Automatically And Efficiently Illustrating Polynomial Equalities in Agda—Extended Abstract

Donnacha Oisín Kidney

February 21, 2019

Abstract

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings and semirings in the programming language Agda [3]. It is asymptotically faster than Agda's existing solver. We use Agda's reflection machinery to provide a simple interface to the solver, and demonstrate a novel use of the constructed relations: step-by-step solutions.

The library is available at oisdk.github.io/agda-ring-solver/README.html.

Contents

1 Introduction

What does it mean to write mathematics in a programming language? One interpretation might involve SciPy [1] or R [4]. These systems definitely seem maths-related: they can run calculations, solve equations, and are used extensively in modern maths research. But this interpretation doesn't feel quite right: if I were to publish a paper tomorrow on something involving SciPy, even if there were loads of code snippets included, the proofs would all be in *English*, not Python.

Another interpretation might look at computer science: a huge amount of maths is devoted to describing programming languages and their behaviour. Again, though, this seems off the mark. Here it feels more like we're doing mathematics "about" a programming language, rather than *with* one.

There is a third interpretation, one which involves a particular type of programming languages, which is the focus of this paper. For these languages, there is no gap between a program and the mathematical concepts it describes.

1.1 Mathematics as a Programming Language

While most maths is still today written in prose, there are small[1]languages (systems of notation, really) that are closer to programming languages. One example might be predicate logic:

$$\text{Raining} \wedge \text{Outside} \implies \text{Wet}$$

Already, you may be thinking, we can do this in a programming language. Surely I'm not saying that we don't have boolean logic in programming?

```
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
False && _     = False
True  && False = False
True  && True  = True
```

[1]Small in comparison to most programming languages!

$$\textit{Type} \Longleftrightarrow \textit{Proposition}$$

$$\textit{Program} \Longleftrightarrow \textit{Proof}$$

Figure 1: The Curry-Howard Isomorphism

That's exactly what I'm saying! The code snippet above isn't meaningful in the same way the proposition is: the symbols `False` and `True` have no meaning other than being two constructors for a type named `Bool`. It's not the programming language which gives the `&&` symbol its meaning as logical conjunction, it's *us*, the readers!

But that's true for anything written in Haskell, you might think. We're the ones who give it meaning, without us a Haskell program is just a text file. However, there is a third party, a higher power which *really* gives that program above its meaning: the *compiler*. For most Haskell written today, GHC is the thing which gets to say what it means, with no arbitrariness of names involved.

So it turns out that we do have a language that can be thought of as objectively as predicate logic. The next question is *can we do maths in it*?

1.2 Programs Are Proofs

To understand how it's possible to "do maths" in a programming language, it's helpful to look at figure 1. The two constructs we're interested on the maths end are propositions ("there are infinitely many prime numbers", or, "the square root of two is irrational") and *proofs* (left to the reader).

At first glance, the isomorphism seems strange: types are things like `Int`, `String`, and so on. To make the two sides match up, imagine before every type you said "there exists". For instance, this is the proposition that "an integer exists"

```
prop :: Integer
```

The proof of this proposition is a very small, but nonetheless valid, program. We say "an integer does exist. Here's one, for example!"

```
prop = 3
```

Still, this seems quite far from even the simple logical statement above. To push the point a little further, let's try and write something which is *false*:

```
head :: [a] -> a
```

This is the proposition that "there exists a program which takes a list of `a`s, and gives you back an `a`". The problem is that such a program *doesn't* exist! What would it do, for instance, in the following circumstance:

```
head []
```

Throw an error? Loop? Nothing sensible, regardless. Because of that, we say this program is *false*, or invalid.

One more example in Haskell: we've seen a better version of `True` (programs which compile and don't throw errors), a better version of `False` (programs which don't compile or *do* throw errors), so to complete the translation, let's write a better version of `&&`:

```
data And a b = And a b
```

Why is this better? Because `And a b` is really and truly a proof that both `a` and `b` are true. After all, the type of the constructor says you can't prove `And a b` without first proving `a` and proving `b`.

```
And
  :: a -- A proof of a
  -> b -- A proof of b
  -> And a b -- A proof of (And a b)
```

We can even write some of our favourite logic rules:

$$A \wedge B \implies A$$

```
fst :: And a b -> a
fst (And x y) = x
```

1.3 Agda

While the logical rules expressed above are cute, it does become (as you might imagine) quite difficult to express more complex propositions in languages like Haskell. The idea of the isomorphism is still valid, though, we just need a slightly fancier type system to make things more ergonomic.

Agda is a language based on (and implemented in) Haskell, which has one of these “slightly fancier type systems”. Its particular flavour was invented by Per Martin-Löf [2].

One other difference worth pointing out is that Agda doesn’t even allow you to *compile* programs like `head`: this ensures that every proof written in Agda is really a proof, not just “a proof as long as it doesn’t crash at any point”[2].

References

[1] E. Jones, T. Oliphant, P. Peterson *et al.*, *SciPy: Open Source Scientific Tools for Python*, 2001. [Online]. Available: http://www.scipy.org/

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

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

[4] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2013. [Online]. Available: http://www.R-project.org/

[2] To be totally accurate, this delineates the difference between what people call the “Curry-Howard Isomorphism” and the “Curry-Howard *correspondence*”. Haskell and the vast majority of programming languages can only be described by the latter: proofs in them are only valid modulo nontermination. Agda, Idris, Coq, and others belong to the former camp, which (along with their fancier type systems) makes them more suited to proving things.