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 significantly 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. 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 proofs like Fig. 1a, making them as easy as Fig. 1b.

While correctness is, of course, an essential feature of any library like ours, it's not the whole story. For this work, we also felt that it was important to achieve the following:

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.

This difficulty comprises a large part of the motivation behind a solver like ours: by reducing the amount of uninteresting proof code a programmer needs to write, we hope to make larger formalisations of mathematics more achievable. It's absolutely *vital*, then, that the solver is easy to use: if it's easier to prove something manually than it is to automate it, what was the point of the automation in the first place?

We feel that we have improved on Agda's current solver [Danielsson 2018] in this regard. While it can automate proofs like Fig. 1a, its interface (Fig. 2) is quite verbose, and it requires programmers to learn another syntax specific to the solver.

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

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.

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. In terms of both practical use and theoretical bounds, our solver is significantly faster than Agda's current solver.

Educational Features Outside the rigorous world of dependently-typed languages, computer algebra systems (of one form or another) have had massive success among programmers and non-programmers alike. These days, many of these systems (like Wolfram|Alpha [Wolfram Research, Inc. 2019]) provide educational features, which have proven invaluable to students learning mathematics.

We will take just one of those features ("pedagogical", or step-by-step solutions [The Development Team 2009]), and re-implement it in Agda using our solver. In doing so, we will formalise the concept, and explore some of the theory behind it.

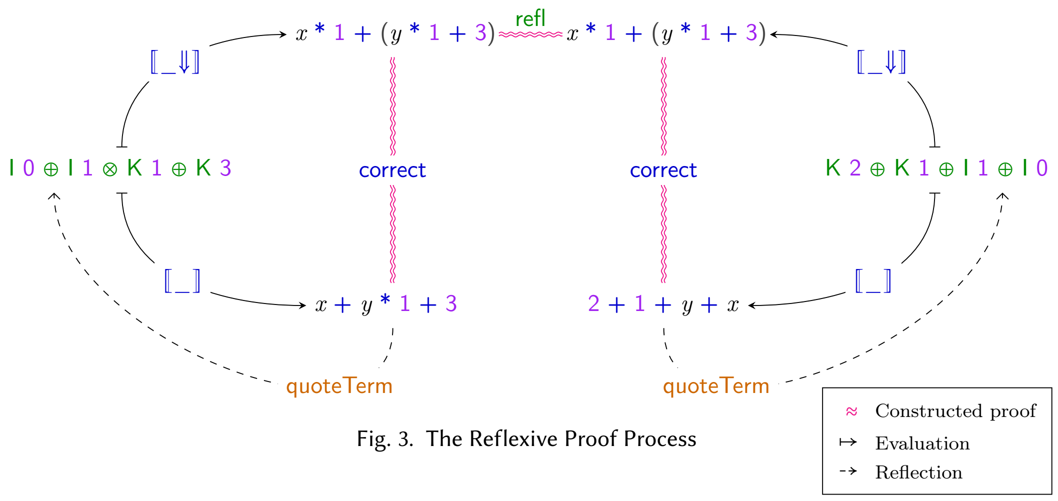

Fig. 3. The Reflexive Proof Process

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

2.1 The Expr AST

To get started, we'll first define an Abstract Syntax Tree (AST) to describe expressions which use the ring operators.

```
data Expr {ℓ} (A : Set ℓ) (n : ℕ) : Set ℓ where
  K   : A → Expr A n
  Ι   : 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
```

Each of the ring operators has a corresponding constructor ($x + y = x \oplus y$, $x^y = x \otimes y$), constants are constructed with K, and variables are referred to by their de Bruijn index (so x becomes Ι 0). The ASTs for both expressions we want to prove can be seen on either side of Fig. 3.

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

2.2 Almost Rings

While the stated domain of the solver is simply "commutative rings", it turns out that we can be slightly more flexible than that if we pick our algebra carefully. As in Grégoire and Mahboubi [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:

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

However, by choosing these slightly more complex laws, we can admit types like 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 equivalence, or provide it (and get the speedup) if it does.

2.3 Correctness

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

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

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

2.4 Flexibility

The solver (including the interface) will work with any type which implements AlmostCommutativeRing, as described above. Furthermore, we prove *equivalences* (rather than equalities), meaning that our solver will work on custom setoids. A consequence of this particular genericity is the step-by-step solutions described below.

3 THE INTERFACE

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

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) 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 [Norell 2018; van der Walt and Swierstra 2013] it is still not fully automatic.

3.1 Maintaining Invariants

One of the great promises of languages like Agda is the ability to encode program correctness in types, allowing programmers to *prove* properties they 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)
```

In contrast to size-indexed lists, this type is one you don't see very often in Agda: the reason is that the index (the size) doesn't match the shape of the data structure. As a result, almost every function which manipulates the tree in someway will have to come accompanied by a verbose, complex proof.

Take this line, for instance, which performs a left-rotation on the tree:

```
rotˡ (node {a} x xl (node {b} {c} y yr yl)) = node y (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:

```
rot' (node {a} x xl (node {b} {c} y yr yl)) = node y (node x xl yl) yr
  : Tree ⟨ solveOver (a :: b :: c :: []) Nat.ring ⟩
```

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.

```
rot' (node {a} x xl (node {b} {c} y yl yr)) = node y (node x xl yl) yr
  : Tree ⟨ solveOver (a :: b :: c :: []) Nat.ring ⟩
```

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.

3.2 Reflection

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 in general it was a *bad* idea to do 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 ⊛. Something worth noticing is that it doesn't care how long the list actually is, it only looks for the last two arguments. Again, we found that this looseness was actually a benefit:

Principle 2 (Don't assume structure). In this example, as well as elsewhere, to find arguments to an n-ary function we 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), you'll get type errors where you call solve, not where it's implemented.

The next thing to point out is 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; however, we advise being as *in*specific as possible, relying on the compiler to guess where you can:

Principle 3 (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.

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 4 (Try and implement as much of the logic outside of reflection as possible). 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 PERFORMANCE

Type-checking proof-heavy Agda code is notoriously slow, so the solver had to be carefully optimised to avoid being so slow as to be unusable. We'll start by first describing the unoptimised 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 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) []
```

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 optimisation comes from Grégoire and Mahboubi [2005]. 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.

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

This will represent the size of the gap from the previous non-zero coefficient. Taking 12 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.

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.

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 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)
```

4.4 Unification

So far, our optimisations 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. 3).

As it happens, we have now optimised 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 optimisations from Grégoire and Mahboubi [2005] applied, was in many cases *slower* than the old, unoptimised 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 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 equivalence.*

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 equivalence. This would be disastrous for performance! Consider the following:

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

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.

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 equivalence:

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

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

$[\![\_]\!]_l$ is the definition we've been working with so far. Some readers might find $[\![\_]\!]_r$ 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 $[\![\_]\!]_r$ is able to make more progress without a concrete x. Taking the polynomial $x^2 + 2$ as an example:

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

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

Surprisingly, this is exactly what you *don't* want! Since both sides of the equation will be coming out of the same normalisation function, they should have similar structures, allowing for syntactic equivalence. The coefficients, though, will be computed during manipulations on the Horner normal form, and so may contain unevaluated expressions, which require normalisation. If we used $[\![\_]\!]_r$ as our definition, then, the typechecker will likely hit an inequivalence very early on, and give up on syntactic equivalence. Even worse, because it's able to make progress, it will likely do so, completely changing the structure of the expression and ruining any chance for later syntactic equalities!

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.

4.4.2 Avoid Constants. It's a good idea to avoid constant coefficient expressions in the normal form. This will reduce the size of your expression, which is helpful in general, but more importantly it will make it less likely that the typechecker will be able to run some normalisation on one of the ring operators.

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

(a) Dense encoding, without identity-avoiding optimisation

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

(b) Sparse encoding, with identity-avoiding optimisation

Fig. 4. Comparison of the normal forms of equation 12

Our sparse representation helps significantly in this case: by removing 0 from the generated expression, we significantly improve the chances of maintaining structural similarity between the two normal forms.

Another place we can cut down on constants is in identity elements in the base case 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. 4.

4.5 Verifying the Optimisations

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 Mu et al. [2009], and relied heavily on abstraction and folds to improve the reuse in proof code. In particular, we defined many operations as *metamorphisms* [Gibbons 2007]. 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, here are the semantics of mapR, a simple morphism which behaves something like map on lists:

```
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))
```

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.

4.6 Benchmarks

As expected, the sparse implementation exhibits a significant speedup in type-checking over the dense implementation. Fig. 5a shows time taken to type check a proof that $(x_1 + x_2 + x_3 + x_4 + x_5)^d$ is equal to its expanded form. The sparse representation is clearly faster overall, with a factor of 7.5 speedup for $d = 8$.

Fig. 5b demonstrates where some of the speedup might come from. The expression x^d is represented very differently in the two implementations: in the sparse encoding, it's a single-element list containing a tuple of 1 and d; in the dense encoding, however, it's a list of length d, with a single 1 at the end. Since the complexities of arithmetic operations are bounded by the length of the list, this explains the difference in cost of the addition operations.

Fig. 5c demonstrates perhaps a more common use-case, with a mix of high and low powers and some constants. The sparse representation's advantage is even more apparent here, with an 30-factor speedup at $d = 8$.

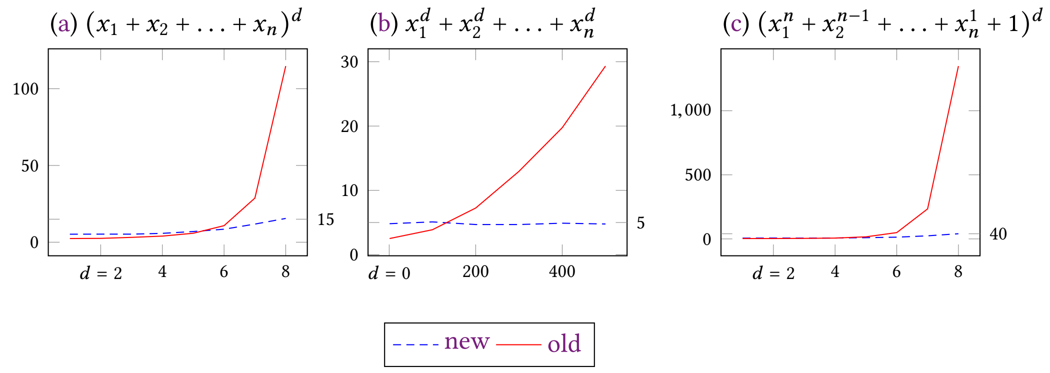

Fig. 5. Time (in seconds) to prove each expression is equal to its expanded form ($n = 5$ for each).
Benchmarks performed on Agda version 2.6-0fa9b13, with the Agda standard library at commit-3bd3334a9552490e396f73f96812105a27e5917b, on a 2016 MacBook Pro, with a 2.9 GHz Intel Core i7 and 16 GB of RAM.

The dense solver does exhibit a small lead (roughly 2-3 seconds) on very simple expressions, possibly caused by the overhead of the sparse 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 several-minute gains. Furthermore, we have not been able to find a case where the dense solver is significantly faster than the sparse. Nonetheless, if a user really wants to use the dense solver, the other components described here are entirely modular, and can work with any underlying solver which uses the reflexive technique.

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 *path-finding*. The left-hand-side and right-hand-side of the equation are vertices in a graph, where the edges are single steps to rewrite an expression to an equivalent form. A* is used to search.

Unfortunately, this approach has to deal with a huge search space: every vertex will have an edge for almost every one of the ring axioms, and as such a good heuristic is essential. Furthermore, what this should be is not clear: Lioubartsev [2016] uses a measure of the "simplicity" of an expression.

So, with an eye to using our solver to help, 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 path-compression heuristics to cut down on the solution length:

(1) First, we run (a version of) Dijkstra's algorithm on the generated path. Fig 6 shows an example solution without this heuristic applied: it crosses the same point multiple times, creating the kind of loops we want to avoid. 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.

$$\begin{tikzcd}
 & & & & 1+2+y+x \\
 & & & & 3+y+x \arrow[u] \\
x+y*1+3 \arrow[drr] & & & & y+3+x \arrow[u] \\
 & & x+y+3 \arrow[dl] \arrow[urr] & & \\
 & 3+x+y \arrow[rr] & & x+3+y \arrow[ul] &
\end{tikzcd}$$

Fig. 6. 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].

Formalisation of mathematics in general is an ongoing project. Wiedijk [2018] tracks how much of "The 100 Greatest Theorems" [Kahl 2004] have so far been formalised (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

Jeremy Gibbons. 2007. Metamorphisms: Streaming Representation-Changers. *Science of Computer Programming* 65, 2 (2007), 108–139. https://doi.org/10.1016/j.scico.2006.01.006

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/

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. 2009. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. *Journal of Functional Programming* 19, 5 (Sept. 2009), 545–579. https://doi.org/10.1017/S0956796809007345

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/