Automatically Proving Equivalence by Type-Safe Reflection

Franck Slama and Edwin Brady

University of St Andrews, Scotland, UK
`fs39@st-andrews.ac.uk, ecb10@st-andrews.ac.uk`

Abstract. One difficulty with reasoning and programming with dependent types is that proof obligations arise naturally once programs become even moderately sized. For example, implementing an adder for binary numbers indexed over their natural number equivalents naturally leads to proof obligations for equalities of expressions over natural numbers. The need for these equality proofs comes, in intensional type theories, from the fact that the propositional equality enables us to prove as equal terms that are not judgementally equal, which means that the typechecker can't always obtain equalities by reduction. As far as possible, we would like to solve such proof obligations automatically. In this paper, we show one way to automate these proofs by reflection in the dependently typed programming language Idris. We show how defining reflected terms indexed by the original Idris expression allows us to construct and manipulate proofs. We build a hierarchy of tactics for proving equivalences in semi-groups, monoids, commutative monoids, groups, commutative groups, semi-rings and rings. We also show how each tactic reuses those from simpler structures, thus avoiding duplication of code and proofs.

Keywords: proof automation, equivalence, equality, proof by reflection, correct-by-construction software, type-driven development

1 Introduction

Proofs assistants like Coq [1] and programming languages like Agda [17] and Idris [3] are based on Intensional Type Theories that contain two notions of equality: propositional equality, that can be manipulated in the language, and judgemental (or definitional) equality. Propositional equality corresponds to the mathematical notion: this is a proposition that can be assumed, negated, proved or disproved. Since in type theory, propositions are types [13], the proposition that two elements x and y are equal corresponds to a type. If x and y are of type a, then the type $Id_a(x, y)$ represents the proposition "x is equal to y". If this type is inhabited, then x is said to be provably equal to y. Thus, Id is a type family (parameterised by the type a) indexed over two elements of a, giving $\texttt{Id}\ (a : \texttt{Type}) : a \rightarrow a \rightarrow \texttt{Type}$. For convenience, we write $(\texttt{Id}_a\ x\ y)$ as $(x\ =_a\ y)$.

Judgemental equality, on the other hand, is a primitive concept of the type theory. Whether or not two expressions are judgementally equal is a matter of evaluating the definitions. For example, if $f : \mathbb{N} \to \mathbb{N}$ is defined by $f\ x := x + 2$, then $f\ 5$ is definitionally equal to 7. Definitional equality entails unfolding of functions and reductions, until no more reduction can be performed. We denote the definitional equality by $\equiv$.

Judgemental equality is included in propositional equality because what is equal by definition is provably equal. This is accomplished by giving a constructor for the type $Id(a, a)$ and no constructor when "a is not b". In these theories, Id is therefore implemented with the following type with one constructor:

```
data Id : a → a → Type where
     Refl : (x : a) → Id x x
```

The only way for ($\mathtt{Id}_a\ x\ y$) to be inhabited is therefore that x and y are equal by definition. In this case, the constructor `Refl` helps to create a proof of this equality: `(Refl x)` is precisely the proof which says that $x =_a x$. Here, we are using the notation of Idris, where unbound variables like `a` in the definition of `Id` are *implicitly* quantified, as a concise programming notation.

The propositional equality does not only contain the judgemental equality, however, because a principle of induction is associated with each inductive type. If `T` is an inductive type with a constant constructor and a recursive constructor, ie, `T = 1 + T`, defined in Idris as:

```
data T : Type where
  T0 : T
  T1 : T → T
```

then we have the following induction principle for `T`:
$\mathtt{T\_ind} : \forall P : \mathtt{T} \to \mathtt{Type},\ (P\ \mathtt{T0}) \to (\forall t : \mathtt{T},\ P\ t \to P\ (\mathtt{T1}\ t)) \to (\forall t : \mathtt{T},\ P\ t)$.

For example, we can prove that $n+0 = n$ for all n by induction on the $Nat\ n$, even if $n+0 \not\equiv n$ with the usual definition of $+$, recursive on its first argument. So, the axiom of induction means the type $Id_a(x, y)$ contains not only the canonical form `Refl`, but also those added by inductive principles. There are therefore things which are *provably* equal, but not *definitionally* equal. Proving equalities is therefore in these theories something which isn't automatically decidable by the type-checker in the general case.

1.1 Motivating example: Verified Binary Arithmetic

Proving that one term is equal to another is common in formal verification, and proof obligations arise naturally in dependently typed programming when indexing types over values in order to capture some logical properties. To demonstrate this, we revisit an example from previous work [2] which shows how proof obligations arise when a type is indexed by natural numbers. Our goal is to implement a verified library of binary numbers. To ensure functional correctness, we define the types `Bit` and `Binary` indexed over the value they represent (expressed as a natural number):

```
data Bit : Nat → Type where
```

```
     b0 : Bit Z
     b1 : Bit (S Z)

data Binary : (width : Nat) → (value : Nat) → Type where
     zero : Binary Z Z
     (#) : Binary w v → Bit bit → Binary (S w) (bit + 2 * v)
```

We will write a function to add two binary numbers. To do so, we begin with an auxiliary function, which adds three bits (the third is a carry bit), and produces the two bits of the result, where the first is the more significant bit:

```
addBit : Bit x → Bit y → Bit c → (bX ** (bY **
         (Bit bX, Bit bY, c + x + y = bY + 2 * bX)))
addBit b0 b0 b0 = (_ ** (_ ** (b0, b0, Refl)))
addBit b0 b0 b1 = (_ ** (_ ** (b0, b1, Refl)))
{- ... remaining cases follow the same pattern ... -}
```

The syntax `(n ** t)` denotes a *dependent pair*, where the type of the second argument `t` can refer to the first argument `n`. So, we can read this type as: "there exists a number `bX`, and a number `bY`, such that we have two bits `Bit bX` and `Bit bY` and the sum of the input bits `c`, `x` and `y` equals `bY + 2 * bX`." For example, on the second line, which corresponds to the computation $0_2 + 0_2 + 1_2 = (01)_2$, the function produces this bits `b0` and `b1`, and a proof that $0+0+1 = 1 + (2 \times 0)$.

We then define the function `adc` that adds two binary numbers and a carry bit. This works for two binary numbers with the same number of bits, and produces a result with one more bit. We would like to write:

```
adc : Binary w x → Binary w y → Bit c → Binary (S w) (c + x + y)
adc zero zero carry = zero # carry
adc (numx # bX) (numy # bY) carry
  = let (vCarry0 ** (vLsb ** (carry0, lsb, _)))
             = addBit bX bY carry in
        adc numx numy carry0 # lsb
```

Unfortunately, this definition is rejected because the types do not match for both patterns. For the second case, the expected index is:
`((c + (bit2 + (v1 + (v1 + 0)))) + (bit + (v + (v + 0))))`
while we're trying to provide a term indexed over:
`vLsb + (((vCarry0 + v1) + v) + (((vCarry0 + v1) + v) + 0))`.

The definition of `adc` we have given would behave correctly, and it has *provably* the expected type, but it does not have it *immediately* or *judgementally*: after full reductions the expected and provided types are still different. To make the previous definition acceptable, we need to solve two proof obligations `adc_lemma_1` and `adc_lemma_2` which demand proofs of equality between the two types. For example, using a proof script:

```
adc_lemma_2 = proof {
    intros;
    rewrite sym (plusZeroRightNeutral x);
    [...]
    rewrite (plusAssociative c (plus bit0 (plus v v)) bit1);
    rewrite plusCommutative bit1 (plus v v);
    [...]
    rewrite (plusAssociative (plus (plus x v) v1) (plus x v) v1);
    trivial;
}
```

Such proofs consist of a potentially long sequence of rewriting steps, each using one of the properties: neutral element, commutativity, associativity. Without some automation, this sequence of rewritings must be done by the programmer. Not only is this time consuming, but a small change in the definition may lead to a different proof obligation, thus invalidating the proof. A minor change in the datatype, or the definition of `addBit` or `adc` will require us to do a new proof, and thus, without support from the machine, these proofs could become the everyday routine in any dependently-typed language. It is worth mentioning that even without using dependent types, these proof obligations for equalities happen very frequently during the formal certification of most applications.

Our handwritten proof `adc_lemma_2` uses only the existence of a neutral element, and the associativity and commutativity of + on `Nat`. Thus, we're rewriting a term by using the properties of a commutative monoid. With the right choice of combinators [4] such proofs could be made much simpler, but we would like a generic prover for commutative monoids to find a proof automatically.

1.2 Our contributions

Provers for some algebraic structures have already been implemented for various proof assistants, including Coq [12] and Agda[1]. In this paper, we describe an implementation[2] of an automatic prover for equalities in a *hierarchy* of algebraic structures, including monoids, groups and rings (all potentially commutative), for the Idris language, making the following contributions:

1. We present a type-safe reflection mechanism (section 2.3), where the reflected terms are indexed over the concrete terms, providing a direct way to extract proofs and guaranteeing that the reflected term is a sound representation.
2. The normalisation procedures are implemented by following a correct by construction approach (section 2.4), instead of proving the correctness afterwards with auxiliary lemmas.
3. We develop a *hierarchy* of tactics where each tactic reuses the rewriting machinery of the structure from which it inherits. For example, simplifying neutral elements is implemented only at the monoid level, and reused at other levels. It is challenging to reuse the prover of a less expressive structure; for

[1] `http://wiki.portal.chalmers.se/agda\%5C?n=Libraries.UsingTheRingSolver`

[2] The implementation of our hierarchy of tactics can be found online at `https://github.com/FranckS/RingIdris/Provers`

example, using the monoid prover to build the group prover is tricky because we lose the ability to express negations ($-x$) and subtractions ($x - y$). We present encodings (section 2.6) to overcome this problem.

The principal novelty is in using *type-safe reflection*. Working by reflection for implementing tactics has been done several times, including the implementation of a ring solver for Coq, but without the type-safety and correctness-by-construction. We compare our approach with other implementations in section 3.

2 A hierarchy of provers

We aim to build a prover not only for equalities on `Nat`, `List`, or any specific type, but for generic datatypes and properties. Using the right abstraction, we can generate proofs of equalities for many datatypes *at once* by implementing a generic hierarchy of provers for semi-groups, monoids, commutative-monoids, groups, commutative-groups, semi-rings and rings. The properties of an algebraic structure are expressed in an interface (an interface in Idris is similar to a type class in Haskell). This interface will extend the interface from which it inherits; for example, `Group` extends `Monoid`. This leads to a hierarchy of interfaces, with one tactic for each. At every level of the hierarchy, we will be able to work on any type, as long as there is a corresponding implementation of the interface.

2.1 Proving equivalences instead of equalities

With some additional effort, we can produce a collection of tactics for proving *equivalences*, rather than only equalities. The machinery is very similar and we gain another degree of genericity, with the freedom of choosing the equivalence relation (which can be the usual equality). The user can define their own notion of equivalence, as long as they provide the proofs of the properties of the relevant algebraic structure. Let's call `c` the *carrier* type, i.e., the type on which we want to prove equivalences. The equivalence relation on `c` has the following profile $(\simeq)$: `c` $\to$ `c` $\to$ `Type`[3], and must be reflexive, symmetric and transitive.

Our tactics need to be able to test this equivalence between elements of the underlying set, that is a way of testing equivalence of constants. We therefore define a notion of `Set`[4], which requires the definition of the equivalence relation and an equivalence test `set_eq`. All the interfaces representing the algebraic structures will later extend `Set`:

```
interface Set c where
    (≃) : c → c → Type
    refl : (x : c) → x ≃ x
```

[3] This `Type` would be a `Prop` in systems, like Coq, that make a distinction between the world of computations and the world of logical statements

[4] This notion of set is a way to talk about the carrier type and an equivalence relation, sometimes called Setoid

```
    sym : {x, y : c} → (x ≃ y) → (y ≃ x)
    trans : {x, y, z : c} → (x ≃ y) → (y ≃ z) → (x ≃ z)
    set_eq : (x : c) → (y : c) → Maybe (x ≃ y)
```

To prove propositional equalities, the user instantiates ($\simeq$) with the built-in ($=$) when implementing `Set`. Note that ($\simeq$) is only weakly decidable in the sense that `set_eq` only produces a proof when the two elements are equivalent, but it doesn't produce a proof of dis-equivalence when they are different, instead producing the value `Nothing`. Our goal is only to generate proofs of equivalance, not to produce counter-examples. There is no tactic associated with `Set`, since we have no operations or properties associated to this structure. Equivalences in a `Set` are "syntactic equivalences" and can be proven with `refl`[5].

Working with equivalences instead of equalities brings one complication : the proofs of correctness that we produce by hand cannot use Idris' "rewrite" mechanism, which enables rewriting of a subterm by another one, provided that the two subterms are propositionally equal. This is a classical problem of working within a setoid, which can be mitigated by programming language support for rewriting terms in setoids. However, Idris is not equipped with any such support. For this reason, we define the following lemma, using the methods of the `Set` interface:

`eq_preserves_eq` : $\{c : \texttt{Type}\} \rightarrow \{\texttt{Set}\ c\} \rightarrow (x : c) \rightarrow (y : c) \rightarrow (c1 : c) \rightarrow (c2 : c) \rightarrow (x \simeq c1) \rightarrow (y \simeq c2) \rightarrow (c1 \simeq c2) \rightarrow (x \simeq y)$.

This lemma says that the equivalence preserves the equivalence, which means that in order to prove $x \simeq y$, we can prove a smaller problem $c1 \simeq c2$, provided that $x \simeq c1$ and that $y \simeq c2$. We will use this lemma extensively.

2.2 Hierarchy of interfaces

We describe operations, constants and properties of each algebraic structure in an interface. The first algebraic theory is `Magma`, which is a structure built on top of `Set` that adds `Plus` operation, and no specific properties:

```
interface Set c => Magma c where
    + : c → c → c
```

This code means that a type `c` (for *carrier*) is a `Magma` if it is already a `Set` (ie, it is equipped with the equivalence relation $\simeq$ and the equivalence test `set_eq`), and if it has a $+$ operation. In fact, there is an additional requirement that will apply to all operations (in this case, the $+$ operation), which is that they need to be "compatible" with the equivalence relation, which is expressed by the following axiom for $+$:

`Plus_preserves_equiv` : $\{c : \texttt{Type}\} \rightarrow \{\texttt{Magma}\ c\} \rightarrow \{c1 : c\} \rightarrow \{c2 : c\} \rightarrow \{c1' : c\} \rightarrow \{c2' : c\} \rightarrow (c1 \simeq c1') \rightarrow (c2 \simeq c2') \rightarrow ((c1 + c2) \simeq (c1' + c2'))$

[5] `refl` is not to be confused with `Refl`, the constructor of $=$, but when ($\simeq$) is instantiated with the equality $=$, `refl` is implemented by `Refl`. Therefore, `refl` of the interface `Set` is a generalisation of `Refl`

We have this requirement because we support any equivalence relation. The user is free to define the equivalence relation of their choice, but it should be compatible with the operations that they are using. As with `Set`, there is no tactic for `Magma`, because there is no property; all equivalences are again syntactic equivalences, and can thus be proven by `refl`.

A semi-group is a magma (ie, it still has a `Plus` operation), but moreover it has the property of associativity for this operation.

```
interface Magma c => SemiGroup c where
  Plus_assoc : (c1 : c) → (c2 : c) → (c3 : c) →
               ((c1 + c2) + c3 ≃ c1 + (c2 + c3))
```

Examples of magma are `Nat` equipped with addition, and `List` with concatenation. Next, a monoid is a semi-group with the property of neutral element for a distinguished element called `Zero`.

```
interface SemiGroup c => Monoid c where
  Zero : c
  Plus_neutral_1 : (c1 : c) → (Zero + c1 ≃ c1)
  Plus_neutral_2 : (c1 : c) → (c1 + Zero ≃ c1)
```

The hierarchy of interfaces continues with `Group`:

```
interface Monoid c => Group c where
  Minus : c → c → c
  Neg : c → c
  Minus_simpl : (c1 : c) → (c2 : c) → Minus c1 c2 ≃ c1 + (Neg c2)
  Plus_inverse : (c1 : c) → (c1 + (Neg c1) ≃ Zero,
                             (Neg c1) + c1 ≃ Zero)
```

The notion of group uses two new operations (`Neg` and `Minus`), but `Minus` can be simplified with $+$ and `Neg`. The important property of a group is that every element `c1` must admit `Neg c1` as inverse element for $+$. For reasons of space, we elide the remaining details of the hierarchy.

2.3 Type-safe reflection

When proving an equivalence $x \simeq y$, the universally-quantified variables are abstracted and they become part of the context. Our tatics normalise both sides of the "potential equivalence" $x \simeq y$, and compare the results by syntactic equivalence. The difficulty is that the normalisation function needs to consider variables, constants and operators. For this reason, we work by reflection, which allows us to work on syntax instead of values. We define one type of reflected terms for each algebraic structure. The novelty is not the use of reflected terms, but the use of a type-safe reflection mechanism where we index the reflected terms by the concrete value that they represent. Each of these datatype is parametrised over a type `c`, which is the type on which we want to prove equalities (the *carrier* type). It is also indexed over an implementation of the corresponding interface for `c` (we usually call it `p`, because it behaves as a *proof* telling that the structure `c` has the desired properties), indexed over a context of abstracted variables (a vector Γ of `n` elements of type `c`). Most importantly, it is indexed over a value of type `c`, which is the concrete value being encoded.

A magma is equipped with one operation, addition. Thus, to reflect terms in a magma we express constants, variables, and addition:

```
data ExprMa : Magma c → (Vect n c) → c → Type where
  ConstMa : (p : Magma c) → (Γ:Vect n c) → (c1:c) → ExprMa p Γ c1
  PlusMa : {p : Magma c} → {Γ:Vect n c} → {c1:c} → {c2:c}
       → ExprMa p Γ c1 → ExprMa p Γ c2 → ExprMa p Γ (c1+c2)
  VarMa : (p:Magma c) → (Γ:Vect n c)
       → (i:Fin n) → ExprMa p Γ (index i Γ)
```

For an expression e_x : `ExprMa` Γ x, we say that e_x denotes the value x in the context Γ. When an expression is a variable `(VarMa _ Γ i)`, the denoted value is the corresponding variable in the context, i.e., `(index i Γ)`. The expression `(ConstMa _ Γ k)` denotes the constant k in any context Γ. Finally, if e_x is an expression encoding the concrete value x, and e_y is an expression encoding the concrete value y, then the expression `PlusMa` e_x e_y denotes the concrete value $(x + y)$. Because the reflected terms embed their corresponding inputs, they are guaranteed to be sound representations. This is a *local* approach to syntax representation [11] in that the reflected representation will *only* represent terms in a magma.

There are no additional operations in semi-groups or monoids, so the reflected datatypes have the same shape as that for magma. However, the datatype for reflected terms in groups introduces two new constructors for `Neg` and `Minus`:

```
data ExprG :  Group c → (Vect n c) → c → Type where
    ConstG : (p : Group c) → (Γ:Vect n c) → (c1:c) → ExprG p Γ c1
    PlusG : {p : Group c} → {Γ:Vect n c} → {c1:c} → {c2:c}
         → ExprG p Γ c1 → ExprG p Γ c2 → ExprG p Γ (c1+c2)
    MinusG : {p : Group c} → {Γ:Vect n c} → {c1:c} → {c2:c}
         → ExprG p Γ c1 → ExprG p Γ c2 → ExprG p Γ (Minus c1 c2)
    NegG : {p : Group c} → {Γ:Vect n c} → {c1:c}
         → ExprG p Γ c1 → ExprG p Γ (Neg c1)
    VarG : (p : Group c) → (Γ:Vect n c)
         → (i:Fin n) → ExprG p Γ (index i Γ)
```

The index of type c (the value encoded by an expression) is always expressed by using the lookup function `index` and the available operations in the implementation p, which for a group are $+$, `Minus` and `Neg`.

2.4 A correct-by-construction approach

We take a *correct-by-construction* approach to implementing normalisation, which means that no additional proof will be required after defining normalisation. The normalisation function `norm` produces a new expression, and a proof that it has the same interpretation as the original. This will be enforced by the fact that all

the datatypes for reflected terms (`ExprMa`, `ExprG`, `ExprR`, etc) are indexed over the concrete value: a term of type `Expr` Γ x is the encoding of the concrete value x in the context Γ. For each structure, the type of `norm` has the following form:
`norm` : `Expr` $\Gamma\ x \ \rightarrow\ (x'$ `**` (`Expr` $\Gamma\ x',\ x\ \simeq\ x'$))
Every instance of `norm` produces a dependent pair: the new concrete value x', and a pair made of an `Expr` $\Gamma\ x'$ which is the new encoded term indexed over the new concrete value we have just produced, and a proof that old and new concrete values x and x' are equivalent. This proof of $x \simeq y$ is the crucial component which allows us to automatically produce proofs of equivalences.

We will explain how to implement the tactic for `Group` specifically, and the other structures are implemented similarly. The equivalence we are trying to prove is $x \simeq y$, where x and y are elements of the type c, which implements a group. The reflected term for x is denoted e_x, and has type `ExprG` $p\ \Gamma\ x$, which means that e_x is guaranteed to be the encoding of x. Similarly, y, is encoded by e_y, and its type is indexed by the value y. Evaluating `norm` on e_x produces the normal form $e_{x'}$ of type `ExprG` $p\ \Gamma\ x'$ and a proof p_x of $x \simeq x'$. Similarly, for e_y, it produces the normal form $e_{y'}$ of type `ExprG` $p\ \Gamma\ y'$ and a proof p_y of $y \simeq y'$. It now suffices to compare the normal forms $e_{x'}$ and $e_{y'}$ using a syntactic equivalence test `ExprG_eq`, because once everything is in normal form, being equivalent is just a matter of being syntactically equivalent.

```
exprG_eq : {c:Type} → {n:Nat} → (p:Group c) → (Γ:Vect n c)
           → {x' : c} → {y' : c}
           → (ex' : ExprG p Γ x') → (ey' : ExprG p Γ y')
           → Maybe(x' ≃ y')
```

This syntactical equivalence test checks if the two input terms ex' and ey' are *syntactically* the same, and if they do, it directly builds a proof of equivalence between their indices $x' \simeq y'$, which is what we need, because we can use it with the two equivalences $x \simeq x'$ and $y \simeq y'$ that we already have, in order to get the desired proof of $x \simeq y$ with `eq_preserves_eq`. We put all of this together in a function `buildProofGroup`:

```
buildProofGroup : (p:Group c) → {Γ:Vect n c} → {x : c} → {y : c}
  → {x':c} → {y':c} → (ExprG p Γ x') → (ExprG p Γ y')
  → (x ≃ x') → (y ≃ y') → (Maybe (x ≃ y))
buildProofGroup p ex' ey' px py with (exprG_eq p ex' ey')
    buildProofGroup p ex' ey' px py | Just ex'_equiv_ey' =
                 Just(eq_preserves_eq x y x' y' px py ex'_equiv_ey')
    buildProofGroup p ex' ey' px py | Nothing = Nothing
```

The arguments of type `ExprG` $p\ \Gamma\ x'$ and `ExprG` $p\ \Gamma\ y'$ are the normalised reflected left and right hand sides of the equivalence, which respectively represent the value x' and y'. This function also expects proofs of $x \simeq x'$ and of $y \simeq y'$, which are built by the normalisation process.

Finally, the main function which tries to prove the equivalence $x \simeq y$ has to normalise the two reflected terms encoding the left and the right hand side, and use the function `buildProof` to compose the two proofs:

```
groupDecideEq : (p:Group c) → {Γ:Vect n c} → {x : c} → {y : c}
              → (ExprG p Γ x) → (ExprG p Γ y) → Maybe (x ≃ y)
groupDecideEq p ex ey =
  let (x' ** (ex', px)) = groupNormalise p ex in
  let (y' ** (ey', py)) = groupNormalise p ey in
            buildProofGroup p ex' ey' px py
```

It remains to define the function `groupNormalise`, which is an instance of `norm` for groups:

```
groupNormalise : {c:Type} → {n:Nat} → (p:Group c) → {Γ:Vect n c}
       → {x:c} → (ExprG p Γ x) → (x' ** (ExprG p Γ x', x ≃ x'))
```

Each algebraic structure is equipped with a function for reducing reflected terms to their normal form. The algebraic theories which concern us admit a canonical representative[6] for any element, a property which we use to decide equalities. Without this property, it would be more complicated to decide equivalence without brute-forcing a series of rewritings, that would have no termination guarantee.

The normalisation function has more work to do for structures with many axioms (commutative-monoids, groups, commutative-groups, semi-rings and rings), than for the simpler structures (semi-groups and monoids). In the next section, we describe the normalisation process.

2.5 Normalisation functions

We describe the normal form for rings, which is our most sophisticated structure. The input to the normalisation function is an expression with sums, products, constants and variables belonging to an ordered set $\mathcal{V}$ of variables. In short, the normalisation function takes in input a polynomial of multiple variables. As output, it produces a normal form representing the same polynomial. Therefore, we need a canonical representation of polynomials. There are several possibilities, but we choose classical mathematical conventions: the polynomial will be completely developed, i.e., the distributivity of $*$ over $+$ will be applied until it cannot be applied further. This is a simple but effective approach: the benefit of simplicity is that we can directly produce a proof of equivalence between the new and old concrete values during normalisation. Because the polynomial is completely developed, at the toplevel, it is a sum:

$$P = \sum_{i=1}^{a} (\prod_{j=1}^{b} Monomial_i^j) \text{ where } Monomial_i^j = C_i^j * \prod_{k=1}^{c} Var_{i,k}^j$$

with C_i^j a constant, and $Var_{i,k}^j$ one of the variable that belong to $\mathcal{V}$.

It may be surprising that the normal form is a sum of product of monomials, and not directly a sum of monomials. This is because a monomial is a product of

[6] It only holds for "pure" algebraic structures, ie, in the absence of additional axioms

a constant C_i^j (e.g. 5) and of a product of variables (e.g. $x * y * z$). For example, $5*(x*(y*z))$ is a monomial. Now let's consider the term $(5*(x*(y*z)))*(4*(z*z))$. This term is not a monomial, but we could be tempted to simplify it into the monomial $20 * (x * (y * (z * (z * z))))$. However, that would assume that the product is always commutative, which is not the case for *every* ring. Therefore, after development, the polynomial is a sum of *product of monomials*, and not directly a sum of monomials. The only rearrangement that can and needs to be done towards the multiplication is to check if two constants are consecutive in a product, and if so, to replace them by the constant that represents their product.

However, because $+$ is always commutative in a ring, the different products of monomials themselves can be rearranged in different ways in this sum. This will be done at the lower level for commutative groups if we can provide an ordering on products of monomials. This ordering will be defined by using an ordering on monomials, called `isBefore_mon`, which looks at the order of variables for comparing two monomials $Monomial_i^j$ and $Monomial_{i'}^{j'}$.

$$Monomial_i^j = C_i^j * (Var_{i,1}^j * \prod_{k=2}^{c} Var_{i,k}^j) \text{ and } Monomial_{i'}^{j'} = C_{i'}^{j'} * (Var_{i',1}^{j'} * \prod_{k=2}^{c'} Var_{i',k}^{j'})$$

The order between these two monomials is decided by looking at the order between the variables $Var_{i,1}^j$ and $Var_{i',1}^{j'}$. If both monomials start with the same variable, we continue by inspecting the remaining variables. If one of the two monomials has fewer variables, that one comes first.

We can now build the order on *product of monomials*, named `isBefore`. Given two products of monomials $Prod_i$ and $Prod_{i'}$ we need to decide which one comes first. We will use the order `isBefore_mon` on the first monomials of these two products. If it says that $Monomial_i^1$ comes before $Monomial_{i'}^1$, then we decide that $Prod_i$ comes before $Prod_{i'}$. Conversely, if $Monomial_{i'}^1$ comes first, then $Prod_{i'}$ comes first. However, if $Monomial_i^1$ and $Monomial_{i'}^1$ have exactly the same position in the order, then we continue by inspecting the remaining monomials recursively. As previously, if one of the two products has fewer monomials than the other, then that one comes first.

Additionally, we use the following conventions when deciding on a normal form:

- The top-level sum of the polynomial is in right-associative form:
  $prodMon_1 \ + \ (prodMon_2 \ + (prodMon_3 \ + \ (... \ + \ prodMon_a)))$
- All the products that we have (the products of monomials and the products of variables), are written in right-associative form.
- We simplify as much as possible with constants. This includes simplifying addition with zero and multiplication with the constants zero and one, doing the computations between two nearby constants, etc...
- We simplify the sum of an expression e and its inverse $-e$ to zero.

2.6 Reusing the provers

A novelty of our work is that instead of building a prover for a specific algebraic structure, we have built a hierarchy of provers. Each prover reuses components of the others so that the simplifications are not duplicated at different levels: normalisation of each structure uses as much as possible the normalisation function of the structure from which it inherits. For example, normalisation on monoids reuses normalisation on semi-groups so that it does not have to deal with associativity. In this case, the datatype reflecting terms in semi-groups has the same expressive power as that for monoids, so a term in a monoid can be transformed directly into a corresponding term of a semi-group. However, there is a difficulty with groups and monoids: if normalisation for groups uses the normalisation for monoids, we will have to encode negations somehow, which can't be directly expressed in a monoid. Therefore, we develop some specific encodings.

The idea is that we encode negations as variables, and let the monoid prover consider them as ordinary variables. To achieve this, we use the following datatype that helps us distinguish between a variable and the encoding of a negation:

```
data Variable : {c:Type} → {n:Nat} → (Vect n c) → c → Type where
  RealVariable : (Γ:Vect n c) →
                  (i:Fin n) → Variable Γ (index i Γ)
  EncodingNeg : (Γ:Vect n c) →
                (i:Fin n) → Variable Γ (Neg (index i Γ))
```

We only need to encode negations of variables, as we can simplify the negation of a constant into a constant. Also, there cannot be a negation of something different non-atomic (i.e. a term that is not a variable or a constant), because normalisation of groups has systematically propagated `Neg` inside the parentheses, following simplification[7] $-(a+b) = -b + -a$.

All the constructors for variables now take a `Variable` as parameter, instead of taking directly an element of (`Fin` n). That gives the following, for groups:

```
    VarG : (p:Group c) → {Γ:Vect n c} → {val:c}
           → (Variable Γ val) → ExprG p Γ val
```

Thanks to this encoding, we can now transform an `ExprG` from the group level to an `ExprMo` at the monoid level. A constant (`ConstG` p Γ $c1$) is transformed into the corresponding constant (`ConstMo` _ Γ $c1$), a `PlusG` into the corresponding `PlusMo`, a variable into the same variable, the negation of a constant into the resulting constant, and finally the negation of a variable i into a (`VarMo` _ (`EncodingNeg` Γ i)).

We use a similar technique for converting an expression from the ring level to the commutative-group level, where we encode the product of monomials, because the product is not defined at the commutative-group level. That enables the function of normalisation for rings to benefit from the normalisation function for commutative-groups.

[7] Note that we have to be careful and not simplify it to $(-a)+(-b)$ as it would assume that $+$ is commutative

2.7 Automatic reflection

We have built an automatic reflection mechanism which enables the machine to build reflected terms. This is not essential to our approach, but it simplifies the usage of the tactics by removing the need to write long encodings by hand. To do so, we used Idris' reflection mechanism, which enables pattern matching on syntax, rather than on constructors. While we omit the full details due to space constraints, reflecting values involves defining functions of the following form:

```
%reflection reflectGroup : (p : Group c) → (Γ : Vect n c) →
                            (x : c) → (Γ' ** ExprG p (Γ ++ Γ') x)
```

The `%reflection` annotation means that Idris treats this as a compile time function on *syntax*. Given a value of type `c`, in some context Γ, it constructs a reflected expression in an extended context Γ `++` Γ'. The context contains references to subexpressions which are not themselves representable by `ExprG`.

3 Related work

Coq's ring solver [12], like ours, is implemented using proof-by-reflection techniques, but without the guarantees obtained with our type-safe reflection mechanism, and without the correct-by-construction approach: first, they define the normalisation of terms, then they prove correctness of the normalisation with an external lemma: $\forall\ (e1\ e2\ :\ \texttt{Expr}),\ beq_{Expr}\ (\texttt{norm}\ e1)\ (\texttt{norm}\ e2)\ =\ true\ \rightarrow\ \texttt{reify}\ e1\ \simeq\ \texttt{reify}\ e2$. This needs a `reify` function, which we do not need. Furthermore, Coq's prover deals with rings and semi-rings (commutative or not), but not with any of the intermediate structures (semi-group, monoid, group, etc). However, their implementation has better performances due to the use of sparse normal form and more optimised algorithms. Our automatic reflection was written with Idris reflection mechanism which allows pattern matching on syntax, and their automatic reflection is programmed in Ltac [7], a proof dedicated and untyped meta-language for the writing of automations. More recently, the Mtac extension [18] provides a typed language for implementating proof automation.

As well as the ring solver, Coq also provides the Omega solver [6], which solves a goal in Presburger arithmetic (i.e. a universally quantified formula made of equations and inequations), and a field [9] decision procedure for real numbers, which plugs to Coq's ring prover after simplification of the multiplicative inverses. Agda's reflection mechanism[8] gives access to a representation of the current goal (that is, the required type) at a particular point in a program. This allows various proof automations to be done in Agda [14, 15].

Proofs by reflection has been intensively studied [5, 16], but without anything similar to the type-safe reflection that we have presented here. If we leave the ground of nice mathematical structures, one can decide to work with arbitrary

[8] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Reflection`

rewriting rules, but in the general case there isn't a complete decision procedure for such systems, because there is usually no normal form. This is where deduction modulo [10, 8] and proof search heuristics start.

4 Results, conclusions and future work

We have implemented a generic solution to the initial problem of index mismatch (section 1.1) when using indexed types. This solution takes the form of a hierarchy of provers for equivalences in algebraic structures. These provers are generic in several ways: they work for many algebraic structures (semi-group, monoid, commutative monoid, group, commutative group, ring and semi-ring); for any type that behaves as one of these structures; and, for any equivalence relation on this type. The implementation is modular and each prover reuses the prover of the structure from which it inherits. These provers can automatically prove equivalences between terms, so the user need not prove obligations by hand, like `adc_lemma_2`. Thus, these provers enable the user to focus on the interesting proofs that requires specific knowledge and creativity, instead of routine, automatable, lemmas.

Our correct-by-construction method involved the design of a type-safe reflection mechanism where reflected terms are indexed over concrete inputs, and from which we are able to extract proofs directly. Unlike Coq's and Agda's ring provers, we do not have the duplication that arise when separating the computational content from the proof of correctness. Instead, construction of the proof is done step by step, following the construction of the normalised terms. In addition to avoiding redundancy, this simplifies the proof generation considerably. This development shows that if dependent types effectively bring some new problems, they are also very expressive tools for building correct-by-construction software where the development is *driven by the types*.

This work can be extended to build new provers for less common algebraic structures and for more specific structures. For example, regular expressions form a "pre semi-ring" with some extra axioms. We will refactor the semi-ring level with the creation of the intermediate structure of pre semi-ring, that will not necessary have the property that 0 is an annihilator element for the product. Then, we could build a specific prover for regular expressions, that would use the normalisation function of the pre semi-ring level and that would only have to deal with the specific properties of regular expressions : the neutral element $\varnothing$ for the concatenation of languages is also a neutral element for the product of languages, the idempotence of the addition of languages, and the rules of simplifications for the new Kleene star operation.

Acknowledgements We thank the anonymous reviewers and Jacques Carette for their insightful comments on an earlier draft. We are also grateful for the support of the Scottish Informatics and Computer Science Alliance (SICSA) and EPSRC grant EP/N024222/1.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
2. Brady, E.: Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. In: Trends in Functional Programming (TFP'07) (2007)
3. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23, 552–593 (September 2013)
4. Carette, J., O'Connor, R.: Theory presentation combinators. In: International Conference on Intelligent Computer Mathematics (CICM) 2012 (2012)
5. Chlipala, A.: Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
6. Crégut, P.: Une procédure de décision reflexive pour un fragment de l'arithmétique de Presburger. In Journées Francophones des Langages Applicatifs. (2004)
7. Delahaye, D.: A proof dedicated meta-language. Electr. Notes Theor. Comput. Sci. 70(2), 96–109 (2002)
8. Delahaye, D., Doligez, D., Gilbert, F., Halmagrand, P., Hermant, O.: Zenon modulo: When achilles outruns the tortoise using deduction modulo. In: Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19. pp. 274–290 (2013)
9. Delahaye, D., Mayero, M.: Field, une procédure de décision pour les nombres réels en Coq. In: Castéran, P. (ed.) Journées francophones des langages applicatifs (JFLA'01). pp. 33–48. Collection Didactique, INRIA (2001)
10. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. J. Autom. Reasoning 31(1), 33–72 (2003), `http://dx.doi.org/10.1023/A:1027357912519`
11. Farmer, W.M.: The formalization of syntax-based mathematical algorithms using quotation and evaluation. In: International Conference on Intelligent Computer Mathematics (CICM) 2013 (2013)
12. Gregoire, B., Mahboubi, A.: Proving Equalities in a Commutative Ring Done Right in Coq. In: Theorem Proving in Higher Order Logics (TPHOLS 2005). pp. 98—-113 (2005)
13. Howard, W.: The formulae-as-types notion of construction. In: Seldin, J., Hindley, J. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism (1980)
14. Kokke, P., Swierstra, W.: Auto in Agda — programming proof search using reflection. In: Mathematics of Program Construction - 12th International Conference, MPC 2015. pp. 276–301 (2015)
15. Lindblad, F., Benke, M.: A tool for automated theorem proving in agda. In: Proceedings of the 2004 International Conference on Types for Proofs and Programs. pp. 154–169. TYPES'04, Springer-Verlag, Berlin, Heidelberg (2006)
16. Malecha, G., Chlipala, A., Braibant, T.: Compositional computational reflection. In: Interactive Theorem Proving - 5th International Conference, ITP 2014. pp. 374–389 (2014)
17. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
18. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: a monad for typed tactic programming in coq. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'13. pp. 87–100 (2013)