Reading and Writing Arithmetic: Automating Ring Equalities in Agda

Donnacha Oisín Kidney

[1] University College Cork
[2] `115702295@umail.ucc.ie`

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 [15]. It is significantly faster than Agda's existing solver. We use reflection to provide a simple interface to the solver, and demonstrate how to use the constructed proofs to provide step-by-step solutions.

1 Introduction

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

This need not be the case! With the right components and libraries, we believe that Agda can be a tool for mathematics that's not only useful, but also easy, friendly, and fun. This work describes one such library: a solver which automates the construction of proofs like Fig. **??**, making them as easy as Fig. **??**.

The main contributions of our library are as follows:

Ease of Use Proofs like the one in Fig. **??** are long, difficult to write, and uninteresting. Our solver, in contrast, is extremely simple to use: the single line in Fig. **??** solves the lemma.
This interface (section 4) is implemented using a lightweight reflection system (section 4.1), which doesn't require the user to write *any* reflection code, even if they use the solver with their own custom type.

Performance Our solver is significantly faster than Agda's current ring solver, cutting type-checking time down from minutes to seconds in several use cases (section 5.3).
As described in [8], we use a sparse internal representation of polynomials (section 5.1). However, because of differences between Agda and Coq's type checker, we found that this optimisation—on its own—did not deliver a significant speedup, and actually damaged performance in a number of cases. Achieving the performance we did required an entirely separate kind of optimisation, described in section 5.2.

Pedagogical Solutions Computer algebra systems (CASs) outside the rigorous world of dependently-typed languages do far more than just check proofs

for mistakes: they have a wealth of other features which can help with learning mathematics as well as verifying it.
We hope that similar systems developed in Agda can do the same: as a demonstration, we implement "step-by-step solutions", one of the most popular features of modern CASs. Far from being ill-suited to Agda, we show that the constructive nature of our proofs allows for a natural implementation (section 7).

2 Scope

In this section, we will explain the intended uses and necessary limitations of the solver.

Equivalences First, an inflexibility: the solver deals very specifically with the domain of *equivalence* proofs, like the one in Fig. **??**. While it may be of use in other settings (finding roots, etc.), that is not explored here.

Setoids On the other hand, we are very flexible about what kind of "equivalence" we prove. In fact, the solver will work with any equivalence relation, as long as it comes with proofs of the relevant ring axioms. This is useful for all of the usual things (approximating quotients and so on), but it also provides the basis for our "step-by-step solutions" implementation in section 7.

Almost Rings As in [8, section 5], we use a peculiar algebraic structure which lies somewhere between a semiring and a ring. These "almost-rings" have all of the usual laws of a commutative ring, but instead of demanding additive inverses, they require the comparatively permissive "pseudo-inverse" operation, which obeys the following equations:

$$-(x * y) = -x * y \tag{1}$$

$$-(x + y) = -x + -y \tag{2}$$

This allows the solver to work on types which don't have an additive inverse (like $\mathbb{N}$): such types just supply the identity function instead of negation, and the two laws above are satisfied.
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.

Weak Decidability A core optimisation in our solver (section 5.1) relies on the ability to test arbitrary coefficients for zero. Instead of requiring decidable equality (which would greatly diminish the number of types the solver can work with), we instead ask for weakly decidable equivalence with zero:

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

Just as in Agda's current solver, this allows users to avail of the optimisation if their type supports it, or skip it (`is-zero = const nothing`) if not.

Correctness The nature of the solver means it is intrinsically sound (i.e. it cannot prove an equivalence unless there is one): since all it does is rearrange and join together the ring axioms, it cannot prove anything that doesn't derive from them. We have not, however, proven completeness (that every equivalence will be found by our solver).
In the internal representation of the solver, we prove several data structure invariants (like sparsity) intrinsically.
The reflection-based interface is unproven, but since the output is type checked our claim of soundness still stands: a bug in our reflection code can only cause the solver to miss a solution, never to prove something it shouldn't.

3 Overview of the Proof Technique

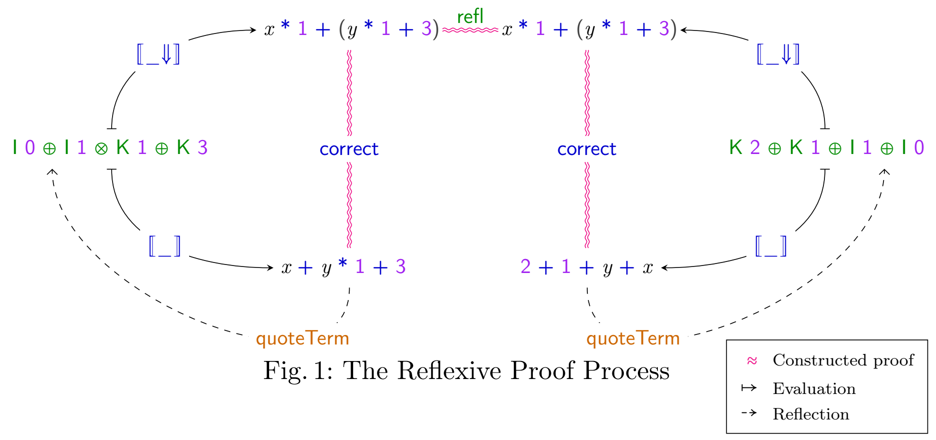


Fig. 1: The Reflexive Proof Process

Before diving into to specifics, we'll first give a quick overview of how the solver works, so it's clear how the bits of implementation described later in the paper fit together.

The technique we use for automating equivalence proofs comes from [1]: the general idea is that we prove two expressions equivalent by proving that they're both equivalent to the same canonical form. The diagram in Fig. 1 demonstrates this for the identity from Fig. **??**: on the bottom of the diagram you can see the left and right hand side of the identity we want to prove, and on the top we can see their normal forms. The actual proof the solver provides is represented by the ≈ path.

```
data Expr {ℓ} (A : Set ℓ) (n : ℕ) : Set ℓ where
  K     : A → Expr A n
  I     : Fin n → Expr A n
  _⊕_  : Expr A n → Expr A n → Expr A n
  _⊗_  : Expr A n → Expr A n → Expr A n
  _⊛_  : Expr A n → ℕ → Expr A n
  ⊝_   : Expr A n → Expr A n
```

Fig. 2: A Type for Ring Expressions

To prove that each expression is equivalent to the canonical form we first represent the expressions using the type in Fig. 2. This type represents the Abstract Syntax Tree (AST) for expressions in the almost-ring algebra: it has constructors corresponding to each ring operation ($x+y = x \oplus y$, $x*y = x \otimes y$), and it can refer to variables via their de Bruijn index (so x becomes I 0).

There are two ways to evaluate the AST: the ⟦_⟧ function converts the AST to the expression we want to prove, whereas ⟦_⇓⟧ converts it to a the canonical form. The implementation of the ⟦_⇓⟧ function is described in section 5.1.

Equivalence of the canonical forms is proven via correct: some of the details of this are explained in section 6.

Finally, instead of asking the user to construct the AST themselves, we use reflection to automate it. This is described in the following section.

4 The Interface

We felt an easy-to-use interface was one of the most important components of the library as a whole. Since we wanted to minimise the amount a user would have to learn to use the solver, we kept the surface area of the library quite small: aside from the almost-ring type, 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. 4: The solveOver Macro

solve is demonstrated in Fig. ??. 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 (Fig. 4). As well as the algebra 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:

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

– We don't ask the user to construct the Expr AST which represents their proof obligation. Compare this to Fig. 3: 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) recognises 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 [19,**?**] it is still not fully automatic.

4.1 Implementation

Agda has powerful metaprogramming facilities, which allow programs to manipulate their own code. Here, we'll use reflection to implement the interface to our solver.

Agda's reflection API is mostly encapsulated by the following three types:

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: the Term type itself doesn't contain a whole lot of type information, and it's quite fragile and sensitive to context. 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.

Building a robust interface using reflection required a great deal of care. To demonstrate some of the techniques we used, we'll look at two functions from the core of the interface. First, toExpr:

```
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 equivalence. It converts it to the corresponding Expr. In other words, it performs the following transformation:

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

When it encounters one of the ring operators, it calls the corresponding helper function (getBinOp, getExp, or getUnOp) which finds the important subterms from the operator's argument list.

If it *doesn't* manage to match an operator or a variable, it assumes that what it has must be a constant, and wraps it up in the K constructor. This is the key trick which allows us to avoid ever asking the user to quote their own type. While it may seem unsafe at first glance, we actually found it to be more robust (for our use case) than the alternative:

Principle 1 (Don't reimplement the typechecker) *While it may seem good and fastidious to rigorously check the structure and types of arguments given to a macro, we found better results by avoiding validity-checking in metaprogramming code. Instead, we preferred to proceed as if there were no errors (if possible), but arrange the output so that the user would still see a type error where the input was incorrect.*

Taking this case as an example, if the user indeed manages to supply something other than the correct type, Agda will catch the error, as an incorrect argument to K.

If, on the other hand, we had asked the user to quote their own type, we would have trouble handling (for instance) closed applications of functions, references to names outside the lambda, etc. This approach, on the other hand, has no such difficulty.

Next, we'll look at one of the helper functions: getExp, which deals with exponentiation.

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

It extracts the last two arguments to the exponentiation operator, and wraps them up with the ⊛. Before the two visible arguments to the exponentiation operator, we first apply 3 ⋯⟅:⟆. This applies three hidden arguments as "unknown", i.e. asks Agda to infer them. We could guess them ourselves: the first is the universe level of the carrier type, the second is the carrier type, and the third is the number of variables in the expression. We decided against it, though, instead being intentionally unspecific:

Principle 2 (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. We found this approach to be error-prone, however: as it turns out, the compiler is better a guessing implicits than we are. Instead, we preferred to* leverage *the compiler, relying on inference over direct metaprogramming as much as possible.*

The final point to make is that the entire interface implementation is itself quite small (fewer than 100 lines). This isn't because our code was terse: rather, we intentionally minimised the amount of metaprogramming we did.

Principle 3 (Keep Metaprogramming to the Edges) *With great power comes 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.*

4.2 Maintaining Invariants

One obvious benefit of reflection is a terse interface. However, we feel that another benefit—resilience to change—is just as important. This section illustrates that resilience with an example.

Agda allows us to encode program correctness in types, so we can *prove* properties we would have otherwise only been able to test. Unfortunately, these kinds of proofs tend to be very tightly coupled to the implementation of the algorithms they verify. This can make iteration difficult, where small optimisations or bug fixes can invalidate proofs for other invariants.

To demonstrate the problem, and how our solver can reduce some of the burden, we'll look at size-indexed binary trees:

```
data Tree : ℕ → Set a where
  leaf : Tree 0
  node : ∀ {n m} → A → Tree n → Tree m → Tree (n + m + 1)
```

We've deliberately chosen an awkward type here: in contrast to the more common size-indexed lists, the index (the size) doesn't match the shape of the data structure. As a result, almost every function which manipulates the tree in some way will have to come accompanied by a verbose, complex proof. Take this line, for instance, which performs a left-rotation on the tree:

$$\mathsf{rot}^{\mathsf{l}}\ (\mathsf{node}\ \{a\}\ x\ xl\ (\mathsf{node}\ \{b\}\ \{c\}\ y\ yr\ yl)) = \mathsf{node}\ y\ (\mathsf{node}\ x\ xl\ yl)\ yr$$

A sensible invariant to encode here is that the function doesn't change the size of the tree. Unfortunately, to *prove* that invariant, we have to prove the following:

$$1 + (1 + a + c) + b = 1 + a + (1 + b + c)$$

Though simple, this is precisely the kind of proof which requires many fussy applications of the ring axioms. Here, our solver can help:

$$\begin{aligned}&\mathsf{rot}^{\mathsf{l}}\ (\mathsf{node}\ \{a\}\ x\ xl\ (\mathsf{node}\ \{b\}\ \{c\}\ y\ yr\ yl)) = \mathsf{node}\ y\ (\mathsf{node}\ x\ xl\ yl)\ yr\\ &\quad \Rightarrow \forall\langle\ a :: b :: c :: []\ \rangle\end{aligned}$$

While cutting down on the amount of code we need to write is always a good thing, the real strength of this method is that it automatically infers the input type. This makes it resilient to small changes in the code. So, when we notice the bug in the code above (yl and yr are swapped in the pattern-match), we can simply *fix it*, without having to touch any of the proof code.

$$\begin{aligned}&\mathsf{rot}^{\mathsf{l}}\ (\mathsf{node}\ \{a\}\ x\ xl\ (\mathsf{node}\ \{b\}\ \{c\}\ y\ yl\ yr)) = \mathsf{node}\ y\ (\mathsf{node}\ x\ xl\ yl)\ yr\\ &\quad \Rightarrow \forall\langle\ a :: b :: c :: []\ \rangle\end{aligned}$$

If we hadn't used the solver, this fix would have necessitated a totally new proof. By automating the proof, we allow the compiler to automatically check what we *mean* ("does the size of the tree stay the same?"), while we worry about other details.

5 Performance

Our solver is significantly faster than the current solver: the following sections will detail how we achieved that speedup. We will start by describing the naive implementation; in section 5.1 we demonstrate how we added the optimisations from [8]; and in section 5.2 we will describe the Agda-specific optimisations which account for the bulk of our speedup. Finally, section 5.3 contains some benchmarks against the current solver.

5.1 Normalisation

Most of code written for the solver is concerned with normalisation: the $[\![\_\Downarrow]\!]$ function in Fig. 1. This converts from the expression AST (Fig. 2) to a canonical form.

Horner Normal Form The particular "canonical form" we'll start with is the same as in Agda's current ring solver: 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:

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

Is represented by this list:

```
3 : 0 : 2 : 0 : 0 : 4 : 0 : 2 : []
```

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

```
_⊞_ : 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) []
```

So to get from Expr to Poly we map each constructor to the relevant polynomial operation. Then, to get from Poly to an expression in the underlying ring, we use Horner's rule: a classic example of the foldr function.

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

Sparse Encodings Our first avenue for optimisation comes from [8]. Notice that the encoding above is quite wasteful: it always stores an entry for each coefficient, even if it's zero. In practice, we're likely to often find long strings of zeroes (in expressions like x^{10}), meaning that our representation will contain long "gaps" between the coefficients we're actually interested in (non-zero ones).

To fix the problem we'll switch to a *sparse* encoding, by storing a "power index" with every coefficient. This will represent the size of the gap from the previous non-zero coefficient. Taking 3 again as an example, we would now represent it as follows:

```
(3 , 0) : (2 , 1) : (4 , 2) : (2 , 1) : []
```

Next, we turn our attention to the task of adding multiple variables. Luckily, there's an easy way to do it: nesting. Multivariate polynomials will be represented as "polynomials of polynomials", where each level of nesting corresponds to one variable. It's perhaps more clearly expressible in types:

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

Inductively speaking, a "polynomial" in 0 variables is simply a constant, whereas a polynomial in n variables is a list of coefficients, which are themselves polynomials in $n-1$ variables.

Before running off to use this representation, though, we should notice that we have created another kind of "gap" which we should avoid with a sparse encoding. For a polynomial with n variables, we will always have n levels of nesting, even if the polynomial doesn't actually refer to all n variables. In the extreme case, representing the constant 6 in a polynomial of 3 variables looks like the following:

```
((((((6 , 0) : []) , 0) : []) , 0) : [])
```

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". In contrast to the previous index, though, this one is type-relevant: we can't just store a ℕ next to the nested polynomial to represent the gap. Because the polynomial is indexed by the number of variables it contains, any encoding of the gap will have provide the proper type information to respect that index.

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.

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 $\leq$:

```
data _≤_ (m : ℕ) : ℕ → Set where
  m≤m : m ≤ m
  ≤-s   : ∀ {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 quite far: in a few functions, for instance, we needed to compare these gaps. By mimicking the inductive structure of ℕ, we were able to directly translate Ordering and compare on ℕ:

```
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
```

into equivalent functions on ≤:

```
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)
```

5.2 Unification

After applying the previous optimisations, we might expect an immediate speedup in the solver: unfortunately, this isn't the case. Without some careful adjustments, the optimisations in the previous section can actually *slow down* the 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.

Up until now, we 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 Fig. 1). Despite having the most complex implementation, this isn't the most expensive step: surprisingly, the innocuous-looking refl 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.

First, the good news. In the general case, unifying two expressions takes time proportional to the size of those expressions, so our hard-won optimisations 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*.

Before the full unification algorithm, Agda runs a quick check to see if the two expressions it's testing for equality are *syntactically* equal. This can make a big difference in unification problems like the following:

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

By noticing that these expressions are syntactically equal, we can avoid actually computing the sum function. Taking advantage of that shortcut is key to achieving decent performance. With that in mind, there are two main strategies we'll use to encourage syntactic equality:

Avoid Progress at all Costs First, we will consider something which may seem inconsequential: the order of arguments to the evaluation functions.

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

$[\![\_]\!]_{\mathsf{l}}$ is the definition we've been working with so far. Some readers might find $[\![\_]\!]_{\mathsf{r}}$ more natural, however. The reason is that it's more productive: in lazy languages, the usual convention is that functions which take multiple arguments should scrutinise those arguments from left to right. The $*$ and $+$ functions (on $\mathbb{N}$, at any rate) follow that convention, meaning that $[\![\_]\!]_{\mathsf{r}}$ is able to make more progress without a concrete x. Taking the polynomial $x^2 + 2$ as an example:

$$\begin{array}{ll} [\![\ 2 : 0 : 1 : [\,]\ ]\!]_{\mathsf{l}}\ x & \equiv\langle\rangle \\ x * (x * (x * 0 + 1) + 0) + 2 & \equiv\langle\rangle \\ x * (x * (x * 0 + 1) + 0) + 2 & \blacksquare \end{array} \qquad \begin{array}{ll} [\![\ 2 : 0 : 1 : [\,]\ ]\!]_{\mathsf{r}}\ x & \equiv\langle\rangle \\ 2 + (0 + (1 + 0 * x) * x) * x & \equiv\langle\rangle \\ \mathsf{suc}\ (\mathsf{suc}\ ((x + 0) * x)) & \blacksquare \end{array}$$

In $[\![\_]\!]_{\mathsf{l}}$, we're blocked pretty much straight away, as x is the first thing we try to scrutinise. In $[\![\_]\!]_{\mathsf{r}}$, since all of the constants are kept to the left, they're scrutinised first, allowing us to perform much more normalisation before being blocked.

This is exactly what you *don't* want! Since both expressions will be coming out of the same evaluation function, they should have the same structure, meaning that we don't *need* the reduction of outer terms that $[\![\_]\!]_{\mathsf{r}}$ gives us. We only need to perform normalisation on the coefficients: these are computed during the manipulations of the polynomial, and so may contain unevaluated expressions. If we used $[\![\_]\!]_{\mathsf{r}}$ as our definition, then the type checker will likely hit an inequality on the *first* term, and as a result we lose all opportunity for syntactic equality. $[\![\_]\!]_{\mathsf{l}}$, on the other hand, front-loads all of the variables, maintaining syntactic equality for as long as possible.

As well as that, we don't have any control of the structure we get from the ring operators. This means that any reduction, as well as being unnecessary, can destroy the structural similarity between the two expressions, and as a result their syntactic equality.

The (counterintuitive) lesson learned is as follows: to speed up unification, keep things which are likely to be syntactically equal to the left, and *don't* structure your functions to encourage progress. Simply swapping the arguments (as we do above) resulted in a performance improvement of several orders of magnitude.

Avoid Identities It's a good idea to avoid identities (expressions like $0 + x$ or $1 * x$) in the normalised expression. This will reduce the size of your expression, which is helpful in general, but more importantly it increases the likelihood of finding syntactic equality in the argument to the identity (x in the examples above).

Our sparse representation helps significantly in this case by entirely removing 0 from the generated expression. Another place we can make improvements is in the base cases for recursive functions. Take exponentiation, for example:

```
_^_ : Carrier → ℕ → Carrier
x ^ zero  = 1
x ^ suc i = x * (x ^ i)
```

We can avoid that 1 in the majority of cases by rewriting the function to have an extra base case:

```
_^_+1 : Carrier → ℕ → Carrier
x ^ zero  +1 = x
x ^ suc i +1 = (x ^ i +1) * x
```

```
_^_ : Carrier → ℕ → Carrier
x ^ zero  = 1
x ^ suc i = x ^ i +1
```

In the library, we employ this idea extensively, avoiding unnecessary identities as much as we could. This has a significant effect on the size of the resulting normal form, but also ensures that normalisation stops exactly where we want it to, preserving the structure of the expressions as much as is possible. This makes a significant difference to both size and syntactic similarity as can be seen in Fig. 5.

5.3 Benchmarks

As expected, the new implementation exhibits a significant speedup in type-checking over the old implementation, even with the added overhead of the reflection-based interface. Fig. 7 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 new representation is clearly faster overall, with a factor of 7.5 speedup for $d = 8$.

Fig. 9 demonstrates perhaps a more common use-case, with a mix of high and low powers and some constants. The new representation's advantage is even more apparent here, with an 30-factor speedup at $d = 8$.

The old solver does exhibit a small lead (roughly 2-3 seconds, which narrows to about 1 second without reflection) on very simple expressions, possibly caused by the overhead of the new solver's more complex implementation. 3 seconds is quite small in the context of Agda type checking (the standard library, for instance, takes several minutes to type check), so we feel this slight loss is more than made up for by the gains. Furthermore, we have not been able to find a case where the old solver is significantly faster than the new. Nonetheless, if a user really wants to use the old solver, the other components described here are entirely modular, and can work with any underlying solver which uses the reflexive technique.

6 Verification

The output of the solver is a constructive proof of equivalence: this is *derived* from a generic proof that the operations on the solver are a ring homomorphism from the carrier type. Put another way, for the solver to work properly, we would need to prove that addition (and multiplication, and negation, etc.) on Horner normal forms corresponds with addition on the carrier type.

These proofs are long (about 1000 lines) and complex. Without careful structuring of the proofs, every new optimisation would require a whole new round of proof code, with very little reuse.

To avoid this problem, we took inspiration from [14], and relied heavily on abstraction and folds to improve the reuse in proof code. In particular, we defined many operations as *metamorphisms* [7]. So, instead of defining (say) negation over the polynomial type itself, we will define a metamorphism to express negation, and then call some higher-order function to run that metamorphism over a polynomial.

```
Meta : ℕ → Set c
Meta n = Poly n × Coeff n ⋆ → Poly n × Coeff n ⋆
```

From here, we can define the *semantics* of a metamorphism. As an example, Fig. 10 shows the semantics of poly-map, a simple morphism which behaves something like map on lists.

Now, each operation only has to be proven up to the semantics defined above. Crucially, optimisations like the sparse encoding *respect* these semantics, so we only have to change our proof in one place: the definition of poly-mapR.

```
poly-mapR
  : ∀ {n} ρ ρs
  → ([f] : Poly n → Poly n)
  → (f : Carrier → Carrier)
  → (∀ x y → x * f y ≈ f (x * y))
  → (∀ x y → f (x + y) ≈ f x + f y)
  → (∀ y → ⟦ [f] y ⟧ ρs ≈ f (⟦ y ⟧ ρs))
  → (f 0# ≈ 0#)
  → ∀ xs
  → Σ?⟦ poly-map [f] xs ⟧ (ρ , ρs)
       ≈ f (Σ⟦ xs ⟧ (ρ , ρs))
```

Fig. 10: The Semantics of poly-map

7 Pedagogical Solutions

One of the core aims of this work is to take a step towards making Agda a useful tool for doing mathematics. The rest of this paper has described our efforts to compensate for Agda's disadvantage in this area: namely, a pedantic typechecker. This section will attempt to show the other side of the coin, and demonstrate some of the unique benefits that come from using a programming language to do your proofs.

Outside of computer scientists and mathematicians, most people's experience of computer algebra probably amounts to the step-by-step solutions from Wolfram|Alpha [17] or some similar system.

Something so high-level and user-facing hardly seems like it's in a dependently-typed language's wheelhouse; on the other hand, the very nature of a proof in Agda is that it has computational content: why not make some of that content an explanation for the equality?

Prior work in this area includes [10]: there, the problem is reformulated reformulates the problem 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: [10] uses a measure of the "simplicity" of an expression.

Notice, however, 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.

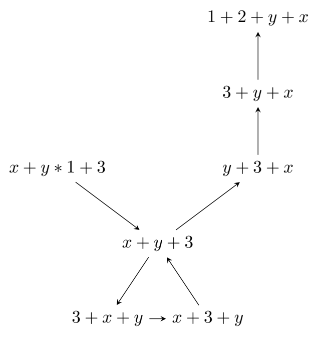


Fig. 11: Graph Containing Loops

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).

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 path-compression heuristics to cut down on the solution length:

1. First, we remove loops from the graph. Fig 11 shows an example solution without this heuristic applied: it crosses the same point multiple times, creating useless steps in the output. 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 the explanation in Fig. 12 for the lemma in Fig. ??

8 Related Work

In dependently-typed programming languages, the state-of-the-art solver for polynomial equalities (over commutative rings) was originally presented in [8], 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 [15] is a dependently-typed programming language based on Martin-Löf's Intuitionistic Type Theory [11]. 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 to the one presented here.

In [16], an implementation of an automated solver for the dependently-typed language Idris [2] is described. The solver is implemented with a "correct-by-construction" approach, in contrast to [8]. 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 [4], and specific to Agda in [18].

Formalisation of mathematics in general is an ongoing project. [20] tracks how much of "The 100 Greatest Theorems" [9] have so far been formalised (at time of writing, the number stands at 93). DoCon [13] 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 [12]. [3] 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 [10]. One of the most well-known such system is Wolfram Alpha [21], which has step-by-step solutions [17].

9 Conclusion

We have presented an efficient, flexible, and easy-to-use ring solver in the programming language Agda. In doing so, we hope we have removed one of the small but significant pain points of doing maths in Agda.

References

1. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) Theoretical Aspects of Computer Software. pp. 515–529. Lecture Notes in Computer Science, Springer Berlin Heidelberg (1997)

2. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(05), 552–593 (Sep 2013). DOI: 10.1017/S095679681300018X, http://journals.cambridge.org/article_S095679681300018X
3. Cheng, C.M., Hsu, R.L., Mu, S.C.: Functional Pearl: Folding Polynomials of Polynomials. In: Functional and Logic Programming. pp. 68–83. Lecture Notes in Computer Science, Springer, Cham (May 2018). DOI: 10.1007/978-3-319-90686-7_5, https://link.springer.com/chapter/10.1007/978-3-319-90686-7_5
4. Christiansen, D.R.: Practical Reflection and Metaprogramming for Dependent Types. Ph.D. thesis, IT University of Copenhagen (Nov 2015), http://davidchristiansen.dk/david-christiansen-phd.pdf
5. Coq Development Team, T.: The Coq Proof Assistant Reference Manual, Version 7.2 (2002), http://coq.inria.fr
6. Danielsson, N.A.: The Agda standard library (Jun 2018), https://agda.github.io/agda-stdlib/README.html
7. Gibbons, J.: Metamorphisms: Streaming Representation-Changers. Science of Computer Programming **65**(2), 108–139 (2007). DOI: 10.1016/j.scico.2006.01.006, http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/metamorphisms-scp.pdf
8. Grégoire, B., Mahboubi, A.: Proving Equalities in a Commutative Ring Done Right in Coq. In: Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 3603, pp. 98–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI: 10.1007/11541868_7, http://link.springer.com/10.1007/11541868_7
9. Kahl, N.W.: The Hundred Greatest Theorems (2004), http://web.archive.org/web/20080105074243/http://personal.stevens.edu/~nkahl/Top100Theorems.html
10. Lioubartsev, D.: Constructing a Computer Algebra System Capable of Generating Pedagogical Step-by-Step Solutions. Ph.D. thesis, KTH Royal Institue of Technology, Stockholm, Sweden (2016), http://www.diva-portal.se/smash/get/diva2:945222/FULLTEXT01.pdf
11. Martin-Löf, P.: Intuitionistic Type Theory. Padua (Jun 1980), http://www.cse.chalmers.se/~peterd/papers/MartinL%00f6f1984.pdf
12. Meshveliani, S.D.: Dependent Types for an Adequate Programming of Algebra. Tech. rep., Program Systems Institute of Russian Academy of sciences, Pereslavl-Zalessky, Russia (2013), http://ceur-ws.org/Vol-1010/paper-05.pdf
13. Meshveliani, S.D.: DoCon-A a Provable Algebraic Domain Constructor (Apr 2018), http://www.botik.ru/pub/local/Mechveliani/docon-A/2.02/
14. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation. Journal of Functional Programming **19**(5), 545–579 (Sep 2009). DOI: 10.1017/S0956796809007345, https://github.com/scmu/aopa
15. Norell, U., Chapman, J.: Dependently Typed Programming in Agda. Tech. rep. (2008), http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf
16. Slama, F., Brady, E.: Automatically Proving Equivalence by Type-Safe Reflection. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics, vol. 10383, pp. 40–55. Springer International Publishing, Cham (2017). DOI: 10.1007/978-3-319-62075-6_4, http://link.springer.com/10.1007/978-3-319-62075-6_4
17. The Development Team: Step-by-Step Math (Dec 2009), http://blog.wolframalpha.com/2009/12/01/step-by-step-math/

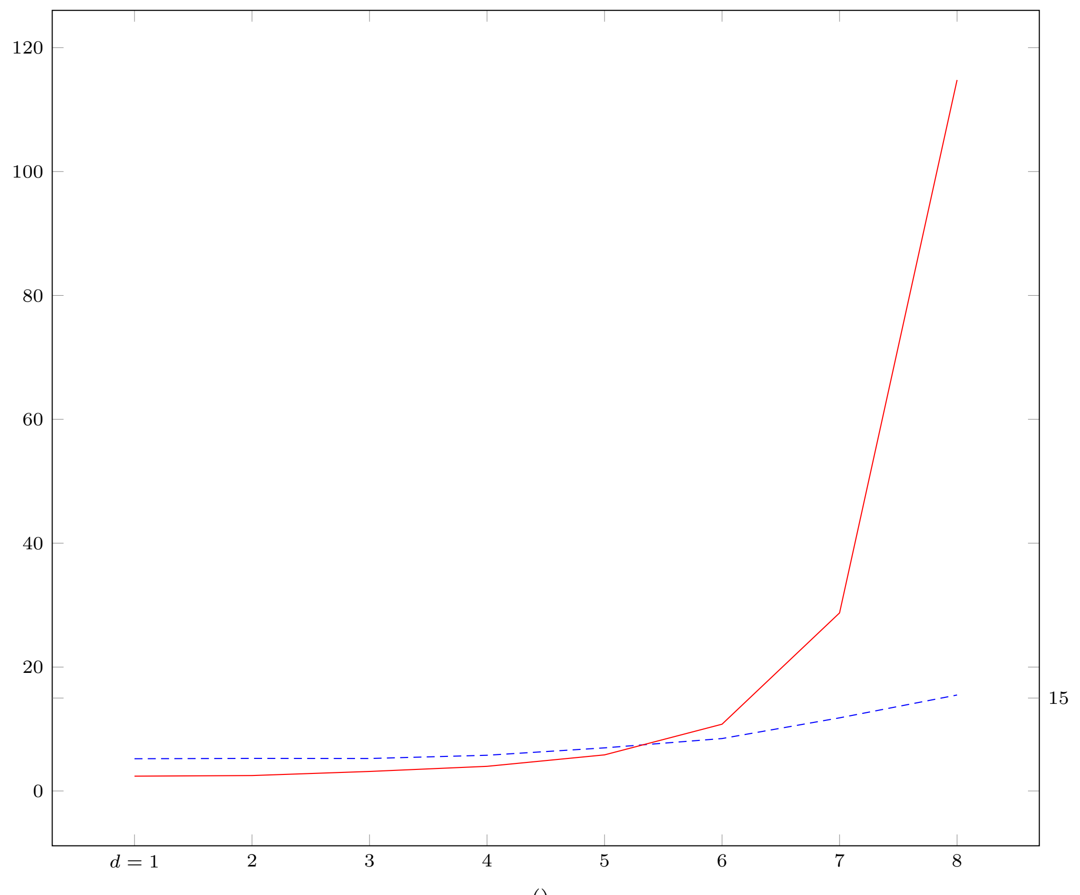
18. van der Walt, P.D.: Reflection in Agda. Master's Thesis, Universiteit of Utrecht (Oct 2012), https://dspace.library.uu.nl/handle/1874/256628
19. van der Walt, P., Swierstra, W.: Engineering Proof by Reflection in Agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages, vol. 8241, pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI: 10.1007/978-3-642-41582-1_10, http://link.springer.com/10.1007/978-3-642-41582-1_10
20. Wiedijk, F.: Formalizing 100 Theorems (Oct 2018), http://www.cs.ru.nl/~freek/100/
21. Wolfram Research, Inc.: Wolfram|Alpha. Wolfram Research, Inc. (2019), https://www.wolframalpha.com/

```
x *
  (x * (x *
    ((x * x) * (x *
      (x * 2)
    + 4))
  + 2))
+ 3
```

```
x *
  (x *
    (x *
      (x *
        (x *
          (x *
            (x *
              (x *
                   0
              + 2)
            + 0)
          + 4)
        + 0)
      + 0)
    + 2)
  + 0)
+ 3
```

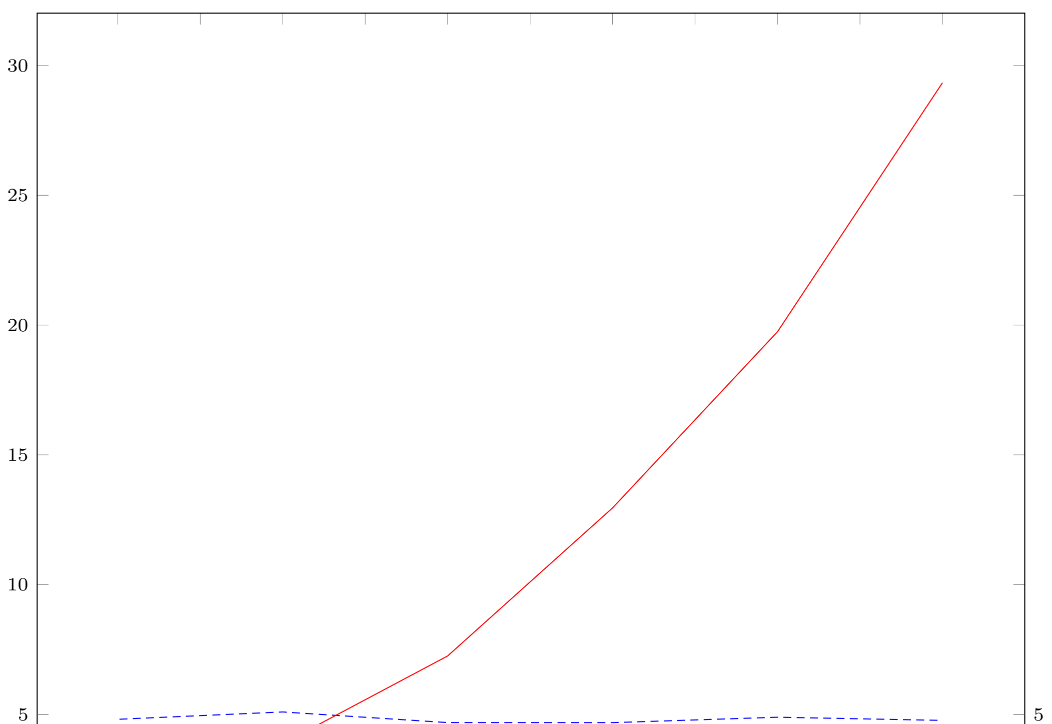
Fig. 5: Comparison of the normal forms of equation 3

(()) $(x_1 + x_2 + \ldots + x_n)^d$



()

(()) $x_1^d + x_2^d + \ldots + x_n^d$



```
x + y + 3
    ={ +-comm(x, y + 3) }
y + 3 + x
    ={ +-comm(y, 3) }
3 + y + x
```

Fig. 12: Step-by-Step Output From Our Solver