them according to some arbitrary relation they have provided. All that we require of the *relation* is that it's an equivalence: transitivity, reflexivity, and symmetry.

The common use case for equivalence rather than equality is proofs "modulo normalization". Say, for instance, you have implemented a set using size-balanced binary trees. Under propositional equality, this implementation can't (efficiently) support the monoid operations: imbalances in the tree mean that two sets which contain the same elements may have different internal representations. However, equality following toList *does* follow the monoid laws. Because we didn't use propositional equality, the solver is able to work with this use case.

Equivalence relations have much more exotic instantiations than that, though. What follows are a few interesting examples.

5.1 Isomorphisms

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

5.2 Pedagogical Solutions

One of the most widely-used and successful computer algebra systems, especially among non-programmers, is Wolfram Alpha [26]. It can generate "pedagogical" (step-by-step) solutions to maths problems [23]. For instance, given the input $x^2 + 5x + 6 = 0$, it will give the following output:

Expand!

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

These tools can be invaluable for students learning basic mathematics. Unfortunately, much of the software capable of generating usable solutions is proprietary (including Wolfram Alpha), and little information is available as to their implementation techniques. [13] is perhaps the best current work on the

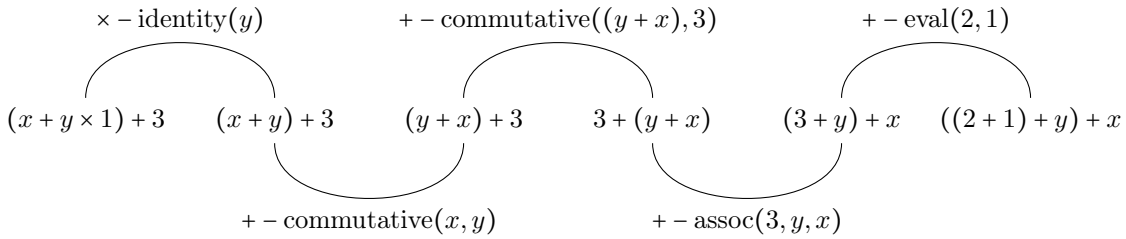

Figure 14: Rewriting Path

topic, but even so very little work exists in the way of the theoretical basis for pedagogical solutions.

[13] reformulates the problem as one of *pathfinding*. The left-hand-side and right-hand-side of the equation are vertices in a graph, where the edges are single steps to rewrite an expression to an equivalent form (figure 14). A* is used to search.

This approach suffers from a huge search space: every vertex will have an edge for almost every one of the ring axioms, and as such a good heuristic is essential. Unfortunately, what this should be is not clear: [13] uses a measure of the "simplicity" of an expression.

So, with an eye to using our solver instead of A*, we can notice that paths in undirected graphs form a perfectly reasonable equivalence relation: transitivity is the concatenation of paths, reflexivity is the empty path, and symmetry is *reversing* a path. Equivalence classes, in this analogy, are connected components of the graph.

More practically speaking, we implement these "paths" as lists, where the elements of the list are elementary ring axioms. When we want to display a step-by-step solution, we simply print out each element of the list in turn, interspersed with the states of the expression (the vertices in the graph).

```
module Trace {a b}
             {A : Set a}
             (_R_ : A → A → Set b)
             (sym′ : Symmetric _R_) where

  data _⋯_ (x : A) : A → Set (a ⊔ b) where
    [] : x ⋯ x
    _::_ : ∀ {y z}
           → x R y
           → y ⋯ z
           → x ⋯ z

  refl : Reflexive _⋯_
  refl = []

  trans : Transitive _⋯_
  trans [] ys = ys
  trans (x :: xs) ys = x :: trans xs ys

  sym : Symmetric _⋯_
  sym = go []
    where
    go : ∀ {x y z} → y ⋯ z → y ⋯ x → x ⋯ z
    go xs [] = xs
    go xs (y :: ys) = go (sym′ y :: xs) ys
```

If we stopped there, however, the solver would output incredibly verbose "solutions": far too verbose to be human-readable. Instead, we must apply a number of heuristics to cut down on the solution length:

1. First, we filter out "uninteresting" steps. These are steps which are obvious to a human, like associativity, or evaluation of closed terms. When a step is divided over two sides of an operator, it is deemed "interesting" if either side is interesting.

2. Next, we remove any "step, reverse-step" chains. Since we're converting expressions to normal form, the path may be Y-shaped, i.e., both sides of the equation will be rewritten into a common form, and then that

both paths may end in a long chain of the same proofs: i.e., the expressions may have been proved equivalent long before that. When we encounter a step which is the "reverse" of the step before or after it, we skip both, maintaining the state.

Given those, our solver outputs the following for the lemma in figure 1:

```
x + y * 1 + 3
    ={ *-ident y }
x + y + 3
    ={ +-comm y x }
y + x + 3
    ={ +-comm (y + x) 3 }
2 + 1 + y + x
```

Figuring out good heuristics and path compression techniques seems to deserve further examination.

6 The Correct-By-Construction Approach

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

There are advantages and disadvantages to each. 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 normal form.

Performing the same task in a correct-by-construction way is explored in [22] (in Idris [3]).

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

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

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

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

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

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

7 Related Work

In dependently-typed programming languages, the state-of-the-art solver for polynomial equalities (over commutative rings) was originally presented in [9], and is used in Coq's `ring` solver. This work improved on the already existing solver [6] in both efficiency and flexibility. In both the old and improved solvers, a reflexive technique (section 2) is used to automate the construction of the proof obligation (as described in [2]).

Agda [21] is a dependently-typed programming language based on Martin-Löf's Intuitionistic Type Theory [14]. Its standard library [8] currently contains a ring solver which is similar in flexibility to Coq's `ring`, but doesn't support the reflection-based interface, and is less efficient due to its use of a dense (rather than sparse) internal data structure.

In [22], an implementation of an automated solver for the dependently-typed language Idris [3] is described. The solver is implemented with a "correct-by-construction" approach, in contrast to [9]. The solver is defined over *non*commutative rings, meaning that it is more general (can work with more types) but less powerful (meaning it can prove fewer identities). It provides a reflection-based interface, but internally uses a dense representation.

Reflection and metaprogramming are relatively recent additions to Agda, but form an important part of the interfaces to automated proof procedures. Reflection in dependent types in general is explored in [5], and specific to Agda in [24].

Formalization of mathematics in general is an ongoing project. [25] tracks how much of "The 100 Greatest Theorems" [12] have so far been formalized (at time of writing, the number stands at 93). DoCon [17] is a notable Agda library in this regard: it contains many tools for basic maths, and implementations of several CAS algorithms. Its implementation is described in [16]. [4] describes the manipulation of polynomials in both Haskell and Agda.

Finally, the study of *pedagogical* CASs which provide step-by-step solutions is explored in [13]. One of the most well-known such system is Wolfram Alpha [26], which has step-by-step solutions [23].

References

[1] A. Abel, *Foetus – Termination Checker for Simple Functional Programs*, 1998.

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

[3] 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

[4] 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

[5] D. R. Christiansen, "Practical Reflection and Metaprogramming for Dependent Types," Ph.D. dissertation, IT University of Copenhagen, Nov. 2015. [Online]. Available: http://davidchristiansen.dk/david-christiansen-phd.pdf

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

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

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

[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, 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] N. W. Kahl, "The Hundred Greatest Theorems," 2004. [Online]. Available: http://web.archive.org/web/20080105074243/http://personal.stevens.edu/~nkahl/Top100Theorems.html

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

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

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

[16] 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

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

[18] 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://github.com/scmu/aopa

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

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

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

[22] 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

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

[24] 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

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

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

A Longer Code Examples

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

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

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

Figure 15: Semantics of Sparse Polynomials

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

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

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

  Coeffs : ℕ → Set a
  Coeffs = List ∘ PowInd ∘ NonZero

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

  Zero : ∀ {n} → Poly n → Set
  Zero (K x       Π _) = T (Zero? x)
  Zero (Σ []      Π _) = ⊤
  Zero (Σ (_ :: _) Π _) = ⊥

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

Figure 16: Final Definition of Sparse Polynomials