An Efficient and Flexible Evidence-Providing Polynomial Solver for Polynomials 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 [2] and Coq [3] 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 [1]

```
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 instead 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 [1].

```
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 ($\oplus$ and e are $\bullet$ 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

$$(\eta x)\mu\rho = [\![\nu x]\!]\,\rho \tag{4}$$

$$(x ++ y)\mu\rho = [\![x \oplus y]\!]\,\rho \tag{5}$$

$$[]\mu\rho = [\![e]\!]\,\rho \tag{6}$$

The latter two proofs comprise a monoid homomorphism.

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.

Fill in reflection section

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

3 Horner Normal Form

3.1 Sparse

4 Multivariate

4.1 Sparse

4.2 K

5 Setoid Applications

5.1 Traced

5.2 Isomorphisms

5.3 Counterexamples

6 The Correct-by-Construction Approach

7 Reflection

References

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

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

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