Automatically And Efficiently Illustrating Polynomial Equalities in Agda—Extended Abstract*

DONNACHA OISÍN KIDNEY†

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings and semirings in the programming language Agda [6]. 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.

1 INTRODUCTION

What does it mean to write mathematics in a programming language? One interpretation might involve SciPy [3] or R [7]. 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[2]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
```

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*?

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

†Student number: 115702295

[2]Small in comparison to most programming languages!

1.2 Programs Are Proofs

To understand how it's possible to "do maths" in a programming language, it's helpful to look at Fig. 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).

$$Type \iff Proposition$$

$$Program \iff Proof$$

Fig. 1. The Curry-Howard Isomorphism

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`. We can even write some of our favourite logic rules: $A \wedge B \implies A$, for instance.

```
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 [5].

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"[3].

[3] 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

```
lemma : ∀ x y → x + y * 1 + 3 ≈ 2 + 1 + y + x

lemma x y = begin
  x + y * 1 + 3 ≈⟨ refl ⟨ +-cong ⟩ *-identityʳ y ⟨ +-cong ⟩ refl {3} ⟩
  x + y + 3     ≈⟨ +-comm x y ⟨ +-cong ⟩ refl ⟩
  y + x + 3     ≈⟨ +-comm (y + x) 3 ⟩
  3 + (y + x)   ≈⟨ sym (+-assoc 3 y x) ⟩
  2 + 1 + y + x ■
```

(a) A Tedious Proof

```
lemma = solve NatRing
```

(b) Our Solver

Fig. 2. Comparison Between A Manual Proof and The Automated Solver

2 A SOLVER FOR RINGS IN AGDA

This work describes a tool to make it easier to do mathematics in Agda: a library which automates the construction of proofs like Fig. 2a, making them as easy as Fig. 2b.

Along with correctness, the main goals of this work were as follows:

Ease of Use Proofs like the one in Fig. 2a are difficult to write. The programmer needs to remember the particular syntax for each step ("is it +-comm or +-commutative?"), and often they have to put up with poor error messages.
Our solver strives to be as easy to use as possible: the high-level interface is simple (Fig. 2b), we don't require anything of the user other than an implementation of one of the supported algebras, and effort is made to generate useful error messages.

Performance Typechecking dependently-typed code is a costly task. Automated solvers like the one presented here can greatly exacerbate this cost: in our experience, it wasn't uncommon for Agda's current ring solver to spend upwards of 10 minutes proving a single identity.
The kind of solver we provide here is based on Coq's [8] `ring` tactic, described in Grégoire and Mahboubi [2]. While we were able to apply the same optimisations that were applied in that paper, we found that the most significant performance improvements came from a different, and somewhat surprising part of the program. In terms of both practical use and theoretical bounds, our solver is significantly faster than Agda's current solver.

Educational Features Outside the rigorous world of dependently-typed languages, computer algebra systems (of one form or another) have had massive success among programmers and non-programmers alike. These days, many of these systems (like Wolfram|Alpha [10]) provide educational features, which have proven invaluable to students learning mathematics.
We will take just one of those features ("pedagogical", or step-by-step solutions [9]), and re-implement it in Agda using our solver. In doing so, we will formalise the concept, and explore some of the theory behind it.

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.

```
lemma = +-*-Solver.solve 2 (λ x y → x :+ y :* con 1 :+ con 3 := con 2 :+ con 1 :+ y :+ x) refl
```

Fig. 3. The Old Solver

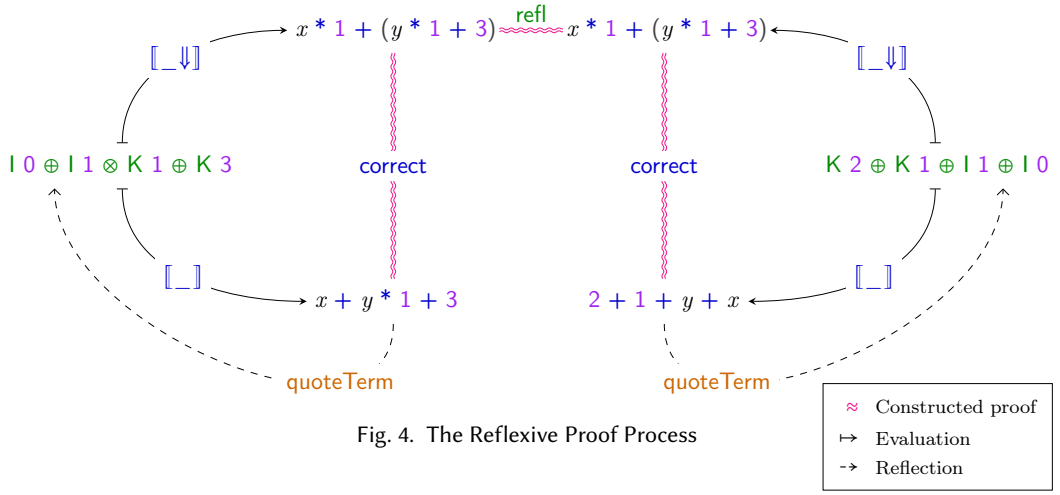

Fig. 4. The Reflexive Proof Process

3 OVERVIEW OF THE PROOF TECHNIQUE

There are a number of ways we can automate proofs in a dependently-typed programming language. Here, we will use a reflexive technique [1] in combination with sparse Horner Normal Form. The high-level diagram of the proof strategy is presented in Fig. 4.

The identity we'll be working with is the lemma in Fig. 2: the left and right hand side of the equivalence are at the bottom of the diagram. Our objective is to link those two expressions up through repeated application of the ring axioms. We do this by converting both expressions to a normal form (seen at the top of the diagram), and then providing a proof that this conversion is correct according to the ring axioms (the correct function in the diagram). Finally, we link up all of these proofs, and if the two normal forms are definitionally equal, the entire thing will typecheck, and we will have proven the equivalence.

3.1 Almost Rings

While the stated domain of the solver is simply "commutative rings", it turns out that we can be slightly more flexible than that if we pick our algebra carefully. As in Grégoire and Mahboubi [2, section 5], we use an algebra called an *almost-ring*. It has the regular operations ($+$, $*$ (multiplication), $-$, 0, and 1), such that the following equations hold:

$$0 + x = x \tag{1}$$
$$x + y = y + x \tag{2}$$
$$x + (y + z) = (x + y) + z \tag{3}$$
$$1 * x = x \tag{4}$$
$$x * y = y * x \tag{5}$$
$$x * (y * z) = (x * y) * z \tag{6}$$
$$(x + y) * z = x * z + y * z \tag{7}$$
$$0 * x = 0 \tag{8}$$
$$-(x * y) = -x * y \tag{9}$$
$$-(x + y) = -x + -y \tag{10}$$

The equations up to 8 represent a pretty standard definition of a (commutative) semiring. From there, though, things are different. The normal definition of a commutative ring would have (instead of 9 and 10) $x + -x = 0$. However, by choosing these slightly more complex laws, we can admit types like $\mathbb{N}$ which don't have additive inverses. Instead, these types can simply supply the identity function for $-$, and then 9 and 10 will still hold.

A potential worry is that because we don't require $x + -x = 0$ axiomatically, it won't be provable in our system. Happily, this is not the case: as long as $1 + -1$ reduces to 0 in the coefficient set, the solver will verify the identity.

3.2 Correctness

The correct function amounts to a proof of soundness: i.e., the solver will only ever prove equations which are genuinely equivalent. We have not, however, proven completeness (i.e., that every genuine equivalence will be proven by our solver), and indeed it is not possible to do so.

In the internal representation of the solver, we prove several data structure invariants (like sparsity) intrinsically.

The reflection-based interface is unproven, but this does not invalidate the solver: any errors will be caught by the typechecker, meaning that we are still prevented from proving things that are untrue.

4 THE INTERFACE

We kept the surface area of the library quite small: aside from the AlmostCommutativeRing type described above, the rest of the interface consists of just two macros (solve and solveOver). We tried to make their usage as obvious as possible: just stick one of them (with the required arguments) in the place you need a proof, and the solver will do the rest for you.

```
lemma : ∀ x y → x + y * 1 + 3 ≈ 2 + 1 + y + x
lemma x y =
  begin
    x + y * 1 + 3   ≈⟨ +-comm (x + y * 1) 3 ⟩
    3 + (x + y * 1) ≈⟨ solveOver (x :: y :: []) Nat.ring ⟩
    3 + y + x       ≡⟨⟩
    2 + 1 + y + x   ■
```

Fig. 5. Demonstration of the solveOver macro

solve is demonstrated in Fig. 2b. It takes a single argument: an implementation of the algebra. solveOver is designed to be used in conjunction with manual proofs, so that a programmer can automate a "boring" section of a larger more complex proof. It is demonstrated in Fig 5.

5 PERFORMANCE

As expected, the sparse implementation exhibits a significant speedup in type-checking over the dense implementation (even with the added overhead of the reflection interface). Fig. 6a shows time taken to type check a proof that $(x_1 + x_2 + x_3 + x_4 + x_5)^d$ is equal to its expanded form. The sparse representation is clearly faster overall, with a factor of 7.5 speedup for $d = 8$.

Fig. 6c demonstrates perhaps a more common use-case, with a mix of high and low powers and some constants. The sparse representation's advantage is even more apparent here, with an 30-factor speedup at $d = 8$.

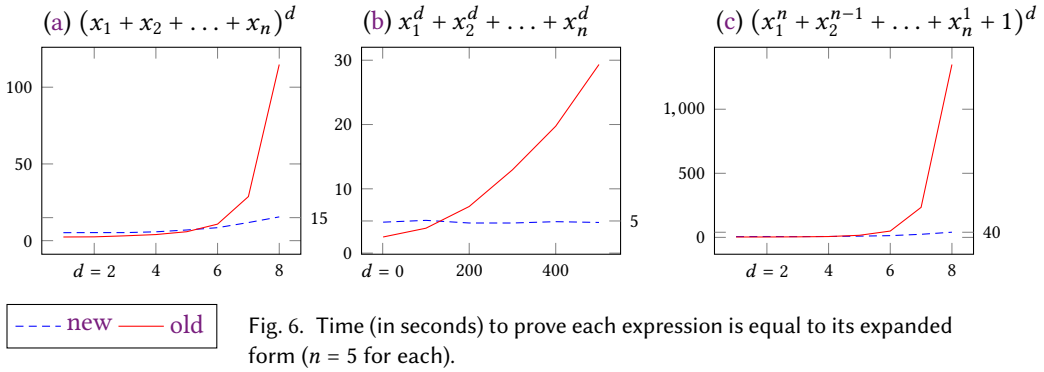

Fig. 6. Time (in seconds) to prove each expression is equal to its expanded form ($n = 5$ for each).

6 PEDAGOGICAL SOLUTIONS

One of the most widely-used and successful computer algebra systems, especially among non-programmers, is Wolfram|Alpha [10]. It can generate "pedagogical" (step-by-step) solutions to maths problems [9].

In Lioubartsev [4] the problem is reformulated as one of *path-finding*. The left-hand-side and right-hand-side of the equation are vertices in a graph, where the edges are single steps to rewrite an expression to an equivalent form. A* is used to search.

Unfortunately, this approach has to deal with a huge search space: every vertex will have an edge for almost every one of the ring axioms, and as such a good heuristic is essential. Furthermore, what this should be is not clear: Lioubartsev [4] uses a measure of the "simplicity" of an expression.

```
x + y * 1 + 3
    ={ eval }
x + y + 3
    ={ +-comm(x,y + 3) }
y + 3 + x
    ={ +-comm(y,3) }
3 + y + x
    ={ eval }
2 + 1 + y + x
```

Fig. 7. Pedagogical Output from Our Solver

So, with an eye to using our solver to help, we can notice that paths in undirected graphs form a perfectly reasonable equivalence relation, where the equivalence classes are connected components of the graph.

In this was, we can have our solver generate steps to rewrite an equation from one form to another. To make that output shorter and more human readable, we use the following techniques:

(1) First, we run (a version of) Dijkstra's algorithm on the generated path. This will remove any unnecessary loops in the solution. In contrast to using just A* on its own, the search space is minimal (with only one outward edge for each vertex).
(2) Then, we filter out "uninteresting" steps. These are steps which are obvious to a human, like associativity, or evaluation of closed terms. When a step is divided over two sides of an operator, it is deemed "interesting" if either side is interesting.

After applying those heuristics, our solver outputs Fig. 7 for the lemma in Fig. 2b.

REFERENCES

[1] Samuel Boutin. 1997. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software (Lecture Notes in Computer Science)*, Martín Abadi and Takayasu Ito (Eds.). Springer Berlin Heidelberg, 515–529.

[2] Benjamin Grégoire and Assia Mahboubi. 2005. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science)*, Vol. 3603. Springer Berlin Heidelberg, Berlin, Heidelberg, 98–113. https://doi.org/10.1007/11541868_7

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001. *SciPy: Open Source Scientific Tools for Python.* http://www.scipy.org/

[4] Dmitrij Lioubartsev. 2016. *Constructing a Computer Algebra System Capable of Generating Pedagogical Step-by-Step Solutions.* Ph.D. Dissertation. KTH Royal Institue of Technology, Stockholm, Sweden. http://www.diva-portal.se/smash/get/diva2:945222/FULLTEXT01.pdf

[5] Per Martin-Löf. 1980. *Intuitionistic Type Theory.* Padua. http://www.cse.chalmers.se/~peterd/papers/MartinL%00f6f1984.pdf

[6] Ulf Norell and James Chapman. 2008. Dependently Typed Programming in Agda. (2008), 41.

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

[8] The Coq Development Team. 2018. The Coq Proof Assistant, Version 8.8.0. https://doi.org/10.5281/zenodo.1219885

[9] The Development Team. 2009. Step-by-Step Math. http://blog.wolframalpha.com/2009/12/01/step-by-step-math/

[10] Wolfram Research, Inc. 2019. Wolfram|Alpha. Wolfram Research, Inc.. https://www.wolframalpha.com/