Reading and Writing Arithmetic: Automating Ring Equalities in Agda

ANONYMOUS AUTHOR(S)

We present a new library which automates the construction of equivalence proofs between polynomials over commutative rings and semirings in the programming language Agda [Norell and Chapman 2008]. It is asymptotically faster than Agda's existing solver. We use reflection to provide a simple interface to the solver, and demonstrate a novel use of the constructed relations: step-by-step solutions.

Additional Key Words and Phrases: proof automation, equivalence, proof by reflection, step-by-step solutions

$$\text{lemma} : \forall \ x \ y \to x + y * 1 + 3 \approx 2 + 1 + y + x$$

```
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  ∎
```

```
lemma = solve NatRing
```

(a) A Tedious Proof   (b) The Solver

Fig. 1. Comparison Between A Manual Proof and The Automated Solver

1 INTRODUCTION

Doing mathematics in dependently-typed programming languages like Agda has a reputation for being tedious, awkward, and difficult. Even simple arithmetic identities like the one in Fig. 1 require fussy proofs (Fig. 1a).

This need not be the case! With some carefully-designed tools, mathematics in Agda can be easy, friendly, and fun. This work describes one such tool: an Agda library which automates the construction of these kinds of proofs, making them as easy as Fig. 1b.

As you might expect, our solver comes accompanied by a formal proof of correctness. Beyond that, though, we also strove to satisfy the following requirements:

Friendliness and Ease of Use Proofs like the one in Fig. 1a aren't just boring: they're *difficult*. 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.

We believe this kind of difficulty is why Agda's current ring solver [Danielsson 2018] enjoys little widespread use. Its interface (Fig. 2) is almost as verbose as the manual proof, and it requires programmers to learn another syntax specific to the solver.

Our solver strives to be as easy to use as possible: the high-level interface is simple (Fig. 1b), 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.

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

Fig. 2. The Old Solver

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.
In practice, this means two things: firstly, large libraries for formalizing mathematics (like
Meshveliani [2018]) can potentially take hours to typecheck (by which time the programmer
has understandably begun to reconsider the whole notion of mathematics on a computer);
secondly, certain identities can simply take too long to typecheck, effectively making them
"unprovable" in Agda altogether!
The kind of solver we provide here is based on Coq's [Team 2018] ring tactic, described in
Grégoire and Mahboubi [2005]. While we were able to apply the same optimizations 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. The end result is that
our solver is asymptotically (and practically) faster than Agda's current solver.

Educational Features While our solver comes with the benefit of formal correctness, it's
still playing catch-up to other less-rigorous computer algebra systems in terms of features.
These features have driven systems like Wolfram|Alpha [Wolfram Research, Inc. 2019] to
widespread popularity among (for instance) students learning mathematics.
We will take just one of those features ("pedagogical", or step-by-step solutions [The Development Team 2009]), and reimplement it in Agda using our solver. In doing so, we will
explore some of the theory behind it, and present a formalism that describes the nature of
"step-by-step" solutions.

2 OVERVIEW OF THE PROOF TECHNIQUE

There are a number of ways we can automate proofs in a dependently-typed programming language,
including Prolog-like proof search [Kokke and Swierstra 2015], Cooper's algorithm over Presburger
arithmetic [Allais 2011], etc. Here, we will use a reflexive technique [Boutin 1997] in combination
with sparse Horner Normal Form. The high-level diagram of the proof strategy is presented in
Fig. 3.

The identity we'll be working with is the lemma in Fig. 1: the left and right hand side of the
equality 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 equality.

2.1 The Expr AST

In Agda, we can't manipulate the expressions we want to prove directly: instead, we will construct
an AST for each expression, and then do our manipulation on that.

The AST type (Expr) has a constructor for each of the ring operators, as well as constructors
for both variables and constants. The ASTs for both expressions we want to prove can be seen on
either side of Fig. 3. Constants are constructed with K, and variables are referred to by their de
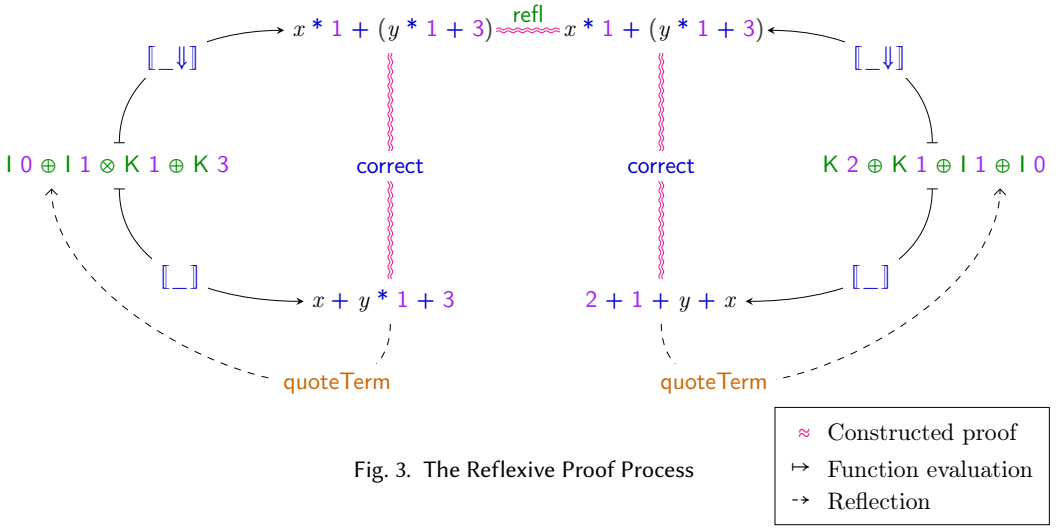Bruijn index (so x becomes I 0).

Fig. 3. The Reflexive Proof Process

≈	Constructed proof
↦	Function evaluation
⇢	Reflection

From here, we can "evaluate" the AST in one of two ways: in a non-normalized way ($[\![\_]\!]$), or in a normalizing way ($[\![\_\Downarrow]\!]$). This means that the goal of the correct function is to show equivalence between $[\![\_]\!]$ and $[\![\_\Downarrow]\!]$.

Finally, we *don't* want to force users to construct the Expr AST themselves. This is where reflection comes in: it automates this construction (the path labeled quoteTerm in the diagram) from the goal type.

2.2 Choice of Algebra

So far, we have been intentionally vague about the precise algebra we're using. As in [Grégoire and Mahboubi 2005], 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. To get from there to a ring, we would usually ask for something like:

$$x + -x = 0 \tag{11}$$

So why do we instead ask for 9 and 10?

Under this formulation, 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 THE INTERFACE

4 PERFORMANCE

5 PEDAGOGICAL PROOFS

6 RELATED WORK

REFERENCES

G Allais. 2011. Deciding Presburger Arithmetic Using Reflection. (May 2011). https://gallais.github.io/pdf/presburger10.pdf

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.

Nils Anders Danielsson. 2018. The Agda Standard Library. https://agda.github.io/agda-stdlib/README.html

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

Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, 276–301. http://www.staff.science.uu.nl/~swier004/publications/2015-mpc.pdfhttp://www.staff.science.uu.nl/~swier004/publications/2015-mpc.pdfhttp://www.staff.science.uu.nl/~swier004/publications/2015-mpc.pdf

Sergei D. Meshveliani. 2018. DoCon-A a Provable Algebraic Domain Constructor. http://www.botik.ru/pub/local/Mechveliani/docon-A/2.02/

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

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

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

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

A APPENDIX

Text of appendix . . .