Automatically And Efficiently Illustrating Polynomial Equalities in Agda

Donnacha Oisín Kidney

December 9, 2018

Abstract

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings in the programming language Agda [12]. The library makes use of Agda's reflection machinery to provide an extremely simple interface, and is extremely flexible in its output, requiring only equivalence (not propositional equality) to construct proofs.

Contents

1 Introduction

Truly formal proofs of even basic mathematical identities are notoriously tedious and verbose. Perhaps the canonical example is Russell and Whitehead's proof that $1 + 1 = 2$, which finally arrives on page 379 of Principia Mathematica [16].

More modern systems have greatly simplified the underlying formalisms, but they still often suffer from a degree of explicitness that makes elementary identities daunting. Dependently-typed programming languages like Agda [12] and Coq [14] are examples of such systems: used in the naïve way, equivalence proofs require the programmer to specify every individual step ("here we rely on the commutativity of $+$, followed by the associativity of $\times$ on its right side", and so on).

Coq and Agda are not just programming languages in name, though: they are fully-fledged and powerful, capable of producing useful software, including automated computer-algebra systems. Unlike most CASs, those written in Coq or Agda come with added guarantees of correctness in their operation. Furthermore, these systems can be used to automate the construction of identity proofs which would otherwise be too tedious to do by hand.

2 Related Work

The state-of-the-art solver for polynomial equalities (over commutative rings) was originally presented in [7], and is used in Coq's `ring` solver. This work improved on the already existing solver [5] in both efficiency and flexibility. In both the old and improved solvers, a reflexive technique is used to automate the construction of the proof obligation (as described in [1]).

Agda [12] is a dependently-typed programming language based on Martin-Löf's Intuitionistic Type Theory [9]. Its standard library [6] 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 [13], an implementation of an automated solver for the dependently-typed language Idris [2] is described. It uses type-safe reflection to provide a simple and elegant interface, and its internal solver algorithm uses a correct-by-construction approach. The solver is defined over *non*commutative rings, however, meaning that it is more general (can work with more types) but less powerful (meaning it can prove fewer identities). It does not use a sparse 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 [4], and specific to Agda in [15].

The progress of various formalization efforts is charted in [17]. DoCon [11] is a notable Agda library in this regard: its implementation and goal is described in [10]. [3] describes the manipulation of polynomials in both Haskell and Agda.

Finally, the study of *didactic* computer algebra systems is explored in [8].

3 Contributions

An New, Efficient Ring Solver We provide an implementation of a polynomial solver which uses the same optimizations described in [7] in the programming language Agda. Along the way, we demonstrate several techniques for writing efficient correct-by-construction code.

A Simple Reflection-Based Interface We use Agda's reflection machinery to provide the following interface to the solver:

```
lemma : ∀ x y →
  (x + y) ^ 2 ≈ x ^ 2 + y ^ 2 + 2 * x * y
lemma = solve NatRing
```

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.

A Didactic Computer-Algebra System As a result of the flexibility of the solver, the equivalence relation it constructs can be instantiated into a number of different forms (not just equality, for instance). While This has been exploited in Agda before to generate isomorphisms over containers, we use it here to construct didactic (or "step-by-step") solutions.

4 Explaining The Reflexive Technique With Monoids

Before jumping into commutative rings, we will first illustrate a general technique for automatically constructing equivalence proofs over a simpler algebra—*monoids*.

Definition 4.1 (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).

4.1 A "Trivial" Identity

As a running example for this section, we will use the identity in figure 2. To a human, the fact that the identity holds may well be obvious: $\bullet$ is associative, so we can scrub out all the parentheses, and ε is the

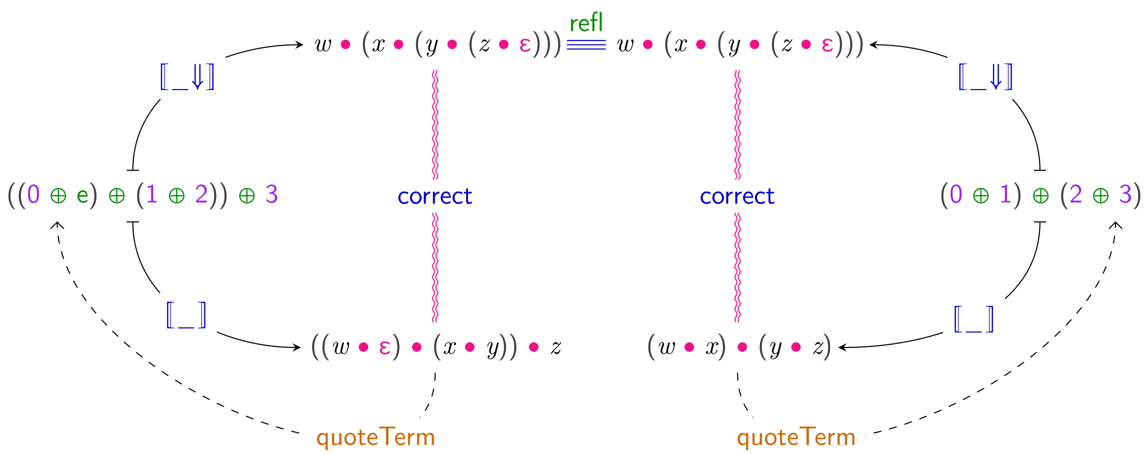

Figure 1: The Reflexive Proof Process

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

Figure 2: A Simple Identity Of Monoids

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:

```
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 handwritten proof: line 3 is the expression on the left-hand side of $\approx$ 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 $\bullet$ is used on line 4.

One trick worth pointing out is on line 6: the $\bullet$-cong lifts two equalities to either side of a $\bullet$. In other words, given a proof that $x_1 \approx x_2$, and $y_1 \approx y_2$, it will provide a proof that $x_1 \bullet y_1 \approx x_2 \bullet y_2$. This function needs to be explicitly provided by the user, as we only require $\approx$ to be an equivalence relation (not just propositional equality). In other words, we don't require it to be substitutive.

4.2 ASTs for the Language of Monoids

The first hurdle for automatically constructing proofs comes from the fact that the identity is opaque: it's hidden behind a lambda. We can't scrutinize or pattern-match on its contents. Our first step, then, is to define an AST 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
```

We have constructors for both monoid operations, and a way to refer to variables. These 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 2:

```
((0 ⊕ e) ⊕ (1 ⊕ 2)) ⊕ 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 refult is *definitionally* equal to the original expression[1]:

```
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.

[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 [6] for an implementation.

4.3 Canonical Forms

In both the monoid and ring solver, we will make use of the *canonical forms* of expressions in each algebra. Like the AST we defined above, these canonical forms represent expressions in the algebra, however *unlike* the AST, they definitionally obey the laws of the algebra.

For monoids, the canonical form is *lists*.

```
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:

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

And finally (as promised) the opening identity 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
```

Or, combining both steps into one:

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

4.4 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 1. 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 this section.

Taking the non-normalizing interpreter as a template, the three cases are as follows[2]:

$$[\![x \oplus y]\!]\rho \approx [\![x \oplus y \Downarrow]\!]\rho \tag{1}$$

$$[\![\mathsf{e}]\!]\rho \approx [\![\mathsf{e}\Downarrow]\!]\rho \tag{2}$$

$$[\![\nu i]\!]\rho \approx [\![\nu i \Downarrow]\!]\rho \tag{3}$$

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] Equations 1 and 2 comprise a monoid homomorphism.

4.5 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 in figure 2:

```
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
```

Some of the duplication is undesirable: we will later use reflection to eliminate it.

References

[1] 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.

[2] 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

[3] 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

[4] 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

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

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

[7] 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

[8] 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

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

[10] 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

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

[12] U. Norell and J. Chapman, “Dependently Typed Programming in Agda,” p. 41, 2008.

[13] 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

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

[15] 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

[16] A. N. Whitehead and B. Russell, *Principia Mathematica. Vol. I*, 1910. [Online]. Available: https://zbmath.org/?q=an%3A41.0083.02

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