Frex: dependently-typed algebraic simplification

GUILLAUME ALLAIS and EDWIN BRADY, University of St. Andrews
NATHAN CORBYN, University of Oxford
OHAD KAMMAR, University of Edinburgh
JEREMY YALLOP, University of Cambridge

We present an extensible, mathematically-structured algebraic simplification library design. We structure the library using universal algebraic concepts: a free algebra; and a free extension — frex — of an algebra by a set of variables. The library's dependently-typed API guarantees simplification modules, even user-defined ones, are terminating, sound, and complete with respect to a well-specified class of equations. Completeness offers intangible benefits in practice — our main contribution is the novel design. Cleanly separating between the interface and implementation of simplification modules provides two new modularity axes. First, simplification modules share thousands of lines of infrastructure code dealing with term-representation, pretty-printing, certification, and macros/reflection. Second, new simplification modules can reuse existing ones. We demonstrate this design by developing simplification modules for monoid varieties: ordinary, commutative, and involutive. We implemented this design in the new Idris2 dependently-typed programming language, and in Agda.

CCS Concepts: • **Theory of computation** → **Type theory**; **Constructive mathematics**; **Equational logic and rewriting**; **Automated reasoning**; *Categorical semantics*; *Algebraic semantics*; • **Software and its engineering** → Formal software verification; Functional languages; • **Mathematics of computing** → Solvers; • **Computing methodologies** → Representation of polynomials.

Additional Key Words and Phrases: dependent types, frex, free extension, mathematically structured programming, universal algebra, algebraic simplification, homomorphism, universal property

1 INTRODUCTION

Dependently-typed programming enables ever stronger program invariants. With some creativity, users can maintain such invariants throughout the program either implicitly by computation or canonically by adding or peeling constructors off of an inductive structure. Ideally, dependent-types provide frictionless programming where invariants, computation, types, and terms line up just right. The traditional textbook vector-append exemplifies this ideal. A vector is a list whose type is indexed by a natural number maintaining its length. Vector append returns a result of length `n + m`, the sum of its inputs' respective lengths (`n` and `m`). When addition on natural numbers recurses on the left argument, two crucial equations hold true by computation alone: `0+m = m` and `(1+n)+m = 1+(n+m)`. These equations are precisely the ones maintaining the vector append invariant in the base and inductive cases. As a consequence the dependently typed code is frictionless — putting the type signatures aside, vector append looks exactly like its loosely-specified list variant:

```
(++) : (xs, ys : List a) -> List a          (++) : Vect n a -> Vect m a -> Vect (n+m) a
[]        ++ ys = ys                        []        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)           (x :: xs) ++ ys = x :: (xs ++ ys)
```

However, this ideal frictionless style is often impossible for general-purpose dependently-typed programming. Maintaining the invariant necessitates algebraic simplification and equational reasoning. For example, consider the binary merge function in Fig. 1, used in merge-sorting, which we adapted from the standard library of the programming language Idris2 [10]. Its structure is identical to its list counterpart, but it requires additional rewriting steps to maintain the vector-length invariants. It combines the two input vectors, of lengths `n` and `m`, into a vector of length `n + m`. In

the first base case (line 3), the type-checker automatically normalises the index `0+m` and accepts the result `m`-vector `ys`. In both the remaining cases, we use rewriting to maintain the invariants:
Base step (lines 4–7): `xs` is an `n`-vector instead of the expected `(n+0)`-vector; and
Inductive step (lines 8-15): `mergeBy` on line 13 returns a `((1+n)+m)`-vector instead of an `(n+(1+m))`-vector.
We use the two auxiliary lemmata (lines 7 and 14–15) from the standard library's `Data.Nat` to discharge these obligations, and the program checks. Index rearrangement doesn't help: changing the result type to `Vect (m+n) a` requires rewriting in lines 3 and 11 instead. Rewriting is unavoidable.

```
mergeBy : (isLT: a -> a -> Bool) ->                    1
  Vect n a -> Vect m a -> Vect (n + m) a               2
mergeBy      _ [] ys = ys                              3
mergeBy {n} _ xs [] =                                  4
  rewrite plusZeroRightNeutral n in xs                 5
  where                                                6
    plusZeroRightNeutral : (n : Nat) -> n + 0 = n      7
mergeBy {n = S n} {m = S m}                            8
  isLT xs'@(x :: xs) ys'@(y :: ys) =                   9
  if isLT x y                                          10
  then x :: mergeBy isLT xs ys'                        11
  else rewrite sym (plusSuccRightSucc n m) in          12
       y :: mergeBy isLT xs' ys                        13
  where plusSuccRightSucc : (n,m : Nat) ->             14
         S (n + m) = n + S m                           15
```

Fig. 1. Merging vectors in Idris2 (from `Data.Vect`)

This essential need of rewriting is the reason that dependently-typed languages and their ecosystems include algebraic simplifiers for common algebraic structures: commutative/ordinary monoids, semi-rings, rings, etc. Users then only need to establish the structures' axioms, such as `plusZeroRightNeutral`, and call simplifiers to discharge derived equations like `plusSuccRightSucc`. In dependently-typed interactive theorem provers, such as Agda, formalisation of algebraic proofs and properties further necessitates algebraic simplifiers. These simplifiers range from tactic-based solutions [6, for example] that simplify the algebraic terms in the typing goals, to simplifiers based on proof-by-reflection that construct propositions that discharge the equation in question [22, e.g.].

We investigate extensible proof-by-reflection simplifier suites. Traditional reflection simplifiers comprise of four parts. A data-structure whose values represent classes of equivalent terms, a function that evaluates terms into these values, an effective procedure, i.e. an algorithm, deciding equivalence of these values, and a proof that the result of this decision procedure reflects that the original terms are equal. For example, take a 2-variable term over the additive integers (left), its representative value (middle) representing the simplified form (right):[1]

$$\texttt{-6 + (x + 3) + (y + x)} \xmapsto{\text{evaluate/reflect}} \texttt{(-3, [2, 1]) : (Integer, Vect 2 Nat)} \xmapsto{\text{reify}} \texttt{-3 + 2*x + 1*y}$$

The representation pairs the sum of the concrete integers with a vector of coefficients for the two variables, and one can prove that equality of representatives implies propositional equality.

We specify a uniform interface to such simplifiers. This uniform characterisation sets this approach apart from existing libraries. The suite shares infrastructure code, and the separation between interface and implementation allows library designers and users to reuse existing simplifiers to extend the library with new simplifiers. Simplifiers use many different data-structures, and a uniform simplifier library needs enough abstraction to accommodate these differences.

Recently, Yallop et al. [38] observed that many data-structures for optimising and partially-evaluating code involving static and dynamic fragments share a universal property: they implement a representation of a free extension (frex) of an algebraic structure with a set of variables. We show

[1]The terminology 'reify' and 'reflect' sometimes seems conflicting, but follows the relationship: syntax $\underset{\text{reify}}{\overset{\text{reflect}}{\rightleftarrows}}$ semantics. Reifying a program (semantics) gives a deeply-embedded representation (syntax) which one can manipulate. When we manipulating two values, as we do here, reifying normal forms (semantics) gives deeply-embedded terms (syntax).

that this observation applies to algebraic simplifiers — the data structures involved in reflection-based simplifications also represent a frex. By recognising the theory and algebra for this frex, we explicate the equations the simplifier is sound and complete with respect to.

Contribution. We investigate a new design for dependently-typed algebraic simplification suites. Our design philosophy is to take the implementation beyond algebra*ic* simplification, and teach it *algebra*: signatures, theories and models, homomorphisms, and universal properties. The resulting solvers are sound and complete by construction, and the library possesses several core software engineering properties: extensibility, modularity, iterative development, and proof extraction to avoid unnecessary dependencies, all realised completely generically. We implemented this design in two dependently-typed languages, Agda and Idris2. We also investigated an additional layer of reflection to ease usage. This layer requires substantial development effort and heuristics, and provides real advantages in Agda and limited advantages in Idris2. We provide a preliminary quantitative and qualitative usability evaluation, demonstrating the design is viable for interactive development. We also evaluate the library's extensibility. This evaluation exercise led us to proving new representation theorems for free involutive algebras and their extensions.

Structure. We proceed as follows. Sec. 2–4 are of more introductory nature. Sec. 2 tours our proposed library, Frex, and gives a feel to what it offers. Sec. 3 contains a brief Idris2 tutorial by reviewing setoid-based equational reasoning. Sec. 4 introduces the relevant universal algebra concepts (signatures, equations, algebras) and their representation in Frex. Sec. 5 presents Frex's core and its representations of free algebras and extensions, using our three monoid variations as running examples. Sec. 6 explains the completeness guarantees of the library, and covers proof extraction, simplification, pretty-printing and certification. Both sections are technically involved and are aimed at library designers, and may be skimmed at first reading. Sec. 7 concerns a natural question: can one use reflection/macros to invoke Frex automatically? The answer is a qualified 'yes', requiring much library-developer effort. Sec. 8 gives a quantitative and qualitative evaluation of Frex. Sec. 9 discusses system design issues that Frex brought up during development. Secs. 10 and 11 conclude and discuss related and further work.

The theory behind Frex is well-established [38], and the main novelty and innovation is in its implementation in a dependently-typed language. The proof of this kind of pudding is in the eating, and we therefore include implementation-code demonstrating the programmatic realisation of Frex's algebraic concepts. We emphasise however that this manuscript is not a literate program: Frex consists of 9,500 lines of Idris2 code. We include only the code we believe is important for the gist of the ideas and concepts, and leave the implementation to speak for itself.

2 OVERVIEW

We propose Frex: a library design for algebraic simplifier suites. Frex takes advantage of the expressive power of dependent type systems and structures the simplifiers around the notions of *free algebras* and *free extensions (frex)*. Each simplifier — *frexlet* — implements the data-structures needed for algebraic simplification, and mechanised proofs that they satisfy the specification of the free algebra or the free extension. In return, Frex provides the following capabilities. To give a flavour of the code involved, we include full Idris2 code listings. The following sections will explain all the syntax and concepts involved.

```
MonoidTheory : Presentation
MonoidTheory = MkPresentation Theory.Signature
                                    Theory.Axiom $ \case
  LftNeutrality => lftNeutrality Neutral Product
  RgtNeutrality => rgtNeutrality Neutral Product
  Associativity => associativity         Product
```

Generic, uniform infrastructure for algebraic reasoning. Frex provides common simplifier code, such as algebras over a signature, equational axioms and presentations, validity, provability, etc.

```
simplify : (n, m, k : Nat) ->
  (n + 6) + (k + n) + (m + 2) = k + 2*n + m + 8
simplify n m k = solve 3 (Monoid.Commutative.Frex Nat.Additive)
  $ (Dyn 0 .+. Sta 6) .+. (Dyn 1 .+. Dyn 0) .+. (Dyn 2 .+. Sta 2) =-=
  Dyn 1 .+. ((the Nat 2) *. Dyn 0) .+. Dyn 2 .+. Sta 8
```

Fig. 2. Discharging equations with free extensions

For example, the library's predefined monoid frexlet on the right uses predefined axiom schemes for neutrality and associativity.

```
%hint
monoidNotation : (a : Monoid) -> NotationHint a Additive1
monoidNotation a = a.notationHint Additive1 a.Additive1

infix 0 ~~
0 (~~) : (monoid : Monoid) => (lhs, rhs : U monoid) -> Type
(~~) = monoid.equivalence.relation
```

Frex provides flexible notation suites such as `Additive1`, `Additive2`, or `Multiplicative1`, `Multiplicative2`, etc. They provide an additive/multiplicative infix binary operator symbol ((`.+.`), (`:+:`), or (`.*.`), (`:*:`), etc.) and an additive/multiplicative neutral constant (`O1`, `O2`, or `I1`, `I2`).[2] For example, the two declarations on the left let us use `(O1 .+. (a .+. O1)) .+. O1 ~~ a` to represent the equation $0 + (a + 0) + 0 = a$. Other languages can use similar idiomatic mechanisms like unification hints [3] or first-class type-classes [35], or modules in Agda, to produce more ergonomic interfaces to the generic infrastructure. The zero (`0`) preceding the declaration of the notation `(~~)` marks this function as erased-at-runtime, one of Idris2's quantitative type theoretic features [4, 10].

Soundness and completeness. Like other algebraic simplifiers, Frex soundly discharges algebraic equations. Unlike other simplifiers, it supports two kinds of simplifiers, based on two related concepts from universal algebra:

Free algebra (fral) simplifiers discharge equations that hold in all algebras, as on the right. Users call the simplifier using `solve`, passing as arguments the number of free variables (`1`), the relevant fral simplifier over the free variables (`FreeMonoidOver`), and the equation to discharge with simplification. The smart constructor `X` gives the term representing a variable.

```
units : {monoid : Monoid} -> {a : U monoid} ->
        (O1 .+. (a .+. O1)) .+. O1 ~~ a
units = solve 1 (FreeMonoidOver (cast $ Fin 1))
             $ (O1 .+. (X 0 .+. O1)) .+. O1 =-= X 0
```

Free extension (frex) simplifiers additionally evaluate closed sub-terms in concrete algebras (Fig. 2). Like the free algebra simplifier, we pass the number of free variables and the relevant frex simplifier. But unlike the fral and other existing simplifiers, here the simplification may contain both free variables, labelled `Dyn`amic, and statically-known, concrete values, labelled `Sta`tic, following the terminology of Yallop et al. [38]. Using a different syntax than the fral's `X` avoids some confusing ambiguity-resolution error-messaged in Idris2 in case of a type-error. The frex simplifiers group these concrete values together, and the type-checker can evaluate these concrete values.

By expressing the universal property of the fral and frex as a dependent-type, Frex guarantees that well-typed simplifiers are complete, and will discharge all provable equations.

Extensibility. Frex exports the universal properties of the fral and frex, and library users may implement their own frexlets. Since the type-system enforces these universal properties, user-defined frexlets are also sound and complete. To test this extensibility, we implemented a new frexlet for involutive monoids, monoids with an additional involutive unary operator that reverses the monoid multiplication. It took 1 experienced developer 2 weeks to develop.

[2] The familiar additive notation (`+`) and `0` clashes too much with Idris2's current numeric tower and overloading mechanisms.

Modular and iterative development. Frexlets share the core Frex infrastructure, so frexlet developers can reuse and combine existing frexlets. For example, the involutive monoids frexlet uses the free extension `MonoidFrex (cast a) (cast Bool `Pair` s)` of the underlying monoid with two copies of the set of variables, including a Boolean tag tracking whether the variable is involuted or not. By appealing to its universal property, the frexlet designers avoid the low-level calculations involved in constructing and manipulating normal forms.

We can also use frals to construct frexes and vice versa. For example, the `ByFrex` construct (see type on the right) constructs fral simplifiers from a frex simplifier for the initial algebra (fral over the empty variable set). We can use this construction to get the monoid fral. We can rapidly develop new frexlets with combinators such as `ByFrex`, and in the future iteratively improve them by fusing abstractions and streamlining data-structures.

```
ByFrex : (initial : Free pres (cast Void)) ->
  Frex initial.Data.Model s -> Free pres s
FreeMonoidOver : (s : Setoid) -> Free MonoidTheory s
FreeMonoidOver s = ByFrex FreeMonoidVoid
                          (MonoidFrex TrivialMonoid s)
```

Proof extraction. As we'll see later, Frex formalises the universal property with respect to *setoid* algebras, and not just algebras that satisfy the equations propositionally. Using setoids complicates the core of Frex, generalising the definitions and proofs to work with the additional equivalence relations. In return we can use the same core interface to extract the equality proofs frexlets compute. The appendix (Fig. 15) shows an automatically extracted proof for the equation $(x \bullet 3) \bullet 2 = 5 \bullet x$ in the additive monoid structure (`Nat`, 0, (+)), using the code on the right. The synthesised proof has 24 steps. While longer than necessary, it is extracted completely generically through recourse to frex's universal property. Concretely, we implement the functions `solve` and `prove` for both the fral and the frex by calling the same function with different parameters: we call `freeSolve` to implement fral simplification and proof synthesis, and we call `frexify` for the frex counterparts. We provide extraction to unicode and LaTeX.

```
extractedProof :
  ((Dyn' 0 :+: Sta' 3) :+: Sta' 2 ~~ Sta' 5 :+: Dyn' 0)
  {vars = Fin 1}
extractedProof
  = Frex.prove _ (Monoid.Commutative.Frex Nat.Additive)
  $ ((Dyn 0 :+: Sta 3) :+: Sta 2 =-= Sta 5 :+: Dyn 0)
```

Certification. We also provide a generic mechanism to compile the extracted deeply embedded proofs into Idris2 modules that are independent of Frex. These modules can be type-checked separately and provide certificates. The appendix (Fig. 16) shows an automatically extracted certificate for the equation $0 + (x + 0) + 0 = x$ in a generic monoid `m` = (`U m`, `O1`, `(.+.)`). We produce the certificate by invoking the function `idris : List (String, Lemma MonoidTheory) -> String`, which takes a list of named lemmata and generates this module. To make the generated code more readable, concrete frexlets such as monoids specialise the generic certification mechanism to support infix notation like addition or multiplication.

3 SETOIDS AND EQUATIONAL REASONING: AN IDRIS2 TUTORIAL

To introduce the relevant features of Idris2, we review some relevant standard constructions in dependent types [18, 19, e.g.]. A *setoid* `X` = (`U X`, `(~~)`) consists of a set `U X` and an equivalence relation `(~~)`. We represent equivalence relations and setoids in Idris2 with *records* in Fig. 3a. Idris2 records are syntactic sugar for a single-constructor `data` declaration and automatically generated *field projections*, as in Fig. 3b. Idris2 also automatically generates the post-fix projections for each field using a dotted notation, writing `b.equivalence.relation` for the nested projection. The *quantity annotation* `0` on the field `U` means that the compiler will erase these fields at runtime, but such fields may be used in types. Quantities are an integral innovation in Idris2's type theory, and also include

```
record Equivalence (A : Type) where
  constructor MkEquivalence
  0 relation: Rel A
  reflexive : (x       : A) -> relation x x
  symmetric : (x, y    : A) -> relation x y -> relation y x
  transitive: (x, y, z : A) -> relation x y -> relation y z
                            -> relation x z

record Setoid where
  constructor MkSetoid
  0 U : Type
  equivalence : Equivalence U
```

```
data Setoid : Type where
  MkSetoid : (0 U : Type) ->
    (equivalence
      : Equivalence U) -> Setoid

0
U : Setoid -> Type
U (MkSetoid x _) = x
equivalence : (s : Setoid) ->
              Equivalence (U s)
equivalence (MkSetoid _ y) = y
```

Fig. 3. (a) Equivalence relations and setoids as records and (b) example desugaring into GADT and projections

a linear quantity annotation, which we do not use here. If you're reading this manuscript in colour, our listings include semantic highlighting, designating the semantic class of each lexeme: `data` constructor, `type` constructor, `defined` function or value, and `variable` in a binding/bound occurence.

A *setoid homomorphism* `f : X ~> Y` consists of a relation-preserving function between the underlying sets, as on the right. Setoids and their homomorphisms form a common technique to complete an intensional type theory. For example, Fig. 4a defines the quotient of a type by a function , taking two elements to be equal when their images under the function `q` are equal, and the setoid of homomorphisms between two setoids together with extensional equality. This example also demonstrates Idris2's local definitions (lines 19-21), possibly with quantities, named-argument function calls (lines 8–15, e.g.), and anonymous functions (lines 9–10, e.g.). Idris2, like Haskell, implicitly quantifies (with quantity `0`) over unbound variables in type-declarations such as the type `a` in `Quotient`. The underscores indicate that the elaborator can fill-in the blanks uniquely using unification.

```
SetoidHomomorphism : (a,b : Setoid)
  -> (f : U a -> U b) -> Type
SetoidHomomorphism a b f
  = (x,y : U a) -> a.equivalence.relation x y
record (~>) (A,B : Setoid) where
  constructor MkSetoidHomomorphism
  H : U A -> U B
  homomorphic : SetoidHomomorphism A B H
```

This technique is affectionately dubbed 'setoid hell', since we often need to prove that all our functions are setoid homomorphisms. Following Hu and Carette [18], we manage setoid hell by structuring code categorically, organising results into homomorphisms between appropriate setoids. E.g., Fig. 4b presents a setoid over `n`-length vectors over a given setoid. The vector functorial action `VectMap` has a setoid homomorphism structure between the two setoids of homomorphisms: (1) `map f.H` is a homomorphism (lines 19–25), and that (2) it maps extensionally equal homomorphisms to extensionally equal homomorphisms (26–30). These proofs use Idris2's equational reasoning notation for setoids (lines 20–25 and 27–30), a deeply-embedded chain of equational steps. Each step `~~` appeals to transitivity, and requires a justification. The last two dots in the thought bubble operator `(...)` modify the reason: plain usage (line 23) appeals to a setoid equivalence; an equals in the middle dot, e.g. `(.=.)`, appeals to reflexivity via propositional equality (lines 22, 25, 28, 30); and a comparison symbol in the end, e.g. `(.=<)`, appeals to symmetry (lines 25, 30).

```
1  Quotient : (b : Setoid) -> (a -> U b)
2    -> Setoid
3  Quotient b q = MkSetoid a $
4    let 0 relation : a -> a -> Type
5          relation x y =
6            b.equivalence.relation (q x) (q y)
7    in MkEquivalence
8      { relation = relation
9      , reflexive  = \x       =>
10         b.equivalence.reflexive   (q x)
11     , symmetric  = \x,y     =>
12         b.equivalence.symmetric   (q x) (q y)
13     , transitive = \x,y,z   =>
14         b.equivalence.transitive
15           (q x) (q y) (q z)
16     }
17 (~~>) : (a,b : Setoid) -> Setoid
18 (~~>) a b = MkSetoid (a ~> b) $
19   let 0 relation : (f, g : a ~> b) -> Type
20         relation f g = (x : U a) ->
21           b.equivalence.relation (f.H x) (g.H x)
22   in MkEquivalence
23   { relation
24   , reflexive = \f,v        =>
25       b.equivalence.reflexive (f.H v)
26   , symmetric = \f,g,prf,w =>
27       b.equivalence.symmetric _ _ (prf w)
28   , transitive = \f,g,h,f_eq_g, g_eq_h, q =>
29       b.equivalence.transitive _ _ _
30         (f_eq_g q) (g_eq_h q)
31   }
```

```
1  0 (.VectEquality) : (a : Setoid) -> Rel (Vect n (U a))
2  a.VectEquality xs ys = (i : Fin n) ->
3    a.equivalence.relation (index i xs) (index i ys)
3  VectSetoid : (n : Nat) -> (a : Setoid) -> Setoid
4  VectSetoid n a = MkSetoid (Vect n (U a))
5    $ MkEquivalence
6    { relation   = (.VectEquality) a
7    , reflexive  = \xs                    , i =>
8        a.equivalence.reflexive  _
9    , symmetric  = \xs,ys, prf            , i =>
10       a.equivalence.symmetric  _ _    (prf  i)
11   , transitive = \xs, ys, zs, prf1, prf2, i =>
12       a.equivalence.transitive _ _ _ (prf1 i) (prf2 i)
13   }
14 VectMap : {a, b : Setoid} -> (a ~~> b) ~>
15   (VectSetoid n a ~~> VectSetoid n b)
16 VectMap = MkSetoidHomomorphism
17   (\f => MkSetoidHomomorphism
18     (\xs => map f.H xs)
19     $ \xs, ys, prf, i   => CalcWith b $
20     |~ index i (map f.H xs)
21     ~~ f.H (index i xs)
22                               .=.(indexNaturality _ _ _)
23     ~~ f.H (index i ys) ...(f.homomorphic _ _ $ prf i)
24     ~~ index i (map f.H ys)
25                                 .=<(indexNaturality _ _ _))
26   $ \f,g,prf,xs,i => CalcWith b $
27   |~ index i (map f.H xs)
28   ~~ f.H (index i xs) .=.(indexNaturality _ _ _)
29   ~~ g.H (index i xs) ...(prf _)
30   ~~ index i (map g.H xs) .=<(indexNaturality _ _ _)
```

Fig. 4. (a) Quotient, function-space, and (b) vector setoids (top) and a higher-order homomorphism (bottom)

4 UNIVERSAL ALGEBRA IN FREX

```
record Signature where
  constructor MkSignature
  OpWithArity : Nat -> Type
```

```
record Op (sig : Signature) where
  constructor MkOp
  {arity : Nat}
  snd : sig.OpWithArity arity
```

To define an interface to algebraic simplifiers, we first specify and represent algebraic structures. A *signature* $\Sigma = (\texttt{Op}\,\Sigma, \texttt{arity})$ consists of a set $\texttt{Op}\,\Sigma$ of *operation symbols* and an assignment $\texttt{arity} : \texttt{Op}\,\Sigma \longrightarrow \texttt{Nat}$ of a natural number to each operation symbol called its *arity*. For example, the additive signature often used for commutative monoids has two operation symbols: $\texttt{Op}\,\textbf{Additive} := \{(+), 0\}$, with arities 2 and 0, respectively. It's standard to write both symbols and arities more succinctly as $\texttt{Op}\,\textbf{Multiplicative} := \{(\cdot) : 2, 1 : 0\}$, taking as example the multiplicative signature often used for ordinary monoids. In Frex, we use the indexed represention on the right. The implementation uses Idris2's implicit record field for `arity`.

```
(^) : Type -> Nat -> Type
(^) a n = Vect n a
algebraOver : (sig : Signature)
  -> (a : Type) -> Type
sig `algebraOver` a =
  (f : Op sig) -> a ^ (arity f) -> a
record Algebra (Sig : Signature) where
  constructor MakeAlgebra
  0 U    : Type
  Semantics : Sig `algebraOver` U
```

```
CongruenceWRT : {n : Nat} -> (a : Setoid) ->
  (f : (U a) ^ n -> U a) -> Type
CongruenceWRT a f = SetoidHomomorphism (VectSetoid n a) a f
record SetoidAlgebra (Sig : Signature) where
  constructor MkSetoidAlgebra
  algebra : Algebra Sig
  equivalence : Equivalence (U algebra)
  congruence : (f : Op Sig) ->
    (MkSetoid (U algebra) equivalence)
       `CongruenceWRT` (algebra.Sem f)
```

Fig. 5. algebras and setoid algebras in Frex

```
data Operation : Nat -> Type where
  Neutral : Operation 0
  Product : Operation 2
```

Users define concrete instances of `Signature`, such as the signature `MkSignature Operation` for monoids, by defining an injective type family as on the right for the indexed field `OpWithArity`. Injectivity avoids projecting the arity out in concrete cases, letting unification extract it automatically.

Signatures determine an algebraic language, and an algebra is its semantic model. An *algebra* $\mathtt{A} = (\mathtt{U}\,\mathtt{A}, \mathtt{A}\llbracket - \rrbracket)$ for a signature Σ consists of a set $\mathtt{U}\,\mathtt{A}$ called the *carrier* and an assignment of a function $\mathtt{A}\llbracket f \rrbracket : (\mathtt{U}\,\mathtt{A})^n \longrightarrow \mathtt{U}\,\mathtt{A}$ for every operation symbol $f : n$ in Σ. In Frex, we replace $(\mathtt{U}\,\mathtt{A})^n$ with vectors (Fig. 5). As in Haskell, back-ticks turn any name into an infix operator. For example, the additive natural numbers form an algebra for the monoid signature on the left. The code uses the smart constructor `MkAlgebra` that uncurries the argument operations before passing them to `MakeAlgebra`. The `\case` keyword is an anonymous function that immediately pattern-matches its argument. *Setoid algebras* further require an equivalence relation that forms a congruence w.r.t. the operations (Fig. 5).

```
Additive : Algebra Monoid.Theory.Signature
Additive =  MkAlgebra {U = Nat, Sem = \case
  Neutral => 0
  Product => plus}
```

The language determined by a signature consists of *terms* and *equations* in context. Given a set $\mathtt{X}$ of variables, the Σ*-terms over* $\mathtt{X}$ are given inductively as either a variable in $\mathtt{X}$ or an application $f(t_1, \ldots, t_n)$ of an operation symbol $f : n$ from Σ to n terms over $\mathtt{X}$, as on the right. Terms form an algebra, the *free* algebra, with symbols denoting term formers.

```
data Term : (0 sig : Signature) -> Type -> Type where
  ||| A variable with the given index
  Done : {0 sig : Signature} -> a -> Term sig a
  ||| An operator, applied to a vector of sub-terms
  Call : {0 sig : Signature} -> (f : Op sig) ->
         Vect (arity f) (Term sig a) -> Term sig a
Free : (0 sig : Signature) -> (0 x : Type) -> Algebra sig
Free sig x = MakeAlgebra (Term sig x) Call
```

An *equation* $\mathtt{X} \vdash t = s$ consists of a set $\mathtt{X}$ of variables and two terms in context $\mathtt{X}$. Frex only needs equations in a finite context, and we call its cardinality the *support* of the equation. A *presentation* $\mathcal{T} = (\Sigma_{\mathcal{T}}, \mathcal{T}.\mathtt{Axiom})$ consists of a signature $\Sigma_{\mathcal{T}}$ and a set $\mathcal{T}.\mathtt{Axiom}$ of $\Sigma_{\mathcal{T}}$-equations in context:

```
record Equation
  (Sig : Signature) where
  constructor MkEq
  support   : Nat
  lhs, rhs :
    Term Sig (Fin support)
```

```
record Presentation where
  constructor MkPresentation
  signature : Signature
  0 Axiom    : Type
  axiom      : (ax : Axiom) ->
    Equation signature
```

```
associativity : {sig : Signature}         1
  -> EqSpec sig [2]                       2
associativity product =                   3
  let (+) = call product in               4
  MkEquation 3 $ X 0 + (X 1 +  X 2)       5
                 =-= (X 0 +  X 1) + X 2   6
```

For example, the monoid presentation **Monoid** in Fig. 6 has three axioms: left and right neutrality, and associativity. Frex defines a generic collection of axiom schemes (above, right). Its type

```
data Axiom
  = LftNeutrality
  | RgtNeutrality
  | Associativity
```

```
MonoidTheory : Presentation
MonoidTheory = MkPresentation Theory.Signature
                                   Theory.Axiom $ \case
    LftNeutrality => lftNeutrality Neutral Product
    RgtNeutrality => rgtNeutrality Neutral Product
    Associativity => associativity         Product
```

```
MonoidStructure : Type
MonoidStructure =
  SetoidAlgebra Signature
Monoid : Type
Monoid = Model MonoidTheory
```

Fig. 6. Axiomatising monoids in Frex

```
models : {sig : Signature} ->
  (a : SetoidAlgebra sig) -> (eq : Equation sig) ->
  (env : Fin eq.support -> U a.algebra) -> Type
models a eq env = a.equivalence.relation
    (a.Sem eq.lhs env)
    (a.Sem eq.rhs env)
(=|) : {sig : Signature} -> (eq : Equation sig) ->
  (a : SetoidAlgebra sig
     ** Fin eq.support -> U a.algebra) -> Type
eq =| (a ** env) = models a eq env
```

```
ValidatesEquation : (eq : Equation sig) ->
  (a : SetoidAlgebra sig) -> Type
ValidatesEquation eq a =
  (env : Fin eq.support -> U a.algebra) ->
  eq =| (a ** env)

Validates : (pres : Presentation) ->
  (a : SetoidAlgebra pres.signature) -> Type
Validates pres a = (ax : pres.Axiom) ->
  ValidatesEquation (pres.axiom ax) a
```

Fig. 7. Equational validity in an algebra

`EqSpec sig [2]` (lines 1–2) states that it's a scheme involving a single binary operation, and its declaration involves 3 variables (`MkEquation 3` in line 5).

A Σ-equation `X` $\vdash t = s$ is *valid* in a Σ-algebra `A` when the `A`-interpretations of both sides are extensionally equal. Frex's representation of this statement is in Fig. 7. We use Idris2's dependent pairing construct to pair an algebra with an environment in the standard entailment syntax `eq =| (a ** env)`. The code on the right validates the monoid axioms for our running example.

```
IsMonoid : Validates MonoidTheory NatAdditive
IsMonoid LftNeutrality env = Refl
IsMonoid RgtNeutrality env = plusZeroRightNeutral _
IsMonoid Associativity env = plusAssociative _ _ _
```

A $\mathcal{T}$-*model* `A` is a $\Sigma_{\mathcal{T}}$-algebra `A` validating all $\mathcal{T}$-equations:

```
record Model (Pres : Presentation) where
  constructor MkModel
  Algebra  : SetoidAlgebra (Pres).signature
  Validate : Validates Pres Algebra
```

We can now define a monoid to be a **Monoid**-model, as in Fig. 6. Putting everything together, we validate the monoid structure of multiplication on the right. Line 3 converts the constructed algebra into a setoid algebra, and lines 10–12 use results about the natural numbers from Idris2's standard library.

```
Multiplicative : Monoid                                         1
Multiplicative = MkModel                                        2
  { Algebra = cast {from = Algebra Signature} $                 3
      MkAlgebra {U = Nat, Sem = \case Neutral => 1              4
                                      Product => mult}          5
  , Validate = \case                                            6
      LftNeutrality => \env => plusZeroRightNeutral _           7
      RgtNeutrality => \env => multOneRightNeutral _            8
      Associativity => \env => multAssociative _ _ _            9
  }                                                             10
```

Using Frex

```
ListInvMonoid : {0 a : Type} -> InvolutiveMonoid
ListInvMonoid = MkModel
  { Algebra = cast $ MkAlgebra
      {sig = Monoid.Involutive.Theory.Signature}
      { U = List a                                        -- Carrier
      , Sem = \case                                       -- Operations
          Mono monoidOp => case monoidOp of               --   Inherited from monoids
               Neutral => []
               Product => (++)
          Involution => reverse
      }
  , Validate = \case                                      -- Validate equations
      Mon LftNeutrality  => \env => Refl                  -- Directly, or
      Mon RgtNeutrality  => \env => appendNilRightNeutral _  -- use existing standard
      Mon Associativity  => \env => appendAssociative _ _ _  -- library functions
      Involutivity       => \env => reverseInvolutive _
      Antidistributivity => \env => sym (revAppend _ _)
  }
```

Fig. 8. The involutive monoids of list reversal

While the definitions in this section are layered and structured, they generalise familiar situations concerning monoids and groups that are usually covered by computer science curricula. We hope users can pick up a working knowledge by modifying such examples.

Unless they're already working abstractly with an algebraic structure, we expect that in practice users start by recognising their concrete algebra validates the axioms of an existing frexlet they want to use. As a concrete running example, we'll take computations with lists that also involve the `reverse` function. These form an *involutive* monoid: a monoid `A` equipped with a unary *involution* operator $x \mapsto \overline{x} : \mathtt{UA} \longrightarrow \mathtt{UA}$ satisfying two axioms $\overline{\overline{\mathtt{x}}} = \mathtt{x}$ and $\overline{\mathtt{xy}} = \overline{\mathtt{y}}\,\overline{\mathtt{x}}$. We then equip our type of interest, lists, with an involutive model structure as in Fig. 8. We can use this algebra and the involutive monoid to discharge equations containing list variables and concrete lists:

```
1  lemma : {x,y : List a} -> (i,j,k : a)
2    -> (reverse ([j, i] ++ reverse y ++ ([] ++ reverse x))) ++ [k]
3     = x ++ y ++ [i, j, k]
4  lemma i j k = solve 2 (Involutive.Frex.Frex ListInvMonoid) $
5   ((Sta [j, i] .*. (Dyn 1) .inv .*. (I1 .*. (Dyn 0) .inv)) .inv) .*. Sta [k]
6    =-= Dyn 0 .*. (Dyn 1 .*. Sta [i, j, k])
```

The `solve` function takes as argument the number of variables (`n=2` on line 2) in the algebraic term to simplify, and an algebraic simplifier from the frexlet (`Involutive.Frex.Frex` on line 4). The final argument is a pair of terms with `n=2` variables (`Dyn 0` and `Dyn 1`) and concrete values from the algebra. By importing notation modules the frexlet provides, we can use infix multiplicative notation such as (`.*.`). The type-checker then infers the terms to substitute for each variable.

In this example, we used `solve` to define a stand-alone lemma, but we may also call `solve` directly from a chain of equational reasoning steps. When we extract lemmas, we often want to prove them more abstractly, for *all* involutive monoids. In that case we use a fral:

	free algebra		free extension	
ordinary monoid	variable lists	$yxxyx$	alternating lists in $\mathbb{M}_{2\times 2}(\texttt{Nat})[y]$	$\begin{pmatrix}1 & 3\\ 0 & 2\end{pmatrix} y \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} y \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$
commutative monoid	origin-intercepting linear polynomials	$a_1x_1 + \ldots + a_nx_n$ $(a_i : \texttt{Nat})$	linear polynomials in $\texttt{A}[x_1, \ldots, x_n]$	$c + a_1x_1 + \ldots + a_nx_n$ $(a_i : \texttt{Nat}, c : \texttt{A})$
involutive monoid	lists over ordinary and involuted variables	$y\overline{x}xx\overline{y}x$	alternating lists with tagged variables in $\texttt{String}[x, y]$	`""`x`"hello"`y`"olleh"`$\overline{x}$`""`

Fig. 9. Frexlets for varieties of monoids

```
1  ExampleFral : {a : InvolutiveMonoid} -> (x,y,z : U a)
2    -> let %hint notation : ?                  -- Open notation hints for the monoid
3             notation = a.Notation1            -- for infix operator (.*.) and
4         in a.rel                              -- postfix operator (.inv)
5           (x .*. y.inv .*. z).inv
6           (z.inv .*. y .*. x.inv)
7  ExampleFral x y z =
8    let %hint notation : ?                     -- ditto, but for terms
9          notation = Involutive.Notation.multiplicative1
10   in solve 3 (Involutive.Free.FreeInvolutiveMonoidOver 3) $
11   (X 0 .*. (X 1).inv .*. X 2).inv =-= (X 2).inv .*. X 1 .*. (X 0).inv
```

Lines 2–3 overloads the infix and postfix notation using the frexlet's built-in notation suites. The `solve` function takes the number of free variables and a corresponding fral simplifier (line 10), as well as the two terms representing the equation of interest.

5 FREE EXTENSIONS AND ALGEBRAS

Before delving into the details of Frex's core, Fig. 9 summarises our frexlet representations using examples for elements in the fral and the frex for ordinary, commutative, and involutive monoids.

The elements in the free monoid are lists of the variables appearing in the term, which are sometimes known as reduced words in the context of freely generated groups. The elements in the free extension of a monoid are lists alternating between concrete elements in the given monoid, and freely-adjoined variables. The figure shows an element in the free extension by 1 variable (y) of the multiplicative monoid of 2×2 matrices with natural-number components. The matrix y is unknown, or `Dyn`amically known, and so its occurrence separates the elements in the list.

Further assuming commutativity equates more terms, resulting in the representation of the free commutative monoid over `n` variables as an `n`-vector of coefficients, representing a linear polynomial. Freely extending a commutative monoid `A` by `n` variables can be represented by a concrete coefficient c : `A` together with an `n`-vector of coefficients, representing a linear polynomial over `A`.

If we instead include an involutive operation $x \mapsto \overline{x}$ over the monoid, we get reduced words and alternating lists whose letters may be tagged as involuted. The figure demonstrates the free extension of the monoid structure of `String` concatenation, with string reversal for the involution.

These examples feel similar, using a notion of a polynomial with coefficients taken from a concrete algebra. The underlying representations are natural, and appear in existing algebraic simplifiers. Frex innovates by exploiting the formal commonality of these examples — the universal property of free algebras and free extensions — when designing simplifier libraries.

```
Preserves : {sig : Signature}
  -> (a, b : SetoidAlgebra sig)
  -> (h : U a -> U b)
  -> (f : Op sig) -> Type
Preserves {sig} a b h f
  = (xs : Vect (arity f) (U a))
    -> b.equivalence.relation
         (h $ a.Sem f xs)
         (b.Sem f (map h xs))
```

```
Homomorphism : {sig : Signature}
  -> (a, b : SetoidAlgebra sig) -> (h : U a -> U b) -> Type
Homomorphism a b h = (f : Op sig) -> Preserves a b h f


record (~>) {Sig : Signature} (a, b : SetoidAlgebra Sig) where
  constructor MkSetoidHomomorphism
  H : cast {to = Setoid} a ~> cast b
  preserves : Homomorphism a b (.H H)
```

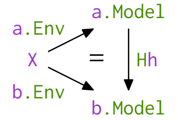
Fig. 10. Setoid algebra homomorphisms in Frex

5.1 Universal properties

In order to talk about 'free' algebras, extensions, and universal properties in general, we need categories of algebras, not just types of them. We then proceed to define the appropriate structures for the fral and the frex and its structure-preserving functions. Then by cranking a handle, we get the definition of the free such structure. The creativity that goes into designing simplifiers becomes more methodological and principled when recast as designing an appropriate fral or frex. The universal property provides a checklist that organises the simplification code.

A *homomorphism* h : `A` $\longrightarrow$ `B` of Σ-algebras is a semantics-preserving function `H`h : `U A` $\longrightarrow$ `U B` between their carriers. Explicitly, for all operation symbols $f : n$ in Σ and $a_1, \ldots, a_n$ in `U A`, we have: `H`h(`A`$[\![f]\!](a_1, \ldots, a_n)$) = `B`$[\![f]\!]$(`H`$h\, a_1, \ldots,$ `H`$h\, a_n$)). Frex extends this notion to setoid algebras in Fig. 10, by requiring the underlying function to be a setoid homomorphism between the corresponding setoids. The code uses an appropriate `cast` function that assembles these setoids from the data in each algebra. Each `a : Algebra sig` defines a homomorphic extension operator `a.Sem : Term sig x -> (x -> U a) -> U a` by structural induction over the term (i.e., folding). For example, `(Nat.Additive).Sem (X 0.+.O1.+.X 1) (\case {0=>5; 1=>7})` evaluates to `5+0+7` in the `Additive Nat` algebra. The free algebra construction, together with the embedding of variables into terms, forms the left adjoint to the forgetful functor from algebras to sets by the uniqueness of this homomorphic extension. Being left-ajoint to the forgetful functor is the category-theoretic definition of the free algebra, justifying the terminology.

Given a presentation $\mathcal{T}$, a $\mathcal{T}$*-algebra* `a` = (`a.Model`, `Env a`) *over a set* `X` consists of a $\mathcal{T}$-algebra `a.Model` and a function `Env a : X` $\longrightarrow$ `Ua.Model`. A *morphism* `h` : `a` $\longrightarrow$ `b` of such algebras is a $\mathcal{T}$-algebra homomorphism that moreover makes the diagram on the right commute. Similarly, given a $\mathcal{T}$-algebra `A`, an *extension* `a` = (`a.Model`, `a.Var`, `a.Embed.H`) of `A` by a set `X` is a triple consisting of a $\mathcal{T}$-algebra `a.Model`, a function `a.Var : X` $\longrightarrow$ `Ua.Model`, and a $\mathcal{T}$-homomorphism `a.Embed.H : A` $\longrightarrow$ `a.Model`, and morphisms of extensions are $\mathcal{T}$-homomorphisms making the diagram on the left commute. Fig. 11 presents the corresponding Frex declarations for algebras over a setoid and extensions. It expresses the equations in the commuting diagrams using the extensionality equivalence relation on the function-space setoid from Fig. 4b and the power of an algebra by a setoid, which we define in this section.



a.Model
a.Var a.Embed.H
X = Hh = UA
b.Var b.Embed.H
b.Model

The *free* algebra over a set (fral) and the *free* extension (frex) of an algebra by a set is then the initial such structure: there is a unique structure-preserving map from the free structure to every structure. This succinct definition, while standard, packs much structure. By way of introduction, we'll unpack it for the free commutative monoid over `Fin n`, the finite set with `n` elements.

```
record ModelOver
    (Pres : Presentation)
    (X : Setoid) where
  constructor MkModelOver
  Model : Model Pres
  Env : X ~> cast Model

PreservesEnv : {Pres : Presentation}
    -> {X : Setoid}
    -> (a, b : Pres `ModelOver` X) ->
    (cast {to = Setoid} a.Model
      ~> cast b.Model) -> Type
PreservesEnv a b h =
  (X ~~> cast b.Model).equivalence.relation
    (h . a.Env) b.Env

record (~>)
    {Pres : Presentation} {X : Setoid}
    (A, B : Pres `ModelOver` X) where
  constructor MkHomomorphism
  H : (A .Model) ~> (B .Model)
  preserves : PreservesEnv A B (H .H)
```

```
record Extension {Pres : Presentation}
    (A : Model Pres)(X : Setoid) where
  constructor MkExtension
  Model : Model Pres
  Embed : A ~> Model
  Var   : X ~> cast Model
record (~>) {Pres : Presentation}
      {A : Model Pres} {X : Setoid}
    (Extension1, Extension2 : Extension A X) where
  constructor MkExtensionMorphism
  H : (Extension1).Model ~> (Extension2).Model
  PreserveEmbed :
    (cast A ~~> (Extension2).Model)
      .equivalence.relation
        (H . (Extension1).Embed)
             (Extension2).Embed
  PreserveVar   :
    (X ~~> cast (Extension2).Model)
      .equivalence.relation
        ((H).H . (Extension1).Var)
                 (Extension2).Var
```

Fig. 11. Structure and its preservation for (a) algebras over a setoid, and (b) extensions of an algebra

First, we designate a commutative monoid for the model structure in the fral. In Fig. 11, we mentioned the carrier consists of origin-intercepting linear polynomials with `Nat` coefficients $p = a_1x_1 + \ldots + a_n x_n$, which we represent with `n`-tuples of natural numbers and pointwise addition:

```
Carrier : (n : Nat) -> Setoid
Carrier n = VectSetoid n
                (cast Nat)
```

$$0 := 0x_1 + \ldots + 0x_n$$
$$:= \texttt{[0,}\ldots\texttt{,0]}$$
$$= \texttt{replicate n 0}$$

$$p+q := (a_1 + b_1)x_1 + \ldots + (a_n + b_n)x_n$$
$$:= \texttt{[}a_1\texttt{+}b_2\texttt{,}\ldots\texttt{,}a_n\texttt{+}b_n\texttt{]}$$
$$= \texttt{map (uncurry (+)) (zip as bs)}$$

Call the resulting algebra `Model n : CommutativeMonoid`. For the `Env` component, use tabulation to define `unit n : Fin n` $\longrightarrow$ `Carrier n`, with `1` in the position corresponding to the argument and `0` elsewhere. The initiality of this structure follows from the normal form property — every origin-intersecting linear polynomial p can be represented as $p = \sum_{i=1}^{n} a_i \cdot \texttt{dirac}\ \texttt{i}$:

$$\texttt{unit n } i := 1x_i$$
$$:= \texttt{[0,}\ldots\texttt{,0,1,0,}\ldots\texttt{,0]}$$
$$= \texttt{tabulate \$ dirac i}$$

where[3]:

$$\texttt{dirac i j} := \begin{cases} \texttt{i} = \texttt{j} : & \texttt{1} \\ \texttt{i} \neq \texttt{j} : & \texttt{0} \end{cases}$$

```
normalForm : (n : Nat) -> (xs : U (Model n)) -> xs =
  (Model n).sum (tabulate $ \i => (index i xs) *. (unit n i))
```

Since monoid homomorphisms preserve the summation and multiplication-by-a-natural, the unique structure preserving map `h : (Model n, unit n)` $\longrightarrow$ `a` is this homomorphism:

```
h xs = a.Model.sum (mapWithPos (\i,k => k *. a.Env.H i) xs)
```

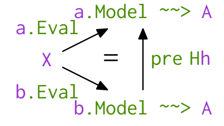
[3]This function is in fact Kronecker's delta, but the shorter name Dirac's delta seems more familiar to readers.

This standard argument lies behind many simplifiers, as well as more advanced techniques like normalisation-by-evaluation. FREX takes the same approach, but also explores how to use general-purpose constructions involving frals and frexes, and bespoke facts about algebraic structures, to construct new frals and frexes.

We can now implement the `solve` functions. The input is a proof that the interpretations of both sides of the equation are equal in the fral or the frex. From such a proof we can conclude that their images under every homomorphism out of the fral or the frex are equal, in particular under the unique homomorphism into the input algebra. This proof is what the type of `solve` guarantees.

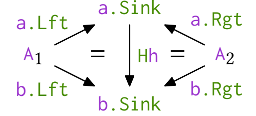
5.2 Powers

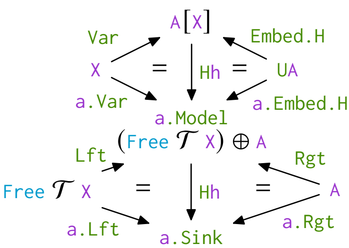
The commutative monoid structure `Model n` instantiates a general construction: $\mathcal{T}$-algebras have powers by setoids. The *power* of an algebra `A` by a set(oid) `X` is the terminal *parameterisation*. Parameterisations, shown succinctly in the diagram on the right, are an `X`-indexed collection of algebra homomorphisms `a.Eval f` : `a.Model` ⟶ `A`. Requiring `a.Eval f` to be homomorphic implies that operations are given pointwise. The structure preservation uses the contravariant action `pre Hh` precomposing a homomorphism `Hh` : `a.Model` ⟶ `b.Model`. Universality singles out the carrier of the power as the function-space `X ~~> a.Model`. For `X` = `Fin n`, we can represent it by `n`-tuples from `U A`.



5.3 Frex via coproducts with fral

The fral and the frex relate: the free extension of `A` by `X` is the *coproduct* of `A` with the free algebra over `X`. Coproducts are the initial *cospans*, showed succinctly on the right. A cospan consists of two homomorphisms with a shared codomain. All algebras have coproducts, but these may be difficult to represent. However, in some cases such as commutative monoids, the coproduct is particularly straightforward to represent: its carrier is the cartesian product of the component carriers.





The universal property of the frex `A[X]` combines those of the fral `Free` $\mathcal{T}$`X` and its coproduct with `A`. The fral's universality equates the left triangles in the two diagrams, and construct:

```
CoproductAlgebraWithFree pres a x : (free : Free pres x) ->
  (coprod : Coproduct a free.Data.Model) -> Frex a x
```

For commutative monoids it gives the commutative monoid of linear polynomials with natural numbers as degree-1 coefficients whose carrier is represented by `(U A, Vect n Nat)`.

5.4 Fral via an initial algebra frex

Since developing the sound and complete frex can be tedious, there is a generic mechanism for reusing this work to derive a corresponding fral with less effort. This method is based on the following calculation that uses a categorical principle: the free algebra construction preserves initial constructions. Let $\mathbb{O}$ be the *initial* algebra. Since the empty set is the initial set, by this principle, the free algebra on the empty set `Free` $\mathcal{T}\emptyset$ is also the initial algebra. We then calculate the free extension as on the right, constructing the fral from an initial algebra and its frex. This generic construction produces suboptimal representations. For example, the initial monoid is easy to construct: its carrier is the unit type. Freely extending this initial monoid produces alternating lists, that interleave the unit value. Taking variable lists instead

$$\mathtt{Free}\,\mathcal{T}\,\mathtt{X} \cong \mathtt{Free}\,\mathcal{T}(\mathtt{X} + \emptyset) \cong (\mathtt{Free}\,\mathcal{T}\,\mathtt{X}) \oplus (\mathtt{Free}\,\mathcal{T}\emptyset) \cong \mathbb{O}[\mathtt{X}]$$

```
ByFrex : (initial : Free pres (cast Void)) ->
  Frex initial.Data.Model s -> Free pres s
```

leads to a simpler representation, but requires more complicated proofs. So FREX allows us to trade rapid prototyping for efficient representation.

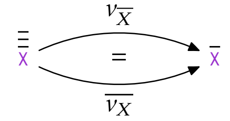
5.5 Reusing frexlets

The final example demonstrates reuse of one simplifier when constructing another. Recall the presentation of involutive monoids from the end of Sec. 4.

PROPOSITION 5.1 (JACOBS). *The free involutive monoid on* `X` *is the free monoid on the product* `(Bool,X)`. *The frex of an involutive monoid by* `X` *is the frex of its underlying monoid by* `(Bool,X)`.

We can prove this proposition directly, establishing the involutive axioms. However, we can phrase this result in much greater generality, and give a higher-level proof, using Jacobs's axiomatisation of involutions [21]. This more abstract proof generalises to other notions of involutive algebras, and we plan to exploit it in the future for generic frexlet reuse. We recount Jacobs's account, albeit in a more advanced categorical jargon, and use it to prove a generic representation theorem for involutive frals and frexes. We don't use this development elsewhere in this manuscript.

Jacobs appeals to the Baez-Dolan *microcosm principle* [5] — an algebraic structure on an object comes from a similar structure on its category — and defines the following concepts. An *involutive structure* on a category C is a pair $(\overline{(-)}, \nu)$ consisting of a functor $\overline{(-)} : C \to C$, called the *involution*, and a natural isomorphism $\nu : \overline{\overline{(-)}} \to -$, called the *involution law*, satisfying the condition on the right.

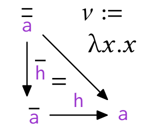


This definition is equivalent to Jacbos's, but reverses the direction of the involution law.

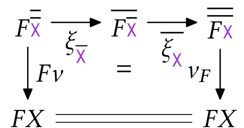
For example, each category has an involutive structure given by the identity functor as involution and the identity natural transformation as the involutive law. This structure, which we call the *trivial* involutive structure, may seem degenerate, but it plays an important role in our development.

The motivating example is **Monoid**, the category of monoids. It has the following non-trivial involutive structure. Given a monoid `a`, construct another monoid $\overline{\texttt{a}}$ with the operation reversed: $\overline{\texttt{a}}[\![\cdot]\!](x, y) := \texttt{a}[\![\cdot]\!](y, x)$. If $\texttt{h} : \texttt{a} \to \texttt{b}$ is a monoid homomorphism, then the same underlying function provides a monoid homomorphism $\overline{\texttt{h}} : \overline{\texttt{a}} \to \overline{\texttt{b}}$. These maps define an involution functor $\overline{(-)} : \textbf{Monoid} \to \textbf{Monoid}$. The identity function is then a monoid isomorphism $\nu := (\lambda x.x) : \overline{\overline{\texttt{a}}} \to \texttt{a}$, the required involution law. We have similar involutive structures on other categories, given by ordinary or commutative: semi-groups, monoids, groups, semirings and rings, and so on.

To see the microcosm principle in action, note that a function $\texttt{h} : \texttt{U a} \to \texttt{U a}$ makes a monoid `a` into an involutive monoid if and only if (1) it is a monoid homomorphism $\texttt{h} : \overline{\texttt{a}} \to \texttt{a}$, so $\texttt{h}(\texttt{x} \cdot \texttt{y}) = \texttt{h y} \cdot \texttt{h x}$, and (2) the diagram on the right commutes, so $\texttt{h}(\texttt{h x}) = \texttt{x}$. These two conditions categorify the notion of an involutive monoid, so we can define it in any involutive category, not just **Monoid**. Jacobs calls these *self-conjugate* objects, and we will study them in more detail soon.



Packaging this structure, an *involutive category* $C = (C_{\text{o}}, \overline{(-)}, \nu)$ is an ordinary category C_{o} equipped with an involutive structure $(\overline{(-)}, \nu)$. An *involutive functor* $F : \mathcal{B} \to C$ between involutive categories is a pair (F_{o}, ξ^F) consisting of an ordinary functor $F_{\text{o}} : \mathcal{B}_{\text{o}} \to C_{\text{o}}$ and a natural transformation $\xi^F : F_{\text{o}}\overline{(-)} \to \overline{F_{\text{o}}(-)}$ called its *distributive law*, satisfying the compatibility condition on the right. Such distributive laws are natural isomorphisms.



The canonical example is the forgetful functor $\texttt{U} : \texttt{Model}\,\mathcal{T} \to \textbf{Set}$ from the category of models of some presentation $\mathcal{T}$ to the category of sets and functions. This functor has an involutive functor structure w.r.t. an involutive structure on $\texttt{Model}\,\mathcal{T}$, when the involution of an algebra only changes the operations of the algebra, but not its carrier. Note the role that the trivial involutive structure

on **Set** plays. All the examples above of monoid varieties and the semi-ring varieties w.r.t. the operation-reversal and trivial involutive structures have such involutive forgetful functors.

An *involutive natural transformation* $\alpha : F \to G$ between two involutive functors is an ordinary natural transformation $\alpha : F_{\text{o}} \to G_{\text{o}}$ between their underlying ordinary functors that moreover satisfies the condition on the right. As Jacobs comments, we therefore have a 2-category **ICat** consisting of involutive categories, functors, and natural transformations, and we may derive involutive adjunctions as two involutive functors and two involutive natural transformations satisfying the triangle laws.

$$\begin{array}{ccc} F\overline{\mathtt{x}} & \xrightarrow{\alpha_{\overline{\mathtt{x}}}} & G\overline{\mathtt{x}} \\ {\scriptstyle \xi^F}\downarrow & = & \downarrow{\scriptstyle \xi^G} \\ \overline{F\mathtt{x}} & \xrightarrow{\overline{\alpha_{\mathtt{x}}}} & \overline{G\mathtt{x}} \end{array}$$

We can turn an ordinary adjunction into an involutive one when one of the functors is involutive:

Proposition 5.2. *Let $G : \mathcal{A} \to C$ be an involutive functor, and $F_{\text{o}} \dashv G_{\text{o}}$ be a left-adjoint to the ordinary functor underlying G with unit η and counit ε. Set $\xi^F_{\mathtt{x}} : F_{\text{o}}\overline{\mathtt{x}} \to \overline{F_{\text{o}}\mathtt{x}}$ to be the mate of the composite $\overline{\mathtt{x}} \xrightarrow{\overline{\eta}} \overline{G_{\text{o}}F_{\text{o}}\mathtt{x}} \xrightarrow{(\xi^G)^{-1}} G_{\text{o}}\overline{F_{\text{o}}\mathtt{x}}$. Then (1) ξ^F equips F_{o} with an involutive functor structure $F : C \to \mathcal{A}$; and (2) $F \dashv G$ is an involutive adjunction with unit η and counit ε.*

As a consequence, the free model functors for models in which the forgetful functor is involutive are all involutive adjunctions. This consequence covers our monoid and semi-ring varieties of interest, namely ordinary and commutative semi-groups, monoids, groups, semi-rings and rings with or without a unit. The distributive laws in these examples are given by the mate of the function: $\lambda x.\eta x : \overline{\mathtt{x}} = \mathtt{x} \xrightarrow{\overline{\eta}=\eta} UF\mathtt{x} \xrightarrow{\xi^U=\lambda t.t} U\overline{F\mathtt{x}}$. One might be tempted to think that the resulting distributive law is the identity homomorphism, because the mate of the unit of an adjunction is the identity function. It is not the case. When we take the mate, we take into account the algebra structure of $\overline{F\mathtt{x}}$, which may change the interpretation of the operations, and consequently changes the resulting mate homomorphism. For the non-trivial involutive structures over monoid and semi-ring varieties, the distributive law will reverse the relevant binary operation.

A *self-conjugate* object $\mathtt{a}$ in an involutive category $\mathcal{A}$ is a pair $(\mathtt{a}_{\mathtt{obj}}, j_{\mathtt{a}})$ consisting of an object $\mathtt{a}_{\mathtt{obj}}$ in $\mathcal{A}$, and an $\mathcal{A}$-morphism $j_A : \overline{\mathtt{a}_{\mathtt{obj}}} \to \mathtt{a}_{\mathtt{obj}}$, satisfying the triangle on the right. As we saw on p. 15, self-conjugate monoids are involutive monoids, and more generally, self-conjugate semi-groups, groups, semi-rings, rings, etc. are the involutive ones. A *homomorphism* $\mathtt{h} : \mathtt{a} \to \mathtt{b}$ of self-conjugate objects is a homomorphism $\mathtt{h} : \mathtt{a}_{\mathtt{obj}} \to \mathtt{b}_{\mathtt{obj}}$ between their underlying objects that moreover satisfies the condition on the left. This condition generalises the usual condition of involutive monoid homomorphisms and so on. Since homomorphisms of self-conjugate objects compose and contain the identities, they form a category which we denote by **SC** $\mathcal{A}$. Jacobs shows that the forgetful functor $\mathtt{U}$: **SC Set** $\to$ **Set** has a left adjoint $\mathtt{F_{SC}}$: **Set** $\to$ **SC Set** sending each set $\mathtt{x}$ to the coproduct of two copies of itself, i.e., by tagging each element with a boolean, and the self-conjugation structure flips this boolean $\mathtt{F_{SC}}\mathtt{x} := ((\mathtt{Bool},\mathtt{x}), \lambda(\mathtt{b},\mathtt{x}).(\neg\mathtt{b},\mathtt{x}))$.

$$\begin{array}{ccc} \overline{\overline{\mathtt{a}}} & & \\ {\scriptstyle \overline{j}}\downarrow & \searrow{\scriptstyle \nu} & \\ \overline{\mathtt{a}} & \xrightarrow{j} & \mathtt{a} \end{array} \qquad \begin{array}{ccc} \overline{\mathtt{a}} & \xrightarrow{\overline{\mathtt{h}}} & \overline{\mathtt{b}} \\ {\scriptstyle j}\downarrow & = & \downarrow{\scriptstyle j} \\ \mathtt{a} & \xrightarrow{\mathtt{h}} & \mathtt{b} \end{array}$$

It will pay-off immediately to include one more level of abstraction. Jacobs (Lemma 6) shows that the **SC** -construction extends to a 2-functor **SC** : **ICat** $\to$ **ICat**. We recall the remaining structure. The action on objects of **ICat** equips the category **SC** $\mathcal{A}$ with an involutive structure, sending each self-conjugate object $\mathtt{a}$ to the self-conjugate object $\overline{\mathtt{a}} := (\overline{\mathtt{a}_{\mathtt{obj}}}, \overline{j_{\mathtt{a}}} : \overline{\overline{\mathtt{a}_{\mathtt{obj}}}} \to \overline{\mathtt{a}_{\mathtt{obj}}})$. The action on the morphisms of **ICat**, sends an involutive functor $F : \mathcal{B} \to C$ to the involutive functor **SC** F : **SC** $\mathcal{B} \to$ **SC** C mapping each self-conjugate object $\mathtt{a}$ to the self-conjugate object $(F_{\text{o}}\mathtt{a}_{\mathtt{obj}}, j_{\mathbf{SC}\,F\mathtt{a}} : \overline{F_{\text{o}}\mathtt{a}_{\mathtt{obj}}} \xrightarrow{(\xi^F)^{-1}} F_{\text{o}}\overline{\mathtt{a}_{\mathtt{obj}}} \xrightarrow{F_{\text{o}}j_{\mathtt{a}}})$, and acting as F_{o} on self-conjugate homomorphisms. The action on 2-cells sends each involutive natural transformations to itself, i.e., a natural transformation between involutive functors also

$$\begin{array}{ccc} \mathbf{SC}\,\mathcal{A} & \xrightarrow{\mathtt{U}} & \mathcal{A} \\ {\scriptstyle \mathbf{SC}\,F}\downarrow & = & \downarrow{\scriptstyle F} \\ \mathbf{SC}\,C & \xrightarrow{\mathtt{U}} & C \end{array}$$

preserves the resulting self-conjugated structure. The forgetful functor $\mathsf{U} : \mathbf{SC}\,\mathcal{A} \to \mathcal{A}$ is then natural as on the left.

We profit off of this obscene level of abstraction immediately: 2-functors preserve all 2-adjunctions, since they transport the triangle equalities to the appropriate triangle equalities. Therefore, if we have an involutive adjunction $F \dashv G : \mathcal{A} \to C$ where C has a free self-conjugate object adjunction $\mathsf{F}^{C}_{\mathsf{SC}} \dashv \mathsf{U} : \mathbf{SC}\,C \to C$, we get the free self-conjugate $\mathcal{A}$-object on X as the composite $F(\mathsf{F}^{C}_{\mathsf{SC}}\mathsf{X})$ completely structurally, as on the right. Applying this result to frals, we get the following generalisation of Prop. 5.1(fral):

$\mathbf{SC}\,\mathcal{A}$ $\xrightarrow{\mathsf{U}}$ $\mathcal{A}$

$\mathbf{SC}\,F \dashv \mathbf{SC}\,G$

$\mathbf{SC}\,C$ $\overset{\text{nat.}}{=}$ $\mathcal{A}$

$\mathsf{F}^{C}_{\mathsf{SC}} \dashv \mathsf{U}$

$C \xleftarrow{G} \mathcal{A}$

Proposition 5.3. *Let $\mathcal{T}$ be a presentation equipped with an involutive structure over* $\mathsf{Model}\,\mathcal{T}$ *and an involutive forgetful functor structure. Let* $(\mathsf{A}, \mathsf{Env})$ *be any free $\mathcal{T}$ model over the product* $(\mathsf{Bool},\mathsf{X})$*. Then the following structure exhibits* A *as the free self-conjugate $\mathcal{T}$-model over* X*:*

$$j_{\mathsf{A}} : \overline{\mathsf{A}} \xrightarrow{\xi^{-1}} \mathsf{A} \xrightarrow{\gg\!=(\neg\times\mathsf{id})} \mathsf{A} \qquad (\gg\!=' \mathsf{f}) : \mathsf{A} \xrightarrow{\gg=\lambda(\mathsf{b},\mathsf{x}).\begin{cases} b = \mathsf{True}: & j_{\mathsf{a}}(\mathsf{f}\,\mathsf{x}) \\ b = \mathsf{False}: & \mathsf{f}\,\mathsf{x}\end{cases}} \mathsf{a}$$

$$\mathsf{Env}' : \mathsf{X} \xrightarrow{\lambda\mathsf{x}.(\mathsf{False},\mathsf{x})} (\mathsf{Bool},\mathsf{X})$$

Having dealt with the fral, we turn to the frex. Jacobs proves that if $(\mathsf{a}_1{+}\mathsf{a}_2, \iota_1, \iota_2)$ is a coproduct of a_1 and a_2 in an involutive category, then $(\overline{\mathsf{a}_1 + \mathsf{a}_2}, \overline{\iota_1}, \overline{\iota_2})$ is a coproduct of $\overline{\mathsf{a}_1}$ and $\overline{\mathsf{a}_2}$. The unique cotupling morphism in the universal property, for each $\mathsf{h}_i : \overline{\mathsf{a}_i} \to \mathsf{b}$, is $\overline{[\mathsf{h}_1, \mathsf{h}_2]} : \overline{\mathsf{a}_1 + \mathsf{a}_2} \xrightarrow{[\overline{\mathsf{h}}_1\circ\nu^{-1}, \overline{\mathsf{h}}_2\circ\nu^{-1}]} \overline{\overline{\mathsf{b}}} \xrightarrow{\nu} \mathsf{b}$. If each a_i has a self-conjugate structure $j_i : \overline{\mathsf{a}_i} \to \mathsf{a}_i$, then the coproduct $\mathsf{a}_1 + \mathsf{a}_2$ has a self-conjugate structure given by $j_1 + j_2 := \overline{[\iota_1 \circ j_1, \iota_2 \circ j_2]} : \overline{\mathsf{a}_1 + \mathsf{a}_2} \to \mathsf{a}_1 + \mathsf{a}_2$. Since the frex $\mathsf{a}[\mathsf{X}]$ can be construct as the coproduct of the model a with the fral on X, we generalise Prop. 5.1(frex):

Proposition 5.4. *Let $\mathcal{T}$ be a presentation equipped with an involutive structure over* $\mathsf{Model}\,\mathcal{T}$ *and an involutive forgetful functor structure, and* a *be a self-conjugate $\mathcal{T}$-model. Let* $(\mathsf{A}, \mathsf{Var}, \mathsf{Embed})$ *be any $\mathcal{T}$-frex of* $\mathsf{a}_{\mathsf{obj}}$ *by the product* $(\mathsf{Bool},\mathsf{X})$*. Then the following structure exhibits* A *as the frex of the self-conjugate $\mathcal{T}$-model* a *by* X*, for* $\mathsf{h} : \mathsf{a} \to \mathsf{c}$ *involutive monoid homomorphism and function* $\mathsf{e} : \mathsf{X} \to \mathsf{c}$*:*

$$j : \overline{\mathsf{A}} \xrightarrow{\overline{\left[\mathsf{a}_{\mathsf{obj}} \xrightarrow{\nu^{-1}} \overline{\overline{\mathsf{a}_{\mathsf{obj}}}} \xrightarrow{\overline{j\mathsf{a}}} \overline{\mathsf{a}} \xrightarrow{\overline{\mathsf{Embed}}} \overline{\mathsf{A}},\ (\mathsf{Bool},\mathsf{X}) \xrightarrow{\neg\times\mathsf{id}} (\mathsf{Bool},\mathsf{X}) \xrightarrow{\mathsf{Var}} \mathsf{U}\,\mathsf{A} \xrightarrow{\xi^{-1}} \mathsf{U}\,\overline{\mathsf{A}}\right]}} \overline{\overline{\mathsf{A}}} \xrightarrow{\nu} \mathsf{A}$$

$$\mathsf{Var}' : \mathsf{X} \xrightarrow{\lambda\mathsf{x}.(\mathsf{False},\mathsf{x})} (\mathsf{Bool},\mathsf{X}) \xrightarrow{\mathsf{Var}} \mathsf{A} \qquad \mathsf{Embed} : \mathsf{a}_{\mathsf{obj}} \xrightarrow{\mathsf{Embed}} \mathsf{A}$$

$$[\mathsf{h}, \mathsf{e}] : \mathsf{A} \xrightarrow{\left[h, \lambda(\mathsf{b},x).\begin{cases} \mathsf{b} = \mathsf{True}: & j_{\mathsf{c}}(\mathsf{e}\,\mathsf{x}) \\ \mathsf{b} = \mathsf{False}: & (\mathsf{e}\,\mathsf{x})\end{cases}\right]} \mathsf{c}$$

Frex does not yet implement this proposition in its full generality, since it would require a substantial amount of additional infrastructure, either inside Frex or as part of a category-theory library for Idris2. For example, the type of the construction requires a categorical equivalence between some $\mathsf{Model}\,\mathcal{T}'$ for the presentation $\mathcal{T}'$ of involutive $\mathcal{T}$-models and the self-conjugate $\mathcal{T}$-models. Frex currently only implements the special case of Prop. 5.1, with its specialised proof.

6 COMPLETENESS AND CERTIFICATION

Frex uses setoids beyond a mere completion: it automatically extracts the proofs frexlets derive. Concretely, the fral and frex can be constructed by quotienting the term algebra with an equivalence relation: provability. For the fral, it is provability with respect to the axioms of the theory, or a postulated equivalence in the given variables setoid. For the frex $\mathsf{A}[\mathsf{X}]$, we further postulate constants

$\underline{a}$ for every element a : `U A`, and the provability relation includes the following evaluation equations, for every operation $f : n$ and constants $\underline{c}_1, \ldots, \underline{c}_n$: $f(\underline{c}_1, \ldots, \underline{c}_n) = \underline{f(c_1, \ldots, c_n)}$. The inhabitants of the provability relations deeply-embed equational proofs.

These resulting candidate 'abstract' fral and frex validate the universal property, and Frex implements this validation. As a consequence, the provability relation coincides with equality in the fral or frex, therefore, these frexlets are complete. The provability relations are not effective — there is no general algorithm deciding, for all algebraic theories and two terms, whether the two terms are provably equal. Therefore, we can't use the abstract fral and frex to simplify terms by simple evaluation, and we need the creativity of frexlet designers. However, as any other model, soundness ensures that we can construct proofs using a given frexlet simplifier. By invoking the universal property, we get a deeply embedded proof that we can inspect, simplify, print, and certify.

For concreteness, lets look at proof extraction for the frex. Extracting equational proofs for all algebras is similar, using the abstract fral and the fral universal property. Take a typical input to the frex `solve` function, namely a concrete algebra `a`, and an equation `X ⊎ U a` $\vdash t = s$ involving variables and concrete elements in the algebra represented by terms over the disjoint union `X ⊎ U a`. The abstract frex in this situation is the term algebra over `X ⊎ U a` quotiented by provability: `A` := `Term (X ⊎ U a)/Provability`. Even though `A` validates the universal property, the interpretation of t and s in `A` are themselves, i.e. `A` $[\![t]\!] = t$. Provability proofs between `A` $[\![t]\!]$ and `A` $[\![s]\!]$ are non-effective — all they amount to are deep-embeddings of equational proofs in `a`, and `A` doesn't help us find them. However, if we have a frex whose equivalence relation is effective, then we can use `A` as follows. The function `solve` requires an environment `env : X ⟶ U a`, and then appeals to the universal property of the frex `a[X]` with respect to the algebra `a` and this environment. The abstract frex `A` is also an algebra with an appropriate environment — it is a frex after all. We can therefore appeal to the universal property for `a[X]` and get a proof in the setoid `A` for the interpretation of our equation of interest. The equality proof in `A` is a deep-embedding of an equational proof in `a`, our goal.

Pause at this point and reflect about this implementation. We don't need to write any proof extraction code for our solvers. The fral's and the frex's universal properties have done all the presentation-specific heavy-lifting. Frexlet designers shallowly construct proofs, but Frex can nonetheless produce, for free, the deeply-embedded proof. The remainder of this section explains how Frex processes (simplifies, prints, certifies) these deeply-embedded proofs.

Frex's `Lemma` over a theory is a pair of terms with finite support together with a proof that they are equal in the free algebra of the theory. Such lemmata are sound: every `Lemma` for a theory holds in all models of this theory. Frex provides a `mkLemma` smart constructor which runs the given free algebra simplifier, constructs a proof that a stated equivalence holds, and returns a valid `Lemma`.

This mechanism allows users to build up a library of lemmata for their theories. Users can then seamlessly invoke these lemmata in any model, avoiding further Frex calls. This approach however forces the user's project to depend on most of Frex indirectly through such modules. To avoid such dependencies, Frex also supports proof extraction, allowing users to produce standalone lemmata libraries independent of the Frex library.

6.1 Extracting certificates

Our goals for extraction are to (1) produce libraries from lemmata, and (2) produce somewhat idiomatic Idris2 code. The derivation found by Frex may not be what a human would have chosen but it should definitely be possible for a sufficiently-patient human to follow the reasoning steps.

The main challenge was to go from a rich type of derivation trees with arbitrarily nested transitivity, symmetry, and n-ary congruence steps to a linear type of derivations that could be pretty-printed using Idris2's combinators for setoid reasoning. We use a layered representation for

```
data RTList : Rel a -> Rel a where
  Nil  : RTList r x x
  (::) : {0 r : Rel a} -> {y : a}
       -> r x y -> RTList r y z
       -> RTList r x z
```

(a) reflexive-transitive closure

```
data Symmetrise : Rel a -> Rel a where
  Fwd : {0 r : Rel a} -> r x y
        -> Symmetrise r x y
  Bwd : {0 r : Rel a} -> r x y
        -> Symmetrise r y x
```

(b) symmetric closure

```
Derivation : (p : Presentation)
  -> (a : PresetoidAlgebra
            p.signature)
  -> Rel (U a)
Derivation p a
  = RTList      -- Reflexive, Transitive
  $ Symmetrise -- Symmetric
               -- Congruence
  $ Locate p.signature a.algebra
  $ Step p a    -- Axiomatic steps
```

(e) linear derivations

```
data Locate : (sig : Signature) -> (a : Algebra sig) ->
                Rel (U a) -> Rel (U a) where
  ||| We prove the equality by invoking a rule at the
  ||| toplevel
  Here : {0 r : Rel (U a)} -> r x y
                -> Locate sig a r x y
  ||| We focus on a subterm `lhs` that may appear in
  ||| multiple locations and rewrite it to `rhs` using a
  ||| specific rule.
  Cong : {0 r : Rel (U a)} ->
         (t : Term sig (Maybe (U a))) ->
         {lhs, rhs : U a} -> r lhs rhs ->
         Locate sig a r (plug a t lhs) (plug a t rhs)
```

(c) unary congruence closure

```
data Step : (pres : Presentation)
          -> (a : PresetoidAlgebra pres.signature)
          -> Rel (U a) where
  Include : {x, y : U a} -> a.relation x y
          -> Step pres a x y
  ByAxiom : {0 a : PresetoidAlgebra pres.signature}
          -> (eq : Axiom pres)
          -> (env : Fin (pres.axiom eq).support -> U a)
          -> Step pres a
               (a .bindTerm (pres.axiom eq).lhs env)
               (a .bindTerm (pres.axiom eq).rhs env)
```

(d) axiomatic steps

Fig. 12. Layered (a–d) representation of linear derivations (e)

derivations (Fig. 12): (a) the reflexive-transitive closure of (b) the symmetric closure of (c) the unary congruence closure of (d) axiomatic reasoning steps.
(a) Reflexive-transitive closure: type-aligned [37] lists of steps in the closed-over relation: the target element of each element in the list is the source element of the next step.
(b) Symmetric closure: either the relation or its opposite.
(c) Unary congruence closure: It suffices to pair a term with a distinguished variable for the contextual hole, together with a step in the closed-over relation. To ease our pretty-printing code, we distinguish between using the closed-over relation in an empty context, and using it in a context with a distinguished variable represented by the Idris value `Nothing`.
(d) Axiomatic steps An atomic step is either a setoid equivalences, or one of the theory's axioms. Putting these together, we get the type of derivations (Fig. 12(e)).

Every provable derivation decomposes into a value in this layered representation. The modular definition makes decomposition straightforward: we use generic combinators for each closure relation-transformers. Closure under congruence is the trickiest part, decomposing an n-ary congruence into n separate unary congruences, pushing them under the reflexive-transitive and symmetric closure layers, and erasing any congruence steps with the identity context.

```
units : {a, b : Nat} -> (0 + (a + 0)) + b + 0 = a + b
units = %runElab frexMagic MonoidFrexlet Additive
```

```
agdaEx : ∀ {x y} → (2 + x) + (y + 3) ≡ x + (y + 5)
agdaEx = fragment CSemigroupFrex +-csemigroup
```

Fig. 13. FREX's reflection mechanism in (a) Idris2 and (b) Agda

6.2 Proof simplication

Certification also allows us to inspect FREX-generated proofs. Frexlet developers can check whether data-structures and proofs are suboptimal, spurring code refactoring. Concretely, when developing FREX, we noticed proofs with loops: multi-step derivations that start and end in the same term. Such loops come from internal data structures that optimise simplifier-development effort, but insert semantically-irrelevant subterms that can be simplified away. FREX implements a generic proof simplifier that automatically removes all of these loops. This mechanism suggests future investigation of FREX's proof-structure, and additional simplification passes.

7 REFLECTION

Thus far, our examples illustrated interaction with FREX using `solve`. The function `solve` takes, apart from the fral or frex simplifier, the number of free variables and the abstract syntax of a goal. The existing simplifiers in, say, Agda's standard library provide a similar interface. These simplifiers also provide ergonomic usage with Agda's *proof reflection* mechanism.

Proof reflection is a metaprogramming paradigm, available in proof assistants and dependently-typed programming languages. It allows bi-directional communication between the language and its implementation. The language provides: a representation of its syntax; operators for constructing, manipulating and destructuring these term-representations; and primitives for *reifying* terms into the representation (quoting), and *reflecting* encoded terms into ordinary terms (anti-quoting).

Given mechanisms for querying unsolved proof obligations, proof reflection enables the implementation of verified decision procedures for automatically discharging such obligations without boilerplate [9, 12]. Coupled with the strong meta-theoretic properties that dependently-typed implementations of decision procedures can enforce (e.g. relative soundness/completeness), reflection-driven interfaces yield easy-to-use tactics with firm guarantees. It is then natural to ask: can we use proof reflection to automatically call FREX without providing the equation to discharge explicitly?

While the answer is 'yes', with example code in Fig. 13, it's challenging to adhere to FREX's design philosophy: extensibility and common core reuse. We avoid custom reflection-based drivers for individual simplifiers, providing instead a single metaprogram. This program can be instantiated for simplifiers, built-in or user-defined, for any signature and presentation.

The Idris2 and Agda implementations of FREX both contain reflection-based drivers with identical interfaces. Each driver receives a frex simplifier and a model of the corresponding presentation. They try to infer the abstract syntax of the goal equation, based on the expected type.

The drivers have no information about the structure of the algebraic signature in question ahead of time. FREX's inductive `Term` representation means that relevant abstract operator names can be extracted from the presentation. However, matching goal fragments against the abstract syntax of the algebraic interpretation is tightly-coupled to the language's reflection primitives. Implementing FREX in both Idris2 and Agda allows us to compare differences in behaviour.

A key example is the normalisation of arithmetic expressions such as `(x + 1) + y = x + (1 + y)`. Currently in Idris2, when the driver receives the reflected syntax, the normaliser has already reduced it to `(x + 1) + y = x + S y`. As far as the theory of monoids is concerned, `S y` is an atomic expression and is therefore treated as another free variable, distinct from `y`. The Idris2 driver then incorrectly infers the invalid equation `Dyn 0 .+. Sta 1 .+. Dyn 1 =-= Dyn 0 .+. Dyn 2`, and fails to discharge the

goal. In contrast, Agda will not preemptively normalise a quoted expression. Consequently, the Agda driver successfully finds the equation, and Frex solves the previous example. We can find similar pathological examples that also confuse the Agda driver, and there's a trade-off involving the engineering effort required to develop heuristics that avoid such pathologies.

8 EVALUATION

Frex is still in its early stages, and we expect substantial future changes in its functionality, expressiveness, ergonomics, and efficiency. It is premature to conduct extensive and expensive studies like usability and large-scale performance, or extensive benchmarking with respect to other, substantially more mature, ecosystems. Nonetheless, it is instructive to provide some reference measurements of user-experience and frexlet-developer experience to check whether this new design is feasible, and identify further directions.

8.1 Using Frex

Quantitative evaluation. Idris2 encourages interactive, type-driven development, thus it is important that the checker is responsive after changing the program. Following Nielsen [28], our Idris2 implementation aims for response times under one second, and we consider a response time of over 10 seconds when type checking a modification to Frex client-code to be a bug in Idris2.

For typical small equalities that arise incidentally in dependently typed programs, Frex's performance falls very comfortably within Nielsen's limits. For example, the checking time[4] is under 0.1s for terms of size six or below with the commutative solver and terms of size fourteen or below with the non-commutative solver, creating an impression of instantaneous response.

As the term size increases, Frex eventually crosses the one second interactivity threshold. Fig. 14 shows how type-checking times grow with term size and with the number of free variables in a randomly-generated term for the commutative and non-commutative monoid solvers. As the figure shows, Frex's type-checking time generally remains below the interactivity threshold up to terms of around size 30, and only exceeds the ten second threshold (beyond which users' attention is lost) for a few terms of size 45 or above. Our experience with Frex development suggests that the anomalously high checking times for these terms is likely to arise from a performance bottleneck in Idris2's evaluator (Sec. 9) and that the ongoing development of Idris2 may eventually eliminate the problem, bringing the type-checking time for most terms up to size sixty down to a few seconds.

Qualitative evaluation. To experience using Frex, we reproduced Brady et al.'s dependently-typed representation of binary arithmetic [11]. They index binary representations by the natural numbers that they represent, and so when they define the arithmetic operations, the programmer needs to prove their correctness. These proofs typically involve insightful equational reasoning steps interleaved with rote calculational steps such as:

```
c_s + 2*(val_s + ((2 `power` width)*c0)) = ((c_s + val_s) + val_s) + (2*((2 `power` width)*c0))
```

which we may discharge by calling `solve` with the equation repeated. We don't use our reflection capabilities since these kinds of examples are beyond their reach at the moment. The experience is reasonable, with the usual pain points involved in invoking an algebraic simplifier without a reflection mechanism: we need to repeat the equation and its relevant rewriting-context when calling Frex, but notably no other pain. The experience was worse in earlier implementations of Frex due to several now-eliminated performance bottlenecks in Idris2.

[4] We use a rather dated AMD FX-8320 machine with 16GB memory, running Idris 2 version 0.5.1-1011cc616 on Debian Linux.

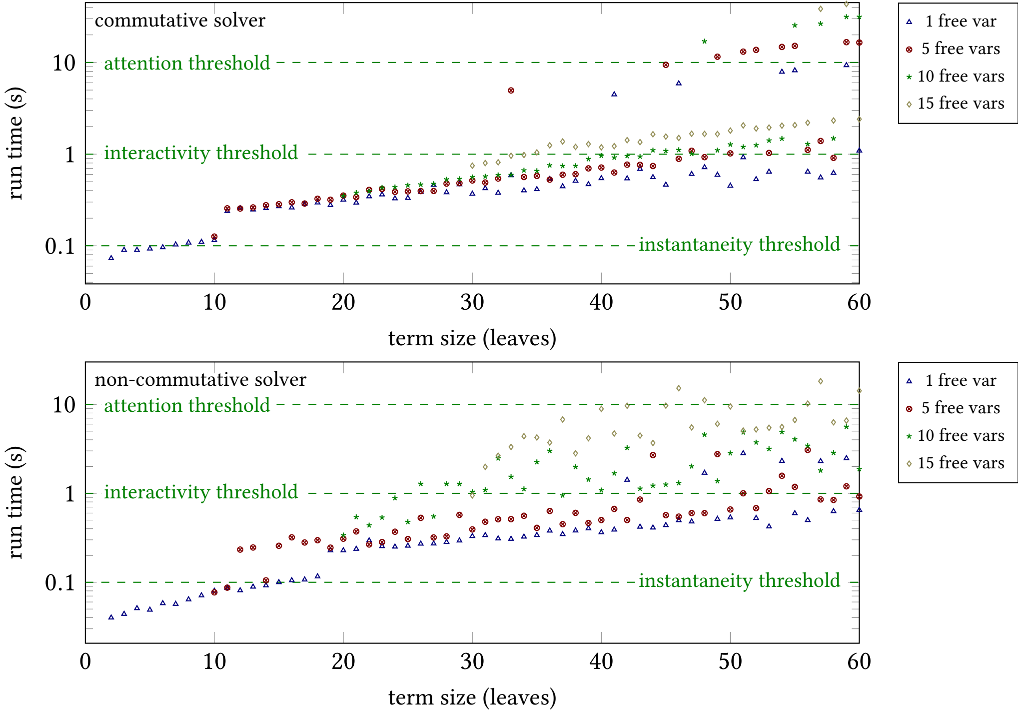


Fig. 14. Frex monoid simplifiers type-checking times

8.2 Extending Frex

The Frex library itself, around 9,500 lines of Idris code, compiles in around 24 seconds. To evaluate its extensibility, we assigned 1 experienced Frex developer the task to extend the library with an involutive monoid frexlet. The development took place over a period of two weeks, with the code-development phase taking 10 days.

This paragraph is for readers who are interested in the breakdown of the experiment. It took the developer around 1 afternoon to design the frex data-structure and produce a pencil-and-paper proof for soundness-and-completeness. However, the developer noticed the structural simplicity of the frexlet, and conjectured a more modular construction might be possible. Within 2 days, they found Jacobs's axiomatisation of involutive algebras [21], and refactored the pencil-and-paper proof using Jacobs's concepts, though still specialised to involutive monoids only. Then code-development began, and the developer discovered a new performance bottleneck in Idris2, which meant that every new 2-3 lines of Idris2 code took 5 minutes or longer to type-check. To work around this issue, the developer proceeded with the 3-buffer approach — using separate buffers for completed; currently-checked; and under-construction definitions. Later, when preparing this manuscript, the developer used the `ByFrex` construct to implement the involutive monoid fral from this frex. This took an additional half-afternoon, with most of the time spent on reducing the construction of the initial involutive monoid to the initial monoid. The management of notation is cumbersome and slowed the fral development by perhaps an hour or so.

Overall the experience was straightforward. We view the experiment as fairly noisy due to the effect of the type-checker bottleneck (now eliminated, see next section). Implementation delays due to algebraic generalisation are likely unavoidable when curious independent-thinking developers

are involved. Since it lead to a new theorem and frexlet design insights, we regard it an advantage. We expect Frex will need an overhaul of its notation-system as it accrues more frexlets. This refactoring might depend on proposing additional notation-management features to Idris2 first.

9 SYSTEM DESIGN LESSONS

Frex uses generic and dependently-typed programming techniques extensively, requiring significant type level computation — an interesting challenge for a language implementation. In developing Frex in Agda and Idris2 we have eliminated some performance bottlenecks in Idris2's type checker, and learned valuable lessons about practical dependently-typed language implementation. We share these lessons here, hoping they will help developers of other systems!

9.1 Idris2

At the heart of the type checker is an implementation of dynamic pattern unification [16, 27, 32], which instantiates implicit arguments, and a conversion checker, which checks whether two terms evaluate to the same common reduct. Each of these requires an evaluator. Idris2 uses a form of normalisation by evaluation [7] with a *syntactic representation* (terms) and a *semantic representation* (values in weak head normal form). The static evaluator is call-by-name and produces a weak head normal form from a term, and Idris2 implements a quotation mechanism which reconstructs a term from a semantic representation of a weak head normal form.

Performance of Evaluation. Most performance bottlenecks we have encountered in developing Frex have been caused by evaluation taking significant time, identified by profiling the Idris2 executable. Idris2 compiles to Scheme, and we have experimented with alternative methods of implementing the evaluator, including evaluating via Scheme to take advantage of the runtime. We have made modest performance gains this way, but in the end nothing is more effective than removing the need to evaluate in the first place! There are various ways we can achieve this, including preserving sharing in subterms, choosing appropriate data representation in unification, and taking advantage of the typical structure of unification and conversion problems.

Preserving Sharing. The nature of dependently-typed programs is that instantiating implicit arguments leads to significant *sharing* of subterms. For example, `[True, False] : Vect 2 Bool` elaborates to `(::) (S Z) Bool True ((::) Z Bool False (Nil Bool))`, sharing the subexpressions `Z` and `Bool`. As the vector gets longer, sharing increases. Following Kovács [23], we preserve sharing by introducing a metavariable for *every* implicit argument, inlining only when we can guarantee that the definition cannot break sharing. Consequently, we inline metavariables whose definition is itself a metavariable applied to local variables. Otherwise, we do not substitute metavariable solutions into terms at all until they are required for unification or display purposes.

Unification. Unification operates on *values*, not *terms*, but sometimes we need to postpone a unification problem if it is blocked due to an unsolved metavariable. When the metavariable is solved, we need to re-evaluate the terms being unified. Previously, we stored postponed problems as a pair of (syntactic) terms in an environment, re-evaluated once the blocking metavariable is solved. However, Frex produces some large postponed problems, for which quotation to syntax is expensive. Now, in addition to the evaluator and quotation, we have introduced a *continue* operation, which re-evaluates the metavariable at the head of a blocked value, and avoids unnecessary quotation.

Conversion Checking. Types in Frex can be large, and sometimes a unification problem that arises while type checking Frex is postponed due to an unsolved metavariable which blocks evaluation. In this case, we might have a unification problem of the form `f x1 ... xn =?= f y1 ... yn` where the `xi`, `yi` etc may be very large subterms, and the terms unify if they are convertible. If most

corresponding terms are equal after evaluation, but one differs, it may take a long time to find the differing subterm which blocks unification, especially since we need to evaluate to check the convertibility of subterms. Fortunately, terms in blocked unification problems tend to differ at the heads, rather than a deeply nested subterm. Therefore, we always check the heads of the values of corresponding `xi` and `yi` first. If any are unequal, we postpone the unification problem. This heuristic has a significant effect on performance, preventing a lot of unnecessary evaluation.

Influence on Language Design and Ecosystem. Developing Frex has identified several desirable language features which have been implemented in Idris2. Many of these have been minor changes to the treatment of implicit arguments and `parameters` blocks. More significantly, Frex makes extensive use of `auto` implicit arguments. These are solved by a search procedure which uses constructors and functions marked as search hints. To help in developing Frex — and in its readability — we have added the ability to mark *local* functions as search hints, which allows us to restrict the scope of search hints and avoid an excessive search-space increase. Frex is now part of the Idris2 test suite, ensuring that it will remain consistent with any updates to Idris2.

9.2 Agda

Agda is a well-established dependently-typed interactive proof environment. Idris2 and Agda and their communities have different goals, leading to subtle Frex implementation differences.

The key differences between the the two languages arise from Agda's focus on proving versus Idris2's focus on programming. At the time of writing, Idris2 uses a single *universe* [29], allowing `Type: Type`, hence inconsistent by Girard's paradox. In contrast, Agda has a well-developed impredicative theory of universes, avoiding Girard's paradox. Agda also protects users from other logical paradoxes of its more experimental features with its '`--safe`' compiler flag. In the spirit of Hu and Carette [18], we adopt a conservative set of compiler options. All our definitions are universe-polymorphic. This broadens the applicability of Frex in the Agda ecosystem by guaranteeing compatability with all of Agda's various configurations. It further assures us about the correctness of Frex itself. Corbyn [15] discusses these ideas in greater detail.

10 RELATED WORK

Within the Coq ecosystem, an abundance of tactics enable algebraic simplification. Boutin's ring [9] and field tactics[5] let programmers discharge proof obligations involving (and requiring!) addition, multiplication, and division operations. Strub's CoqMT [36] extends Coq's Calculus of Inductive Constructions, allowing users to extend the conversion rule with arbitrary decision procedures for first order theories (e.g. Presburger arithmetic). To guarantee this extension preserves good meta-theoretical properties, Strub only extends term level conversion. Implementations of Hilbert's Nullstellensatz theorem (Harrison's in HOL Light [17] and Pottier's in Coq [30]) help users discharge proofs obligations involving polynomial equalities on a commutative integral domain.

Coq's setoid_rewrite is an advanced tactic library for setoid rewriting.[6] Disregarding the difference between the direct manipulation of proof-terms in Idris2 and the tactic-based manipulation in Coq, setoid_rewrite provides abstractions for manipulating parameterised relations (covariant and contravariant), and users can register setoids of interest and custom 'morphisms' — horn-like equational clauses — with the library. The various tactics in the library apply these user-defined axioms to the goal. Users may also register tactics, and the library includes an expressive collection of term-traversal primitives (climbing up and down the syntax tree, repeating sub-tactics, and so on). While setoid_rewrite doesn't deal with algebraic simplification directly, it may help in

[5]See the Coq documentation: https://coq.inria.fr/distrib/current/refman/addendum/ring.html .
[6]See the Coq documentation: https://coq.inria.fr/refman/addendum/generalized-rewriting.html .

generalising equality-based simplifiers to setoid-based simplifiers. In comparison, Frex's setoid reasoning is minimal, implementing only the necessary features for the library.

In Idris1, Slama and Brady [33, 34] implement a hierarchy of rewriting procedures for algebraic structures of increasing complexity. Though the procedures' completeness is not enforced by type like in Frex, these simplifiers are based on a Knuth-Bendix resolution of critical pairs, and so are likely to be complete. Frex also investigates a hierarchy of rewriting procedures, but: (1) frexlets are complete by construction, (2) Frex is based on normalisation-by-evaluation (like Boutin's tactic, and unlike Slama-Brady), and (3) our library is extensible, where sufficiently motivated users can extend the library with bespoke solvers, and we provide some support for them to do so.

Normalisation-by-evaluation is an established technique for simplifying terms in a concrete equational theory, often involving function types. One compelling example is Allais et al.'s work [2], which demonstrates by a careful model construction that the equational theory decided by normalisation-by-evaluation can be enriched with additional rules. They implement a simply-typed language internalising the functorial and fusion laws for list `fold`, `map`, and `append`. They prove their construction sound and complete with respect to the extended equational theory.

In Agda, Cockx et al.'s '`--rewriting`' flag [13, 14] allows users to enrich the existing reduction relation with new rules. Their implementation goes beyond Allais et al.'s: it may restart stuck computations. Guaranteeing the soundness of user-provided reduction rules by ensuring they neither introduce non-termination nor break canonicity is left to future work. Concretely comparing both of their techniques to our proposed technique, neither currently deals with commutativity.

The Meta-F$\star$ language [26] provides normalisation tactics for commutative monoids and semirings through its metaprogramming facilities. Frex's usage resembles these tactics' usage. We hope a Frex port can use their metaprogramming facilities to reduce some syntactic noise.

11 CONCLUSIONS AND FURTHER WORK

We have presented a novel, mathematically structured, design for algebraic simplification suites that guarantees sound and complete simplification, even of user-defined simplifiers. Our preliminary evaluation shows that, despite its high level of abstraction, the resulting library is responsive, and provides comparable functionality to other libraries, in a combination of features no single library provides. Frex's unique design — the frex and the fral — offer new prospects and questions.

Yallop et al.'s [38] partial evaluators include additional frexlets (abelian groups, semirings, distributive lattices). We plan to follow suit and port the remaining simplifiers. The main challenge: unlike Yallop et al., we need to mechanically prove these frexlet are complete, which is more costly. We would then be in a fair position to conduct larger evaluation and comparison studies. One particularly elegant motivation for including more simplifiers is the following. The frex generalises the 'ring of polynomials over a ring' to that of an algebra of polynomials over an algebra. By porting Yallop et al.'s family of representations, we will fully realise this generalisation.

Our experiment with reflection as well as the reflection-based interfaces of existing solvers show that with enough engineering efforts, library designers can extract the goal equation from the goal type. Unfortunately, software engineers for dependently-typed languages are a scarce resource. We want other principled approaches. In practice, when invoked inside a chain of equational steps, the goal equation already appears in the source-code, albeit in a context. Programmers seem willing to type the goal equation once, since it documents the reasoning steps, but seem unhappy to do so *twice*. Perhaps generic programming with holes[7] could use this already-available information.

Simplifier certification may enable bootstrapping the library along the following lines. In the first iteration, we start with a hierarchy of less efficient, but easy to implement, simplifiers. Then,

[7]See Brad Hardy's Agda-Holes library: https://github.com/bch29/agda-holes .

using the library, one develops a hierarchy of more efficient simplifiers and proof-simplifiers. With the certification mechanism, one then extracts proofs to complete the bootstrap.

We would like to extend Frex's design beyond algebraic structures. More general notions of theories abound: multisorted, second-order/parameterised, and essentially algebraic. We may then cover much more complex situations, such as decision procedures for first order theories (e.g. Presburger arithmetic, cf. Strub's CoqMT [36]) and normalisation-by-evaluation for fusion laws [2], or equational manipulation of big-operators [8, 24, 25]. Note that Frex can deal with big-operators such as `sum` so long as the argument list is a concrete collection of constants and variables such as `sum [2, x]`. We only need the more sophisticated theories when the length of the lists is abstract.

Frex uses many category-theoretic concepts, but the library itself is oblivious to category theory. We hope that a rich category theory library like Hu and Carette's [18] `agda-categories` would lead to a sleeker and even more modular Frex implementation. In particular, we want to explore a general treatment of involutive algebras following Jacobs [21], and Power's *distributive tensor* of equational theories [20, 31] for a uniform treatment of semi-ring varieties. By instantiating each of the 6 semi-group varieties, we can cover each combination of the following combinations:

$$\begin{pmatrix} \{\text{ordinary}\} \times \{\text{ordinary, involutive, non-reversing involutive}\} \\ \cup \{\text{commutative}\} \times \{\text{ordinary, involutive}\} \end{pmatrix} \times \{\text{semigroup, monoid, group}\}$$

and modularly construct $(2+3) \times 3 = 15$ semi-ring varieties, including rings and semirings. Such a modular treatment would provide a multiplicative development boost.

ACKNOWLEDGMENTS

Supported by the Engineering and Physical Sciences Research Council grant EP/T007265/1 and an Industrial CASE Studentship, a Royal Society University Research Fellowship, a Facebook Research Award, and an Alan Turing Institute seed-funding grant. An earlier, unpublished, outline of this work appeared as part of a short-abstract in TyDe'20 [1]. We are grateful to Jacques Carette, Donovan Crichton, Joey Eremondi, Conor McBride, James McKinna, Sam Lindley, Wojciech Nawrocki, Kasia Marek, and Robert Wright for useful discussions and suggestions.

REFERENCES

[1] Guillaume Allais, Edwin Brady, Ohad Kammar, and Jeremy Yallop. 2020. Frex: indexing modulo equations with free extensions. (2020). The 5th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe'2020).

[2] Guillaume Allais, Conor McBride, and Pierre Boutillier. 2013. New equations for neutral terms: a sound and complete decision procedure, formalized. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, Stephanie Weirich (Ed.). ACM, 13–24. https://doi.org/10.1145/2502409.2502411

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2009. Hints in Unification. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–98.

[4] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

[5] John C. Baez and James Dolan. 1998. Higher-Dimensional Algebra III.n-Categories and the Algebra of Opetopes. *Advances in Mathematics* 135, 2 (1998), 145–206. https://doi.org/10.1006/aima.1997.1695

[6] Bruno Barras, Benjamin Grégoire, Assia Mahboubi, Laurent Théry, Patrick Loiseleur, and Samuel Boutin. 2021. *The Coq Proof Assistant: Reference Manual. Ring and field: solvers for polynomial and rational equations.* Technical Report. INRIA. Section 3.2.4.

[7] U. Berger and H. Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda -calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 203–211. https://doi.org/10.1109/LICS.1991.151645

[8] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. 2008. Canonical Big Operators. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–101.

[9] Samuel Boutin. 1997. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1281 (1997), 515–529. https://doi.org/10.1007/BFB0014565

[10] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[11] Edwin Brady, James McKinna, and Kevin Hammond. 2007. Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. 159–176. 8th Symposium on Trends in Functional Programming 2007, TFP 2007 ; Conference date: 02-04-2007 Through 04-04-2007.

[12] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (2016). https://doi.org/10.1145/2951913

[13] Jesper Cockx. [n.d.]. Type theory unchained: Extending type theory with user-defined rewrite rules. ([n. d.]). Submitted to the TYPES 2019 post-proceedings.

[14] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proc. ACM Program. Lang.* 5, POPL, Article 60 (jan 2021), 29 pages. https://doi.org/10.1145/3434341

[15] Nathan Corbyn. 2021. *Proof Synthesis with Free Extensions in Intensional Type Theory.* Technical Report. University of Cambridge. MEng Dissertation.

[16] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. https://personal.cis.strath.ac.uk/adam.gundry/thesis/thesis-2013-07-24.pdf

[17] John Harrison. 2007. Automating Elementary Number-Theoretic Proofs Using Gröbner Bases. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 51–66. https://doi.org/10.1007/978-3-540-73595-3_5

[18] Jason Z. S. Hu and Jacques Carette. 2021. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 327–342. https://doi.org/10.1145/3437992.3439922

[19] Gérard P. Huet and Amokrane Saïbi. 2000. Constructive category theory. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 239–276.

[20] Martin Hyland and John Power. 2006. Discrete Lawvere theories and computational effects. *Theoretical Computer Science* 366, 1 (2006), 144–162. https://doi.org/10.1016/j.tcs.2006.07.007 Algebra and Coalgebra in Computer Science.

[21] Bart Jacobs. 2021. Involutive Categories and Monoids, with a GNS-Correspondence. *Foundations of Physics* 42 (2021), 874–895. Issue 7. https://doi.org/10.1007/s10701-011-9595-7

[22] Donnacha Oisín Kidney. 2019. *Automatically and Efficiently Illustrating Polynomial Equalities in Agda.* Technical Report. University College Cork. BSc Dissertation.

[23] András Kovács. 2019. Fast Elaboration for Dependent Type Theories. Talk at EU Types WG meeting, 2019.

[24] Stella Lau. 2017. Theory and implementation of a general framework for big operators in Agda. Bachelor's thesis, University of Cambridge.

[25] Leonhard Markert. 2015. *Big operators in Agda.* Master's thesis. MSc thesis, University of Cambridge.

[26] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[27] Dale Miller. 1992. Unification under a mixed prefix. *Journal of Symbolic Computation* (1992). http://www.sciencedirect.com/science/article/pii/074771719290011R

[28] Jakob Nielsen. 1993. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[29] Erik Palmgren. 1998. On Universes in Type Theory. In *Twenty-Five Years of Constructive Type Theory*, Giovanni Sambin and Jan M. Smith (Eds.). Oxford University Press, Oxford, United Kingdom, Chapter 12, 191–204. https://doi.org/10.1093/oso/9780198501275.003.0012

[30] Loic Pottier. 2008. Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008 (CEUR Workshop Proceedings, Vol. 418)*, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-418/paper5.pdf

[31] John Power. 2005. Discrete Lawvere Theories. In *Algebra and Coalgebra in Computer Science*, José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan Rutten (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–363.

[32] Jason Reed. 2009. Higher-order constraint simplification in dependent type theory. *ACM International Conference Proceeding Series* (2009), 49–56. https://doi.org/10.1145/1577824.1577832

[33] Franck Slama. 2018. *Automatic generation of proof terms in dependently typed programming languages.* Ph.D. Dissertation. http://hdl.handle.net/10023/16451

[34] 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.). Springer International Publishing, Cham, 40–55.

[35] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.

[36] Pierre-Yves Strub. 2010. Coq Modulo Theory. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 529–543. https://doi.org/10.1007/978-3-642-15205-4_40

[37] Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 133–144. https://doi.org/10.1145/2633357.2633360

[38] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-Static Data as Free Extension of Algebras. *Proc. ACM Program. Lang.* 2, ICFP, Article 100 (July 2018), 30 pages. https://doi.org/10.1145/3236795

A PROOF PRINTING AND CERTIFICATION

Fig. 15 shows an automatically extracted proof for the equation $(x \bullet 3) \bullet 2 = 5 \bullet x$ in the additive monoid structure (`Nat`, 0, (`+`)). The extracted proof has 24 steps — far from the shortest proof possible.

Extraction removes reflexivity and transitivity steps, and the pointed bracket tells whether the step uses the axiom directly (angle points right) or using symmetry (angle points left). Square brackets mean appealing to congruence, where the context is the congruence's context, and the term in the hole is the equation's LHS. Fig. 16 shows an automatically extracted certificate for the equation $0 + (x + 0) + 0 = x$ in a generic monoid `m` = (`U m`, `O1`, (`.+.`)). The certificate is generated inside a module that parameterises over the generic monoid `m` and introduces the various notations and reasoning functions.

$$(x \bullet 3) \bullet [2] \underset{\langle \text{ Right neutrality }]}{\underset{\uparrow}{=}} (x \bullet 3) \bullet 2 \bullet [\varepsilon] \overset{\langle \text{ Left neutrality }]}{\overset{\downarrow}{=}} (x \bullet [3]) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{\langle \text{ Right neutrality }]}{\underset{\uparrow}{=}} (x \bullet 3 \bullet [\varepsilon]) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{\langle \text{ Left neutrality }]}{\overset{\downarrow}{=}} ([x] \bullet 3 \bullet \varepsilon \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{\langle \text{ Right neutrality }]}{\underset{\uparrow}{=}}$$

$$([x \bullet \varepsilon] \bullet 3 \bullet \varepsilon \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{\langle \text{ Right neutrality }]}{\overset{\downarrow}{=}} ([(x \bullet \varepsilon) \bullet \varepsilon] \bullet 3 \bullet \varepsilon \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{\langle \text{ Left neutrality }]}{\underset{\uparrow}{=}} (([\varepsilon] \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet 3 \bullet \varepsilon \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Evaluate } \rangle}{\overset{\downarrow}{=}}$$

$$[(0 \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet 3 \bullet \varepsilon \bullet \varepsilon] \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{\langle \text{ Associativity }]}{\underset{\uparrow}{=}} (0 \bullet [((x \bullet \varepsilon) \bullet \varepsilon) \bullet 3 \bullet \varepsilon \bullet \varepsilon]) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Associativity } \rangle}{\overset{\downarrow}{=}} (0 \bullet [((x \bullet \varepsilon) \bullet \varepsilon) \bullet 3] \bullet \varepsilon \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{[\text{ Commutativity } \rangle}{\underset{\uparrow}{=}}$$

$$(0 \bullet [(3 \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet \varepsilon \bullet \varepsilon]) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{\langle \text{ Associativity }]}{\overset{\downarrow}{=}} [0 \bullet 3 \bullet ((x \bullet \varepsilon) \bullet \varepsilon) \bullet \varepsilon \bullet \varepsilon] \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{[\text{ Associativity } \rangle}{\underset{\uparrow}{=}} ((0 \bullet 3) \bullet ((x \bullet \varepsilon) \bullet \varepsilon) \bullet [\varepsilon \bullet \varepsilon]) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Left neutrality } \rangle}{\overset{\downarrow}{=}}$$

$$((0 \bullet 3) \bullet [(x \bullet \varepsilon) \bullet \varepsilon] \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{\langle \text{ Associativity }]}{\underset{\uparrow}{=}} ((0 \bullet 3) \bullet (x \bullet [\varepsilon \bullet \varepsilon]) \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Left neutrality } \rangle}{\overset{\downarrow}{=}} ([0 \bullet 3] \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \underset{[\text{ Evaluate } \rangle}{\underset{\uparrow}{=}}$$

$$(3 \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon \overset{\langle \text{ Associativity }]}{\overset{\downarrow}{=}} 3 \bullet [((x \bullet \varepsilon) \bullet \varepsilon) \bullet 2 \bullet \varepsilon \bullet \varepsilon] \underset{[\text{ Associativity } \rangle}{\underset{\uparrow}{=}} 3 \bullet [((x \bullet \varepsilon) \bullet \varepsilon) \bullet 2] \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Commutativity } \rangle}{\overset{\downarrow}{=}}$$

$$3 \bullet [(2 \bullet (x \bullet \varepsilon) \bullet \varepsilon) \bullet \varepsilon \bullet \varepsilon] \underset{\langle \text{ Associativity }]}{\underset{\uparrow}{=}} 3 \bullet 2 \bullet ((x \bullet \varepsilon) \bullet \varepsilon) \bullet \varepsilon \bullet \varepsilon \overset{[\text{ Associativity } \rangle}{\overset{\downarrow}{=}} (3 \bullet 2) \bullet ((x \bullet \varepsilon) \bullet \varepsilon) \bullet [\varepsilon \bullet \varepsilon] \underset{[\text{ Left neutrality } \rangle}{\underset{\uparrow}{=}}$$

$$(3 \bullet 2) \bullet [(x \bullet \varepsilon) \bullet \varepsilon] \bullet \varepsilon \overset{\langle \text{ Associativity }]}{\overset{\downarrow}{=}} (3 \bullet 2) \bullet (x \bullet [\varepsilon \bullet \varepsilon]) \bullet \varepsilon \underset{[\text{ Left neutrality } \rangle}{\underset{\uparrow}{=}} [3 \bullet 2] \bullet (x \bullet \varepsilon) \bullet \varepsilon \overset{[\text{ Evaluate } \rangle}{\overset{\downarrow}{=}} 5 \bullet [(x \bullet \varepsilon) \bullet \varepsilon] \underset{[\text{ Right neutrality } \rangle}{\underset{\uparrow}{=}} 5 \bullet [x \bullet \varepsilon] \overset{[\text{ Right neutrality } \rangle}{\overset{\downarrow}{=}} 5 \bullet x$$

Fig. 15. Frex-extracted proof of $(x \bullet 3) \bullet 2 = 5 \bullet x$ in the additive monoid over `Nat`

```
units : (x : U m) -> O1 .+. (x .+. O1) .+. O1 =~= x
units x = CalcWith (cast m) $
  |~ O1 .+. (x .+. O1) .+. O1
  ~~ O1 .+. (O1 .+. x .+. O1) .+. O1
    ..<( Cong (\ focus => O2 :+: (focus :+: O2) :+: O2) $ lftNeutrality x )
  ~~ O1 .+. (O1 .+. (x .+. O1)) .+. O1
    ..<( Cong (\ focus => O2 :+: focus :+: O2) $ associativity O1 x O1 )
  ~~ O1 .+. O1 .+. (x .+. O1) .+. O1
    ...( Cong (\ focus => focus :+: O2) $ associativity O1 O1 (x .+. O1) )
  ~~ O1 .+. O1 .+. x .+. O1 .+. O1
    ...( Cong (\ focus => focus :+: O2) $ associativity (O1 .+. O1) x O1 )
  ~~ O1 .+. x .+. O1 .+. O1
    ...( Cong (\ focus => focus :+: Val x :+: O2 :+: O2) $ lftNeutrality O1 )
  ~~ O1 .+. x .+. (O1 .+. O1)
    ..<( associativity (O1 .+. x) O1 O1 )
  ~~ O1 .+. x .+. O1
    ...( Cong (\ focus => O2 :+: Val x :+: focus) $ lftNeutrality O1 )
  ~~ O1 .+. x
    ...( rgtNeutrality (O1 .+. x) )
  ~~ x
    ...( lftNeutrality x )
```

Fig. 16. FREX-certificate for the of $0 + (x + 0) + 0 = x$ in a generic monoid m