Reading and Writing Arithmetic: Automating Ring Equalities in Agda

ANONYMOUS AUTHOR(S)

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

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

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.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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 formalising 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 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. 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 re-implement 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 Ι 0).

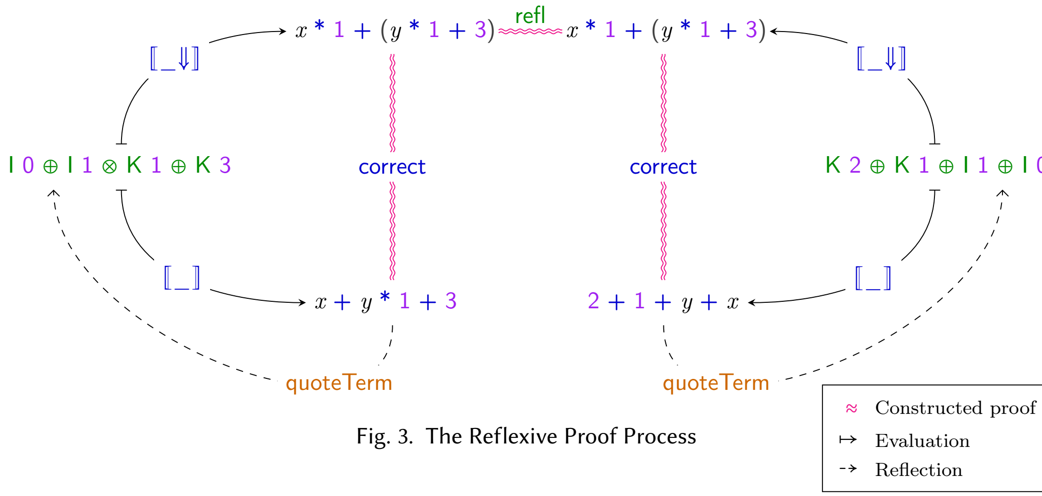

Fig. 3. The Reflexive Proof Process

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

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 Almost Rings

So far, we have been intentionally vague about the precise algebra we're using. As in Grégoire and Mahboubi [2005, 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) the following:

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

The reason for the difference is *flexibility*. 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.

In the library, the algebra is represented by the AlmostCommutativeRing type, a record with fields for each of the ring axioms, defined over a user-supplied equivalence relation. Just as in Agda's current solver, we also ask for one extra function: a weakly decidable predicate to test if a constant is equal to zero.

```
is-zero : ∀ x → Maybe (0# ≈ x)
```

This function is used to speed up some internal algorithms in the solver, but it isn't an essential component. By making it *weakly* decidable, we allow users to skip it (is-zero = const nothing) if their type doesn't support decidable equality, or provide it (and get the speedup) if it does.

3 THE INTERFACE

A decent interface is crucial if we want the solver to be broadly useful. We strove to make our interface as simple and *small* as possible. Aside from the AlmostCommutativeRing type described above, the user-facing portion of our library consists of just two macros: solve and solveOver. Each of these infer the goal from their context, and automatically construct the required machinery to prove the equality.

solve is demonstrated in Fig. 1b. 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 called like so:

```
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   ∎
```

As well as the AlmostCommutativeRing implementation, this macro takes a list of free variables to use to compute the solution.

Because this interface is quite small, it's worth pointing out what's missing, or rather, what we *don't* require from the user:

- We don't ask the user to construct the Expr AST which represents their proof obligation. Compare this to Fig. 2: we had to write the type of the proof twice (once in the signature and again in the AST), and we had to learn the syntax for the solver's AST.
  As well as being more verbose, this approach is less composable: every change to the proof type has to be accompanied by a corresponding change in the call to the solver. In contrast, the call to solveOver above effectively amounts to a demand for the compiler to "figure it out!" Any change to the expressions on either side will result in an *automatic* change to the proof constructed.
- We don't ask the user to write any kind of "reflection logic" for their type. In other words, we don't require a function which (for instance) recognizes and parses the user's type in the

reflected AST, or a function which does the opposite, converting a concrete value into the AST that (when unquoted) would produce an expression equivalent to the quoted value. This kind of logic is complex, and very difficult to get right. While some libraries can assist with the task [Norell 2018; van der Walt and Swierstra 2013] it is still not fully automatic.

Finally, despite the simplicity and ease-of-use described above, the solver is *not* specialized to a small number of types like ℕ and so on. The whole library, including the reflection-based interface, will work with any type with an AlmostCommutativeRing instance.

3.1 Reflection

Agda has good support for reflection, which we will use to build our interface. Agda's reflection system consists of three main parts:

Term The representation of Agda's AST, retrievable via quoteTerm.

Name The representation of identifiers, retrievable via quote.

TC The type-checker monad, which includes scoping and environment information, can raise type errors, unify variables, or provide fresh names. Computations in the TC monad can be run with unquote.

While quote, quoteTerm, and unquote provide all the functionality we need, they're somewhat low-level and noisy (syntactically speaking). Agda also provides a mechanism (which it calls "macros") to package metaprogramming code so it looks like a normal function call (as in solve).

Reflection is obviously a powerful tool, but it has a reputation for being unsafe and error-prone. Agda's reflection system doesn't break type safety, but we *are* able to construct Terms which are ill-typed, which often result in confusing error-messages on the user's end. Unfortunately, constructing ill-typed terms is quite easy to do: every quoted expression comes with heaps of contextual information, making the whole thing very fragile. Variables, for instance, are referred to by their de Bruijn indices, meaning that the same Term can break if it's simply moved under a lambda.

All of that considered, we feel we managed to construct a reasonably robust reflection-based interface. In doing so, we came up with the following general guidelines for metaprogramming in Agda:

Supply the minimal amount of information. There were several instances where, in constructing a term, we were tempted to supply explicitly some argument that Agda usually infers. Universe levels were a common example. In general, though, this is a bad idea: AST manipulation is fragile and error-prone, so the chances that you'll get some argument wrong are very high. Instead, you should *leverage* the compiler, relying on inference over direct metaprogramming as much as possible.

Don't assume structure. A common pattern we used to try and find arguments to an n-ary function was to simply extract the last n visible arguments to the function. While in theory we might be able to statically know all of the implicit and explicit arguments that will be used at the call-site, it's much simpler to ignore them, and try our best to be flexible. Remember, none of this is typed, so if something changes (like, say, a new universe level in AlmostCommutativeRing) in the order of arguments, you'll get type errors where you call solve, not where it's implemented.

Ask for forgiveness, not permission. We could also here say "don't roll your own type-checker". While it may seem good and fastidious to rigorously check the structure of the arguments given to a macro, often we found better results by assuming the argument was correct (where possible), and then carefully structuring the output in such a way to funnel a type error to the place where the input was incorrect. For instance, one section of the solver algorithm expects a proof that the two normal forms of the equations are the same. Here, we simply supply refl, assuming that they are, in fact, the same. When they're *not*, for instance, in the following type:

$$x + y * 1 + 3 \approx 2 + 1 + y + y$$

A call to solve will provide the reasonably helpful error message:

$$x \neq y \text{ of type } \mathbb{N}$$

Try and implement as much of the logic outside of reflection as possible Finally, after all of that, we advise minimizing the amount of actual metaprogramming code in a program, and confining it to the edges as much as possible. With great power comes with poor error messages, fragility, and a loss of first-class status. Therefore, If something can be done without reflection, *do it*, and use reflection as the glue to get from one standard representation to another.

At the core of the implementation you will find the following function:

```
toExpr : Term → Term
toExpr (def (quote AlmostCommutativeRing._+_) xs) = getBinOp (quote _⊕_) xs
toExpr (def (quote AlmostCommutativeRing._*_) xs) = getBinOp (quote _⊗_) xs
toExpr (def (quote AlmostCommutativeRing._^_) xs) = getExp xs
toExpr (def (quote AlmostCommutativeRing.-_)  xs) = getUnOp (quote ⊖_) xs
toExpr v@(var x _) with x ℕ.<? numVars
... | yes p = v
... | no ¬p = constExpr v
toExpr t = constExpr t
```

This function is called on the Term representing one side of the target equality. It converts it to the corresponding Expr. In other words, it performs the following transformation:

$$x + y * 1 + 3 \mapsto \mathsf{I}\ 0 \oplus \mathsf{I}\ 1 \otimes \mathsf{K}\ 1 \oplus \mathsf{K}\ 3$$

It also demonstrates the principles described above. The first four clauses all match for the ring operators. Taking exponentiation as an example, it calls off to the getExp function, which is implemented as follows:

```
getExp : List (Arg Term) → Term
getExp (x ⟨∷⟩ y ⟨∷⟩ []) = quote _⊛_ ⟨ con ⟩ 3 ⋯⟅∷⟆ toExpr x ⟨∷⟩ y ⟨∷⟩ []
getExp (x ∷ xs) = getExp xs
getExp _ = unknown
```

As described above, this function looks only for the last two visible arguments to the exponentiation operator, ignoring all else. When it finds them, it applies the corresponding constructor for Expr, using the ⋯⟅∷⟆ function, which fills in all the hidden arguments we want the compiler to infer.

Looking back to toExpr, we notice that the last line is a catch-all, which simply constructs a constant expression. This is the trick which lets us avoid any custom quotation machinery from the user. It's also more robust than asking for custom quotation machinery: if, for instance, there's a function call or something similar hidden in this case, quotation won't work. This solution, though, which just packages up the expression as-is, will have no trouble.

4 PERFORMANCE

Type-checking proof-heavy Agda code is notoriously slow, so the solver had to be carefully optimized to avoid being so slow as to be unusable. We'll start by first describing the unoptimized solver, and demonstrate how to improve its performance iteratively.

4.1 Horner Normal Form

The representation used in Agda's current ring solver (and the one we'll start out with here) is known as Horner Normal Form. A polynomial (more specifically, a monomial) in x is represented as a list of coefficients of increasing powers of x. As an example, the following polynomial:

$$3 + 2x^2 + 4x^5 + 2x^7 \tag{12}$$

Is represented by the following list:

$$3, 0, 2, 0, 0, 4, 0, 2$$

Operations on these polynomials are similar to operations in positional number systems.

```
Poly : Set c
Poly = List Carrier

_⊞_ : Poly → Poly → Poly
[] ⊞ ys = ys
(x :: xs) ⊞ [] = x :: xs
(x :: xs) ⊞ (y :: ys) = x + y :: xs ⊞ ys

_⊠_ : Poly → Poly → Poly
_⊠_ [] _ = []
_⊠_ (x :: xs) =
  foldr (λ y ys → x * y :: map (_* y) xs ⊞ ys) []
```

And finally, evaluation of the polynomial (given x) is a classic example of the foldr function.

```
⟦_⟧ : Poly → Carrier → Carrier
⟦ xs ⟧ ρ = foldr (λ y ys → ρ * ys + y) 0# xs
```

4.2 Sparse Encodings

Our first avenue for optimization comes from Grégoire and Mahboubi [2005]. Our list encoding above is quite wasteful: it always stores an entry for each coefficient, even if it's zero. Since expressions with long strings of zeroes are common (things like x^{10}), it stands to reason that removing them should improve performance.

The solution is to store what's known as a "power index" with every coefficient. Intuitively, you can think of it as the "distance to the next non-zero coefficient". Taking 12 again as an example, we would now represent it as follows:

$$(3, 0), (2, 1), (4, 2), (2, 1)$$

In Agda, we can go one step further, by disallowing zeroes in the representation altogether. This statically ensures that the polynomial is always in its smallest possible form. We don't include that detail here (it is in the library), instead we will use this somewhat simplified type:

```
Poly : Set c
Poly = List (Carrier × ℕ)
```

Next, we turn our attention to the task of adding multiple variables. Luckily, there's an easy way to do it: nesting. The idea is that a polynomial in n variables is the same as before, except that its coefficients are themselves polynomials in $n - 1$ variables. A polynomial in 0 variables in just a constant. It's perhaps more clearly expressible in types:

```
Poly : ℕ → Set c
Poly zero = Carrier
Poly (suc n) = List (Poly n × ℕ)
```

Before jumping into proving this, though, it's worth noting that another opportunity for a "sparse" encoding has arisen. This time, polynomials which don't include every variable contain gaps. In a polynomial of n variables, a constant will always be stored behind n layers of nesting (we also prove that this minimal form is maintained).

The solution is another index: this time an "injection" index. This represents "how many variables to skip over before you get to the interesting stuff". This particular optimization is considerably more complex than the previous, though: the number of variables in a polynomial is a type-relevant piece of information, so any *manipulation* of that index will have to justify itself to the typechecker.

4.3 Hanging Indices

The problem is a common one: we have a piece of code that works efficiently, and we now want to make it "more typed", by adding more information to it, *without* changing the complexity class or slowing it down.

We found the following strategy to be useful: first, write the untyped version of the code, forgetting about the desired invariants as much as possible. Then, to add the extra type information, look for an inductive type which participates in the algorithm, and see if you can "hang" some new type indices off of it.

In our case, the injection index (distance to the next "interesting" polynomial) was simply stored as an ℕ, and the information we needed was the number of variables in the inner polynomial, and the number of variables in the outer. All of that is stored in the following proof of ≤:

```
data _≤_ (m : ℕ) : ℕ → Set where
  m≤m : m ≤ m
  ≤-s   : ∀ {n}
          → (m≤n : m ≤ n)
          → m ≤ suc n
```

A value of type $n \leq m$ mimics the inductive structure of the ℕ we were storing to represent the distance between n and m. We were able to take this analogy to the extreme: where we needed an equivalent of Ordering:

```
data Ordering : ℕ → ℕ → Set where
  less    : ∀ m k → Ordering m (suc (m + k))
  equal   : ∀ m   → Ordering m m
  greater : ∀ m k → Ordering (suc (m + k)) m
```

We were able to construct one, with transitivity replacing addition.

```
data ≤-Ordering {n : ℕ} : ∀ {i j}
                          → (i≤n : i ≤ n)
                          → (j≤n : j ≤ n)
                          → Set
  where
  ≤-lt  : ∀ {i j-1}
        → (i≤j-1 : i ≤ j-1)
        → (j≤n : suc j-1 ≤ n)
        → ≤-Ordering (≤-trans (≤-s i≤j-1) j≤n)
                     j≤n
  ≤-gt : ∀ {i-1 j}
        → (i≤n : suc i-1 ≤ n)
        → (j≤i-1 : j ≤ i-1)
        → ≤-Ordering i≤n
                     (≤-trans (≤-s j≤i-1) i≤n)
  ≤-eq : ∀ {i}
        → (i≤n : i ≤ n)
        → ≤-Ordering i≤n
                     i≤n

≤-compare : ∀ {i j n}
          → (x : i ≤ n)
          → (y : j ≤ n)
          → ≤-Ordering x y
≤-compare m≤m     m≤m     = ≤-eq m≤m
≤-compare m≤m     (≤-s y) = ≤-gt m≤m y
≤-compare (≤-s x) m≤m     = ≤-lt x m≤m
≤-compare (≤-s x) (≤-s y)
  with ≤-compare x y
... | ≤-lt i≤j-1 _ = ≤-lt i≤j-1 (≤-s y)
... | ≤-gt _ j≤i-1 = ≤-gt (≤-s x) j≤i-1
... | ≤-eq _       = ≤-eq (≤-s x)
```

4.4 Unification

So far, our optimizations have focused on the *operations* performed on the polynomial. Remember, though, the reflexive proof process has several steps: only one of them containing the operations (⟦_⇓⟧ in figure 3).

As it happens, we have now optimized these operations so much that they are no longer the bottleneck in the process. Surprisingly, the innocuous-looking refl now takes the bulk of the time! Typechecking this step involves unifying the two normalised expressions, a task which is quite expensive, with counterintuitive performance characteristics. So counterintuitive, in fact, that early versions of the solver, with all the optimizations from Grégoire and Mahboubi [2005] applied, was in many cases *slower* than the old, unoptimized solver!

In this section, we'll try and explain the problem and how we fixed it, and give general guidelines on how to write Agda code which typechecks quickly.

First, the good news. In the general case, unifying two expressions takes time proportional to the size of those expressions, so our hard-won optimizations do indeed help us.

Unfortunately, though, the "general case" isn't really that general: Agda's unification algorithm has a very important shortcut which we *must* make use of if we want our code to typecheck quickly: *syntactic equality*.

Because Agda is a dependently-typed language, types can contain functions, variables, and all sorts of complex expressions. One might expect that the unification algorithm should compute these expressions as far as it can, getting them to normal form, before it checks for any kind of equality. This would be disastrous for performance! Consider the following:

$$\mathsf{sum}\ [1..100] \stackrel{?}{=} \mathsf{sum}\ [1..100]$$

Running both computations here is an unnecessarily expensive task, and one which Agda does indeed avoid. Before the full unification algorithm, the typechecker does a quick pass to spot any syntactic equalities like the one above: if it sees one, it can avoid any more computation on that particular expression.

Crucially, missing syntactic equality on a term doesn't just hurt performance once: once syntactic equality fails, the next step is normalisation, which can change the shape of the entire subexpression, *destroying* any chance for syntactic equality later on. This can result in a cascade of missed syntactic equalities, causing a serious performance problem.

With that in mind, our optimisation will consist of two main strategies.

4.4.1 Avoiding Progress. First, we will consider something which may seem inconsequential: the order of arguments to the evaluation functions.

$$[\![\, xs \,]\!]\ \rho = \mathsf{foldr}\ (\lambda\ y\ ys \to \rho * ys + y)\ 0\ xs \qquad\qquad [\![\, xs \,]\!]\ \rho = \mathsf{foldr}\ (\lambda\ y\ ys \to y + ys * \rho)\ 0\ xs$$

The definition on the left is the one we've been working with so far. To a seasoned functional programmer, however, the version on the right might seem much more natural. By one measure, the version on the right is better! When applied to $x^2 + 2$, it gives a much smaller normal form:

$$x * (x * (x * 0 + 1) + 0) + 2 \qquad\qquad \mathsf{suc}\ (\mathsf{suc}\ ((x + 0) * x))$$

Surprisingly, this is exactly what we *don't* want! The left-hand-side equation above, though larger, *starts* with all of the variables, which must be trivially equal to the same variables in the other expression. Since the equality check proceeds left-to-right, this maintains that syntactic equality for as long as possible.

On the right hand side, however, we first examine the coefficients. These are computed from manipulations of the Horner Normal Form, and so are likely to be *not* syntactically equal. Even worse: since the expression *can* be normalised, we'll mess up its whole structure, ruining later syntactic checks!

4.4.2 Avoiding Identities.

4.5 Benchmarks

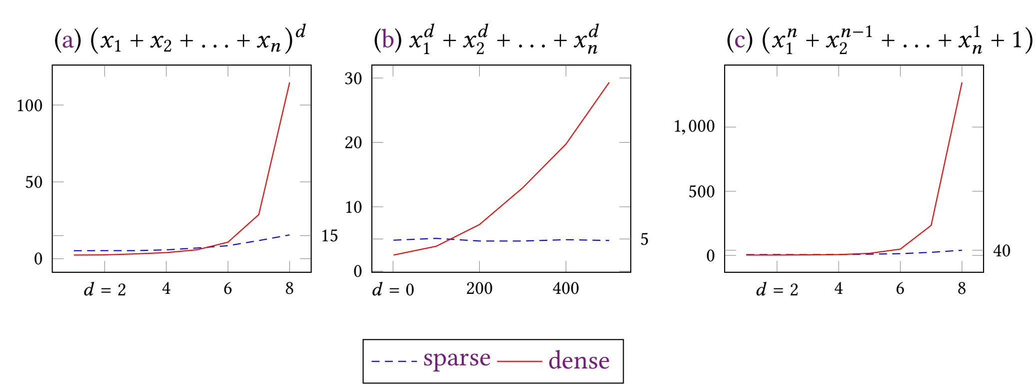

Fig. 4. Time (in seconds) to prove each expression is equal to its expanded form ($n = 5$ for each).

5 PEDAGOGICAL SOLUTIONS

One of the most widely-used and successful computer algebra systems, especially among non-programmers, is Wolfram|Alpha [Wolfram Research, Inc. 2019]. It can generate "pedagogical" (step-by-step) solutions to maths problems [The Development Team 2009]. For instance, given the input $x^2 + 5x + 6 = 0$, it will give the following output:

$$
\begin{aligned}
x^2 + 5x + 6 &= 0 \\
(x + 2)(x + 3) &= 0 \\
x + 2 = 0 \text{ or } x + 3 &= 0 \\
x = -2 \text{ or } x + 2 &= 0 \\
x = -2 \text{ or } x &= -3
\end{aligned}
$$

These tools can be invaluable for students learning basic mathematics. Unfortunately, much of the software capable of generating usable solutions is proprietary (including Wolfram Alpha), and little information is available as to their implementation techniques.[Lioubartsev 2016] is perhaps the best current work on the topic, but even so very little work exists in the way of the theoretical basis for pedagogical solutions.

Lioubartsev [2016] reformulates the problem as one of *pathfinding*. 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.

This approach suffers from 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. Unfortunately, what this should be is not clear: Lioubartsev [2016] uses a measure of the "simplicity" of an expression.

So, with an eye to using our solver instead of A*, we can notice that paths in undirected graphs form a perfectly reasonable equivalence relation: transitivity is the concatenation of paths, reflexivity is the empty path, and symmetry is *reversing* a path. Equivalence classes, in this analogy, are connected components of the graph.

More practically speaking, we implement these "paths" as lists, where the elements of the list are elementary ring axioms. When we want to display a step-by-step solution, we simply print out each element of the list in turn, interspersed with the states of the expression (the vertices in the graph).

```
module Trace {a}
             {A : Set a} where

  data _⋯_ (x : A) : A → Set a where
    [] : x ⋯ x
    _::_ : ∀ {y z}
         → String
         → y ⋯ z
         → x ⋯ z

  refl : Reflexive _⋯_
  refl = []

  trans : Transitive _⋯_
  trans [] ys = ys
  trans (x :: xs) ys = x :: trans xs ys

  sym : Symmetric _⋯_
  sym = go []
    where
    go : ∀ {x y z} → y ⋯ z → y ⋯ x → x ⋯ z
    go xs [] = xs
    go xs (y :: ys) =
      go (("sym(" ++ y ++ ")") :: xs) ys
```

If we stopped there, however, the solver would output incredibly verbose "solutions": far too verbose to be human-readable. Instead, we must apply a number of heuristics to cut down on the solution length:

(1) First, 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.
(2) Next, we remove any “step, reverse-step” chains. Since we’re converting expressions to normal form, the path may be hourglass-shaped. Fig. 5 is the output from our solver without this heuristic applied.
    The problem is that both expressions hit a common form early on, at $x + y + 3$. Nonetheless, the proof will soldier on, giving superfluous steps.
    These sections are what we want to detect, and remove.

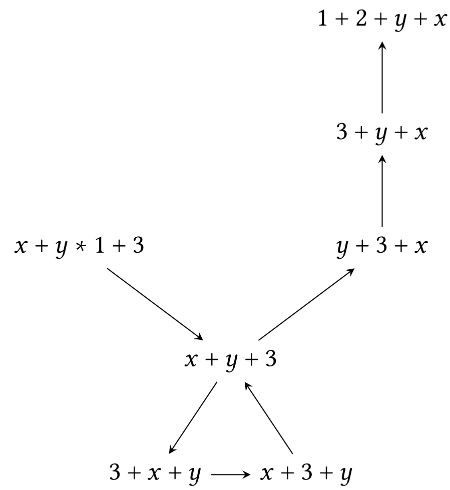

Fig. 5. Hourglass-Shaped Graph

After applying those heuristics, our solver outputs the following for the lemma in Fig. 1b:

```
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
```

Figuring out good heuristics and path compression techniques seems to deserve further examination.

6 RELATED WORK

In dependently-typed programming languages, the state-of-the-art solver for polynomial equalities (over commutative rings) was originally presented in Grégoire and Mahboubi [2005], and is used in Coq’s `ring` solver. This work improved on the already existing solver [Coq Development Team 2002] 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 Boutin [1997]).

Agda [Norell and Chapman 2008] is a dependently-typed programming language based on Martin-Löf’s Intuitionistic Type Theory [Martin-Löf 1980]. Its standard library [Danielsson 2018] 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 to the one presented here.

In Slama and Brady [2017], an implementation of an automated solver for the dependently-typed language Idris [Brady 2013] is described. The solver is implemented with a “correct-by-construction” approach, in contrast to Grégoire and Mahboubi [2005]. The solver is defined over *non*commutative rings, meaning that it is more general (can work with more types) but less powerful (meaning it can prove fewer identities). It provides a reflection-based interface, but internally uses a dense 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 Christiansen [2015], and specific to Agda in van der Walt [2012].

Formalization of mathematics in general is an ongoing project. Wiedijk [2018] tracks how much of "The 100 Greatest Theorems" [Kahl 2004] have so far been formalized (at time of writing, the number stands at 93). DoCon [Meshveliani 2018] is a notable Agda library in this regard: it contains many tools for basic maths, and implementations of several CAS algorithms. Its implementation is described in Meshveliani [2013]. Cheng et al. [2018] describes the manipulation of polynomials in both Haskell and Agda.

Finally, the study of *pedagogical* CASs which provide step-by-step solutions is explored in Lioubartsev [2016]. One of the most well-known such system is Wolfram Alpha [Wolfram Research, Inc. 2019], which has step-by-step solutions [The Development Team 2009].

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.

Edwin Brady. 2013. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 05 (Sept. 2013), 552–593. https://doi.org/10.1017/S095679681300018X

Chen-Mou Cheng, Ruey-Lin Hsu, and Shin-Cheng Mu. 2018. Functional Pearl: Folding Polynomials of Polynomials. In *Functional and Logic Programming (Lecture Notes in Computer Science)*. Springer, Cham, 68–83. https://doi.org/10.1007/978-3-319-90686-7_5

David Raymond Christiansen. 2015. *Practical Reflection and Metaprogramming for Dependent Types*. Ph.D. Dissertation. IT University of Copenhagen. http://davidchristiansen.dk/david-christiansen-phd.pdf

The Coq Development Team. 2002. *The Coq Proof Assistant Reference Manual, Version 7.2*. http://coq.inria.fr

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

Nathan W. Kahl. 2004. The Hundred Greatest Theorems. http://web.archive.org/web/20080105074243/http://personal.stevens.edu/~nkahl/Top100Theorems.html

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.pdf

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

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

Sergei D Meshveliani. 2013. *Dependent Types for an Adequate Programming of Algebra*. Technical Report. Program Systems Institute of Russian Academy of sciences, Pereslavl-Zalessky, Russia. 15 pages. http://ceur-ws.org/Vol-1010/paper-05.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. 2018. Agda-Prelude: Programming Library for Agda. https://github.com/UlfNorell/agda-prelude

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

Franck Slama and Edwin Brady. 2017. Automatically Proving Equivalence by Type-Safe Reflection. In *Intelligent Computer Mathematics*, Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke (Eds.). Vol. 10383. Springer International Publishing, Cham, 40–55. https://doi.org/10.1007/978-3-319-62075-6_4

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/

Paul van der Walt and Wouter Swierstra. 2013. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Vol. 8241. Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173. https://doi.org/10.1007/978-3-642-41582-1_10

P. D. van der Walt. 2012. *Reflection in Agda*. Master's Thesis. Universiteit of Utrecht. https://dspace.library.uu.nl/handle/1874/256628

Freek Wiedijk. 2018. Formalizing 100 Theorems. http://www.cs.ru.nl/~freek/100/
Wolfram Research, Inc. 2019. Wolfram|Alpha. Wolfram Research, Inc.. https://www.wolframalpha.com/

A APPENDIX

Text of appendix …