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 : Set c where
  [] : List
  _::_ : Carrier → List → List
```

In the "language of monoids", the neutral element and binary operator have their equivalents in lists: ϵ is simply the empty list, whereas $\bullet$ is list concatenation.

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

Translating between the language of lists and the language of monoids is accomplished with μ and η.

```
_μ : List → Carrier
[] μ = ε
(x :: xs) μ = x • xs μ

η : Carrier → List
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). Showing that the list language is equivalent to the monoid language is the next task.

2.3 Homomorphism

We need to show that the equality proof above is a valid proof of the same identity in figure 2. To do

```
obvious
  : ∀ {w x y z}
  → η w ++ (((η x ++ []) ++ η y) ++ η z)
  ≡ (η w ++ η x) ++ (η y ++ η z)
obvious = ≡.refl
```

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

that, we will need to prove the following:

$$(\eta x)\mu = x \tag{1}$$
$$(x + y)\mu = x\mu \bullet y\mu \tag{2}$$
$$[\,]\mu = \epsilon \tag{3}$$

1 simply states that translation to and from doesn't change anything, while 2 and 3 give us a monoid homomorphism. Once done, we can use a separate AST type, something potentially like:

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

This AST can be evaluated in a straightforward way:

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

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