An Efficient and Flexible Evidence-Providing Polynomial Solver in Agda

Donnacha Oisín Kidney

December 9, 2018

Abstract

We present a new implementation of a ring solver in the programming language Agda [2]. The new implementation is significantly more efficient than the version in the standard library [1], bringing it in line with Coq's `ring` tactic [3], [**?**].

We demonstrate techniques for constructing proofs based on the theory of lists, show how Agda's reflection system can be used to provide a safe and simple interface to the solver, and compare the "correct by construction" approach to that of auxiliary proofs.

We also show that, as a by-product of proving equivalences rather than equalities, the prover can be used to provide artifacts other than equational proofs, including step-by-step solutions, and isomorphisms.

Contents

1 Introduction

Dependently typed programming languages allow programmers and mathematicians alike to write proofs which can be executed. For programmers, this often means being able to formally verify the properties of their programs; for mathematicians, it provides a system for writing machine-checked (rather than hand-checked) proofs.

Naïve usage of these systems can be tedious: the typechecker is often over-zealous in its rigor, demanding justification for every minute step in a proof, no matter how obvious or trivial it may seem to a human. For algebraic proofs, this kind of thing usually consists of long chains of rewrites, of the style "apply commutativity of +, then associativity of +, then at this position apply distributivity of $*$ over +" and so on, when really the programmer wants to say "rearrange the expression into this form, checking it's correct".

However, since our proof assistant is also a programming language, we can automate this process by writing a verified program to compute these proofs for us.

2 A Case Study in Monoids

Before describing the ring solver, first we will explain the technique of writing a solver in Agda in the simpler setting of 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:

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

```
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]

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).

Monoids can be represented in Agda in a straightforward way, as a record (see Figure 1). Immediately it should be noted that we're no longer talking about a monoid over a set, but rather one over a setoid. In other words, rather than using propositional equality (indicated by the ≡ symbol), we will use a user-supplied equivalence relation (≈ in Figure 1) in our proofs.

2.1 Equivalence Proofs

Propositions are stated in type signatures in dependently typed languages. Figure 2 is an example of such a proposition. To a human, the fact that the

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

Figure 2: Example Identity

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, to convince the compiler we need to specify every instance of associativity and identity, rewriting the left-hand-side repeatedly until it matches the right:

```
1  ident w x y z =
2    begin
3      ((w • ε) • (x • y)) • z
4    ≈⟨ assoc (w • ε) (x • y) z ⟩
5      (w • ε) • ((x • y) • z)
6    ≈⟨ identityʳ w ⟨ •-cong ⟩ assoc x y z ⟩
7      w • (x • (y • z))
8    ≈⟨ sym (assoc w x (y • z)) ⟩
9      (w • x) • (y • z)
10   ■
```

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

Because we're not using propositional equality, some familiar tools are unavailable, like Agda's rewrite mechanism, or function congruence (this is why we have to explicitly specify the congruence we're using on line 6). The purpose of this particular hair shirt is flexibility: users can still use the solver even if their type only satisfies the monoid laws modulo some equivalence relation (perhaps they are have an implementation of finite, mergeable sets as balanced trees, and want to treat two sets as equivalent if their elements are equal, even if their internal structures are not). Beyond flexibility, we get some other interesting applications, which are explored in section **??**.

Despite the pleasant syntax, the proof is mechanical, and it's clear that similar proofs would become tedious with more variables or more complex algebras (like rings). To avoid the tedium, then, we automate the procedure.

2.2 Canonical Forms

Automation of equality proofs like the one above can be accomplished by first rewriting both sides of the equation into a canonical form. Not every algebra has a canonical form: monoids do, though, and it's the simple list.

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

We're going to treat this type like an AST for a simple "language of lists". This language supports two functions: the empty list, and concatenation.

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

The type itself parameterized by the number of variables it contains. Users can refer to variables by their index:

```
infix 9 η_
η_ : ∀ {i} → Fin i → List i
η x = x :: []
```

And we can interpret this language with values for each variable supplied in a vector:

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

Compare this language to the language of monoid expressions that Figure 2 uses: both have identity elements and a binary operator, and both refer to variables. Our language of lists, however, has one significant advantage: the monoid operations don't depend on the contents of the lists, only the structure. In other words, an expression in the language of lists will reduce to a flat list even if it has elements which are abstract variables. As a result, the identity from Figure 2 is *definitionally* true when written in the language of lists:

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

2.3 Extracting Evidence

While this is beginning to look like a solver, it's still not entirely clear how we're going to join up the pieces. The first step is to get a concrete representation of expressions which we can manipulate and pattern-match on (Figure 3). It has constructors for

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

Figure 3: The AST for monoid expressions

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.

We can convert from this AST to an unnormalized expression like so[1]:

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

Because this function performs no normalization or transformation, the output is definitionally equal to its equivalent expression. This means that we can discharge the proof obligation with refl:

[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 [1] for an implementation.

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

The AST might look ugly, but it gives us a link between the expressions we had in Figure 2 and a concrete representation. The next link is between the AST and the normalized expression. We can convert from the AST to a normal form like so:

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

And since we already defined how to "evaluate" the list normal form, we can combine both steps into one:

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

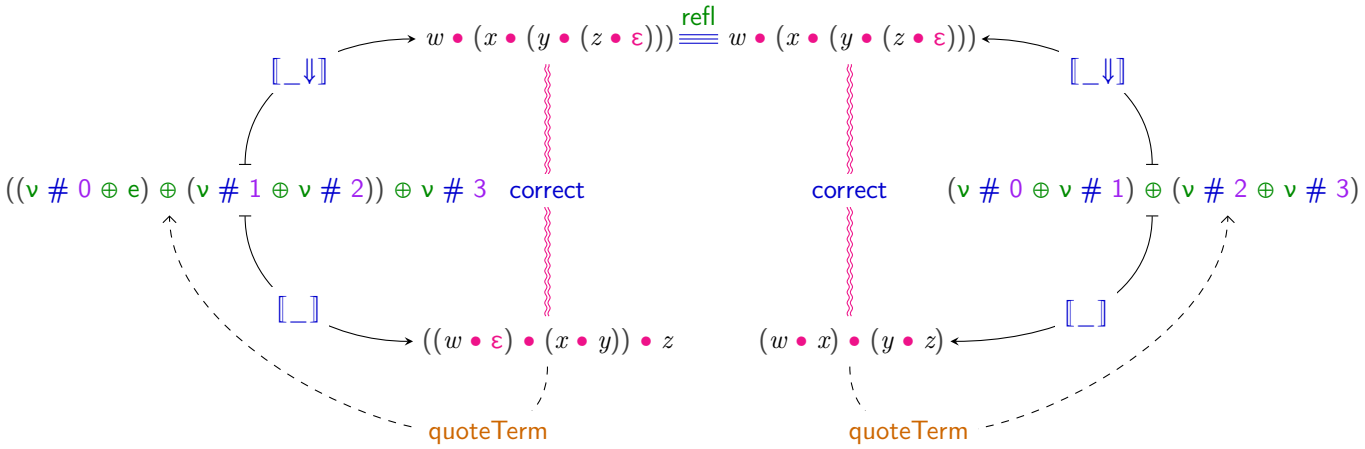
Now we have a concrete way to link the normalized and non-normalized forms of the expressions. A diagram of the strategy for constructing our proof is in Figure 4. The goal is to construct a proof of equivalence between the two expressions at the bottom: to do this, we first construct the AST which represents the two expressions (for now, we'll assume the user constructs this AST themselves. Later we'll see how too construct it automatically from the provided expressions). 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, which is the task of the next section.

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



;

Figure 4: The Reflexive Proof Process