An Efficient and Flexible Evidence-Providing Polynomial Solver in Agda

D Oisín Kidney

September 12, 2018

Abstract

We provide an efficient implementation of a polynomial solver in the programming language Agda, and demonstrate its use in a variety of applications.

Contents

1 Introduction

Dependently typed languages such as Agda [5] and Coq [8] allow programmers to write machine-checked proofs as programs. They provide a degree of reassurance that handwritten proofs cannot, and allow for exploration of abstract concepts in a machine-assisted environment.

We will describe an efficient implementation of an automated prover for equalities in ring and ring-like structures, and show how it can be extended for use in settings more exotic than simple equality.

2 Monoids

Before describing the ring solver, first we will explain the simpler case of a monoid solver.

A monoid is a set equipped with a binary operation, $\bullet$, and a distinguished element ϵ, which obeys the laws:

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

2.1 Equality Proofs

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

These come equipped with their own equivalence relation, according to which proofs for each of the monoid laws are provided. Using this, we can prove identities like the one in figure 2.

```
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]

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

Figure 2: Example Identity

While it seems like an obvious identity, the proof is somewhat involved (figure 3).

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

Figure 3: Proof of identity in figure 2

The syntax mimics that of normal, handwritten proofs: the successive "states" of the expression are interspersed with equivalence proofs (in the brackets). Perhaps surprisingly, the syntax is not built-in: it's simply defined in the standard library.

Despite the powerful 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. This form depends on the particular algebra used in the pair of expressions. For instance, a suitable canonical form for monoids is lists.

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

This type can be thought of as an AST for the "language of lists". Crucially, it's equivalent to the "language of monoids": this is the language of expressions written using only variables and the monoid operations, like the expressions in figure 2. The neutral element and binary operator have their equivalents in lists: ϵ is simply the empty list, whereas $\bullet$ is list concatenation.

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

We can translate between the language of lists and monoid expressions [1] with μ and η.

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

infix 9 η_
```

[1] For simplicity's sake, instead of curried functions of n arguments, we'll deal with functions which take a vector of length n, that refer to each variable by position, using Fin, the type of finite sets. Of course these two representations are equivalent, but the translation 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].

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

We have one half of the equality so far: that of the canonical forms. As such, we have an "obvious" proof of the identity in figure 2, expressed in the list language (figure 4).

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

Figure 4: The identity in figure 2, expressed in the list language

2.3 Homomorphism

Figure 4 gives us a proof of the form:

$$\text{lhs}_{list} = \text{rhs}_{list} \tag{1}$$

What we want, though, is the following:

$$\text{lhs}_{mon} = \text{rhs}_{mon} \tag{2}$$

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

$$\text{lhs}_{mon} \overset{a}{=} \text{lhs}_{list} = \text{rhs}_{list} \overset{b}{=} \text{rhs}_{mon} \tag{3}$$

The proofs labeled a and b are the task of this section.

First, we'll define a concrete AST for the monoid language (figure 5). It has constructors for each of the monoid operations (⊕ and e are • and ϵ, respectively), and it's indexed by the number of variables it contains, which are constructed with ν. Converting back to an opaque function is accomplished in figure 6.

Finally, then, we must prove the equivalence of the monoid and list languages. This consists of the following proofs:

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

Figure 5: The AST for the Monoid Language

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

Figure 6: Evaluating the Monoid Language AST

$$(\eta x)\mu\rho = [\![\nu x]\!]\,\rho \tag{4}$$
$$(x \mathbin{+\!\!+} y)\mu\rho = [\![x \oplus y]\!]\,\rho \tag{5}$$
$$[\,]\mu\rho = [\![e]\!]\,\rho \tag{6}$$

The latter two proofs comprise a monoid homomorphism.

The proofs are constrained: we are only permitted to use the laws provided in the Monoid record, and the equivalence relation is kept abstract. The fact that we're not simply using propositional equality allows for some interesting applications (see section 6), but it also removes some familiar tools we may reach for in proofs. Congruence in particular must be specified explicitly: the combinator •-cong is provided for this purpose. With this understood, the proofs can be written:

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

⋕-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)
        → (conv x μ) ρ ≈ ⟦ x ⟧ ρ
correct (x ⊕ y) ρ =
  begin
    ((conv x ++ conv y) μ) ρ
  ≈⟨ ++-hom (conv x) (conv y) ρ ⟩
    (conv x μ) ρ • (conv y μ) ρ
  ≈⟨ •-cong (correct x ρ) (correct y ρ) ⟩
    ⟦ x ⟧ ρ • ⟦ y ⟧ ρ
  ■
correct e ρ = refl
correct (ν x) ρ = identityʳ _
```

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 • (((x • ε) • y) • z)
       ≈ (w • x) • (y • z)
ident′ = solve 4
  ( λ w x y z
    → w ⊕ (((x ⊕ e) ⊕ 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 ⊕ (((x ⊕ e) ⊕ 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 [7], [4], [9], [1], and [6], but perhaps the state-of-the-art (at least in terms of efficiency) is Coq's `ring` tactic [8], which is based on an implementation described in [3].

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 [3] 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 addiction 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 so:

```
open import Data.List as List
  using (List; _::_; []; map)

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

As it stands, the above representation has two problems:

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

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

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

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

In [3], the problem is addressed primarily from the efficiency perspective: they add a field for the "power index". For our case, we'll just store a list of pairs, where the second element of the pair is the power index[2].

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:

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

[2] In [3], 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 [3].

5 Multivariate

5.1 Sparse

5.2 K

6 Setoid Applications

6.1 Traced

6.2 Isomorphisms

6.3 Counterexamples

7 The Correct-by-Construction Approach

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] 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

[4] 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

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

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

[7] 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

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

[9] 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