Frex: dependently typed algebraic simplification

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

We present a new design for an algebraic simplification library structured around concepts from universal algebra: theories, models, homomorphisms, and universal properties of free algebras and free extensions of algebras. The library's dependently typed interface guarantees that both built-in and user-defined simplification modules are terminating, sound, and complete with respect to a well-specified class of equations. We have implemented the design in the Idris 2 and Agda dependently typed programming languages and shown that it supports modular extension to new theories, proof extraction and certification, goal extraction via reflection, and interactive development.

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

Algebraic simplification involves using algebraic laws to normalise expressions with unknowns. For example, the commutative monoid axioms—associativity, neutrality, and commutativity—over integers with addition serve to simplify the left hand expression to the right hand expression below:

$$-6 + (x + 3) + (y + x) \xmapsto{\text{simplify}} -3 + 2x + y$$

Many application domains make use of this kind of simplification. For example, a program optimiser, such as a compiler or partial evaluator, can be conveniently structured as an algebraic simplifier that converts each expression to a canonical form followed by a code generator that deals only with canonical expressions, guaranteeing uniform treatment of many syntactically distinct programs. The present work focuses on another application domain: interactive theorem provers and programming languages based on dependent type theory. In these systems, users often need to prove to a type checker that terms are equivalent, but constructing the proofs often involves rote algebraic simplification steps that users resent producing manually.

To free users from the need to construct rote algebraic proofs, dependently typed languages and their ecosystems often include simplifiers for common algebraic structures such as monoids, semi-rings, and rings. With these simplifiers, users need only establish the structures' axioms, such as neutrality, associativity and commutativity, and can then call the simplifiers to discharge rote simplification steps. Implementation strategies for simplifiers vary, from using tactics to simplify

Authors' Contact Information: Guillaume Allais, guillaume.allais@ens-lyon.org, University of Strathclyde, Glasgow, Scotland, UK; Edwin Brady, ecb10@st-andrews.ac.uk, University of St. Andrews, St. Andrews, Fife, Scotland, UK; Nathan Corbyn, nathan.corbyn@cs.ox.ac.uk, University of Oxford, Oxford, England, UK; Ohad Kammar, ohad.kammar@ed.ac.uk, University of Edinburgh, Edinburgh, Scotland, UK; Jeremy Yallop, jeremy.yallop@cl.cam.ac.uk, University of Cambridge, Cambridge, England, UK.

2025. ACM 2475-1421/2025/1-ART
https://doi.org/

algebraic terms in typing goals [e.g. Barras et al. 2021] to using proof-by-reflection to construct propositions to discharge equations [e.g. Kidney 2019]. Some simplifiers group together several algebraic structures [e.g. Barras et al. 2021], and others generalise several distinct structures to one structure [Grégoire and Mahboubi 2005], but the state-of-the-art are standalone simplifiers that, through heuristics and long-term development, can deal with common cases.

1.1 Representation Theorems

Universal algebra has a long tradition concerning algebraic simplification under the collective name 'representation theorems'. Each such representation theorem characterises canonical representatives of algebraic expressions in terms of (typically inductive) constructions such as reduced-words and formal polynomials. Such characterisations often reuse existing representation theorems of simpler algebraic structures or familiar algebraic structures such as the integers or the natural numbers.

The present work makes use of two kinds of representation theorems. For a free algebra (abbreviated *fral*), a representation theorem amounts to an algebraic structure that chooses a canonical representation for expressions using only the algebraic laws. For a free extension (abbreviated *frex*), a representation theorem chooses a canonical representative using the algebraic laws while also evaluating concrete elements. Free algebras and free extensions are related: each free extension is also a free algebra of a theory specialised to a concrete algebra by adding evaluation axioms.

For example, for commutative monoids the fral representation theorem states that the free commutative monoid over n variables is represented by the set of n-tuples of naturals $\mathbb{N}^n$. We can use this fral representation to perform simplification by evaluating a term in the fral and then reifying it back as a term:

$$-6+(x+3)+(y+x) \xmapsto{\text{evaluate}\,(x_0+(x_2+x_1)+(x_3+x_2))} (1,1,2,1) \xmapsto{\text{reify}\,(x_0\mapsto -6, x_1\mapsto 3, x_2\mapsto x, x_3\mapsto y)} -6+3+2x+y$$

The fral is unaware of the distinction between the variables we extend by (x and y) and the concrete elements of the algebra (-6 and 3 here), and so fral simplification treats both -6 and 3 as abstract and distinct indeterminates.

In contrast, the frex representation theorem for commutative monoids states that the free extension of a commutative monoid over C by n variables is represented by the set $C \times \mathbb{N}^n$, where the element $(c, a_1, \ldots, a_n)$ represents the expression $c + a_1x_1 + \ldots + a_nx_n$. Simplifying a term using the frex representation involves evaluating a term in the frex and then reifying it back as a term:

$$-6 + (x + 3) + (y + x) \xmapsto{\text{evaluate}_{\mathbb{Z}}} (-3, 2, 1) \xmapsto{\text{reify}} -3 + 2x + y$$

The frex representation distinguishes variables from concrete elements, gathering the latter together and evaluating them using the operations of the concrete commutative monoid in use.

This technique of normalizing terms by evaluating and then reifying also applies to more sophisticated notions of algebra that include the equational theories of λ-calculi, and is familiar in those settings as *normalization-by-evaluation*. Its first systematic applications were in the formal study of various simply typed calculi [Altenkirch et al. 2001, 1995; Čubrić et al. 1998] and category theoretic constructions [Beylin and Dybjer 1996]. It has served as a conversion-checking technique during the type-checking of dependently typed calculi from their inception [Martin-Löf 1975], gaining adoption after the seminal works of Abel et al. [2007a,b], even for sophisticated calculi [Abel et al. 2017; Hu et al. 2023; Sterling and Angiuli 2021].

This manuscript describes an extensible dependently typed library for algebraic simplifiers based on fral and frex representation theorems, drawing inspiration from previous work that uses free extensions for partial evaluation [Yallop et al. 2018]. Current implementations of algebraic simplifiers, even in dependently typed settings, are restricted to implementing the computational

representation—i.e. the data-structures needed for the normal form together with the normalisation-by-evaluation algorithm—alongside a formalisation of the soundness proofs. This work investigates what can be gained by the more radical approach of encoding, in addition, the full meta-theory of these representation theorems, including generic representations of theories, their algebras and algebra homomorphisms, and the universal properties of the fral and the frex. For simplicity of development and exposition we apply this generic machinery to a handful of familiar monoid varieties. Moreover, we ensure all of these concepts remain computational by avoiding the temptation of postulating axioms that could hinder reduction of closed terms. This last task is challenging to satisfy while retaining interactive performance, as formalising universal properties tends to produce large terms that slow type-checkers [Gross et al. 2014].

1.2 Paper Outline and Contributions

Sections 2 and 3 present background material: a review of the mathematical foundation for the Frex library (Section 2), and a brief Idris2 tutorial that reviews setoid-based equational reasoning (Section 3).

Sections 4 and 5 present our central contribution, a fundamentally new approach to building algebraic simplifiers. The standard existing approach is to write an ad-hoc simplifier for some particular algebraic structure such as rings. Our approach is radically different: we teach the implementation the basic concepts of universal algebra — signatures, theories and models, homomorphisms, and universal properties (Section 4) — then build a completely generic solver based on free algebras and free extensions that can be instantiated with a particular algebra to discharge concrete proof obligations (Section 5). This new approach is inherently modular and extensible, and delivers solvers that are sound and complete by construction. We have implemented our design in two dependently typed languages, Agda Fragment[1] and Idris2 Frex[2]

Section 6 explains the completeness guarantees of the library, and covers proof extraction, simplification, pretty-printing and certification. (Sections 5 and 6 are technically involved and are aimed at library designers, and may be skimmed at first reading.)

Section 7 considers a natural question: can one use reflection to invoke Frex automatically? The answer is a qualified 'yes', requiring much library-developer effort, but leading to real advantages in Agda and limited advantages in Idris2.

Section 8 reports some supplementary evaluation of Frex. The key properties of our design are guaranteed by the type theories of the languages in which we realise it: it delivers sound and complete solvers in a completely generic way, with support for proof extraction, certification, etc. However, the practical questions of usability and viability for interactive development cannot be established by theorems and so we have also carried out some experiments. These experiments focus on the varieties of monoids that also serve as our running example, and establish that the generic solver is comfortably fast enough for interactive use, and can be extended with new algebras in a modular way and without enormous effort.

Section 9 discusses system design issues that we encountered with Frex, and Sections 10 and 11 conclude with related and further work.

Appendixes A–D, which are included in the full version of the paper submitted as supplementary material, have more information about Frex's codebase, example code extraction, and involutive monoids.

[1] Available here: https://github.com/frex-project/agda-fragment.
[2] Available here: https://github.com/frex-project/idris-frex.

We also include as supplementary material our implementation, Frex, which consists of 9,500 lines of Idris2 code. The paper includes only those excerpts of code necessary to convey the key ideas, and we refer the reader to the implementation for full details.

2 Mathematical Overview

Universal algebra concerns the generic description of, and relationship between, algebraic structures. We summarise briefly the concepts underlying the Frex library.

2.1 Presentations of Algebraic Structures

A *finitary signature* $\Sigma = (\texttt{Op}\,\Sigma, \texttt{arity})$ consists of a set $\texttt{Op}\,\Sigma$ of *operators*, also known as *operation symbols*, and an assignment $\texttt{arity} : \texttt{Op}\,\Sigma \to \texttt{Nat}$ of a natural number to each operator called its *arity*. For example, the additive signature often used for commutative monoids has two operators: $\texttt{Op}$ **Additive** $:= \{(+), 0\}$, with arities 2 and 0, respectively. It is standard to use a more succinct notation that groups operators and their arities, as in $\texttt{Op}$ **Multiplicative** $:= \{(\cdot) : 2, 1 : 0\}$, the multiplicative signature used for ordinary monoids.

Signatures determine an algebraic language, and an algebra is its semantic model. An *algebra* $\texttt{A} = (\texttt{U}\,\texttt{A}, \texttt{A}\,[\![-]\!])$ for a signature Σ consists of a set $\texttt{U}\,\texttt{A}$ called the *carrier* and an assignment of an *n-ary operation* over this carrier for every n-ary operator $f : n$ in Σ, i.e. a function $\texttt{A}\,[\![f]\!] : (\texttt{U}\,\texttt{A})^n \to \texttt{U}\,\texttt{A}$. So **Additive**-algebras and **Multiplicative**-algebras amount to triples $(X, [\![(+)]\!] : X^2 \to X, [\![0]\!] \in X)$. For example, we can equip the natural numbers $\mathbb{N}$ with several algebra structures: arithmetic addition $(\mathbb{N}, (+), 0)$, arithmetic multiplication $(\mathbb{N}, (\cdot), 1)$; maximum $(\mathbb{N}, \max(a, b), 0)$. Similarly, $n \times n$ matrices over $\mathbb{N}$ have such algebra structures given by matrix addition and multiplication with the zero and identity matrix respectively, and so on.

Each signature determines a language consisting of *terms*. Given a set $\texttt{X}$ of variables, the Σ*-terms over* $\texttt{X}$ are given inductively as either a variable in $\texttt{X}$ or an application $f(t_1, \ldots, t_n)$ of an operation symbol $f : n$ from Σ to n terms over $\texttt{X}$. The primary role of terms is to designate *equations in context* $\texttt{X} \vdash t = s$, i.e. triples consisting of a set $\texttt{X}$ of variables and two terms in context $\texttt{X}$. For example, the associativity equation, expressed over the **Additive** signature, is $x, y, z \vdash x + (y + z) = (x + y) + z$.

An *environment* for a context (=set) $\texttt{X}$ in an algebra $\texttt{A}$ is a function $\texttt{e} : \texttt{X} \to \texttt{U}\,\texttt{A}$. An algebra $\texttt{A}$ determines, for each term in context $\texttt{X} \vdash t$, an *interpretation* function $\texttt{A}\,[\![t]\!] : (\texttt{U}\,\texttt{A})^{\texttt{X}} \to \texttt{U}\,\texttt{A}$ that, given an environment $\texttt{e}$, uses the algebra structure to interpret each operator as its corresponding operation structurally: $\texttt{A}\,[\![\texttt{x}]\!]\,\texttt{e} := \texttt{e}\,\texttt{x}$ and $\texttt{A}\,[\![f(t_1, \ldots, t_n)]\!]\,\texttt{e} := \texttt{A}\,[\![f]\!]\,(\texttt{A}\,[\![t_1]\!]\,\texttt{e}, \ldots, \texttt{A}\,[\![t_n]\!]\,\texttt{e})$. For example, the interpretation of the left-hand-side (LHS) of the associativity axiom in the **Additive**-algebra $(\mathbb{N}, \max, 0)$ given the environment $x \mapsto 5, y \mapsto 3, z \mapsto 8$ is $\max(5, \max(3, 8)) = 8$.

We say that an equation is *valid* in an algebra $\texttt{A}$, writing $\texttt{A} \models (\texttt{X} \vdash t = s)$, when $\texttt{A}\,[\![t]\!]\,\texttt{e} = \texttt{A}\,[\![s]\!]\,\texttt{e}$ for all environments $\texttt{e} : \texttt{X} \to \texttt{U}\,\texttt{A}$. It is this implicit universal quantification over the environment that gives universal algebra its name. For example, the **Additive**-algebras and **Multiplicative**-algebras presented so far validate the associativity axiom, whereas interpreting the binary operation as subtraction over the integers $(\mathbb{Z}, (-), 0)$ does not validate the associativity equation, e.g. taking the environment $x \mapsto 0, y \mapsto 0, z \mapsto 1$, we have $0 - (0 - 1) = 1 \neq -1 = (0 - 0) - 1$.

A *presentation* $\mathcal{T} = (\Sigma_{\mathcal{T}}, \mathcal{T}.\texttt{Axiom})$ consists of a signature $\Sigma_{\mathcal{T}}$ and a set $\mathcal{T}.\texttt{Axiom}$ of $\Sigma_{\mathcal{T}}$-equations in context, which we call *axioms*. A $\mathcal{T}$*-algebra* $\texttt{A}$ is a $\Sigma_{\mathcal{T}}$-algebra $\texttt{A}$ validating all $\mathcal{T}$-axioms. For example, the axioms of the **Monoid** presentation consist of associativity and neutrality $(x \vdash x * 1 = x, 1 * x = x)$ over the multiplicative monoid signature. The axioms for the **CommutativeMonoid** presentation, typically phrased over the additive monoid signature, additionally include commutativity $x, y \vdash y + x = x + y$. We can now generically manipulate classes of algebraic structures using these concepts, while generalising the usual examples: **Monoid**-algebras are monoids, **CommutativeMonoid**-algebras are commutative monoids, etc.

2.2 Simplification and Universality

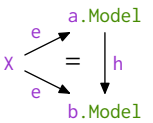
The input to the simplification problem is a term; terms may be purely abstract, definable using the operations and constants in the signature, or partially concrete, involving also constants from a given algebra. In both cases, the goal of the simplification problem is to use this information as much as possible to find a representative modulo the presentation's axioms and the rules of deduction, and, in the partially concrete case, using the evaluation semantics of the given concrete algebra. The goal 'use this information as much as possible' is informal, and an algebraically natural way to formulate it is a free algebra. The term 'free' in this context intends to capture formally the intuition of using all and only the information given by the algebraic axioms, equational logic, and the given abstract and concrete algebra elements. To cast formal meaning to this notion, we need to define the relevant classes of structure-preserving maps. Each such class isolates free constructions, and bringing these maps to the fore is a hallmark of modern algebra. These abstract concepts connect to the frex and fral simplification examples from page 2:

$$-6+(x+3)+(y+x) \xmapsto{\text{evaluate}_{\mathbb{Z}}} (-3,2,1) \xmapsto{\text{reify}} -3+2x+y \tag{1}$$

$$-6+(x+3)+(y+x) \xmapsto{\text{evaluate}\,(x_0+(x_2+x_1)+(x_3+x_2))} (1,1,2,1) \xmapsto{\text{reify}\,(x_0\mapsto -6, x_1\mapsto 3, x_2\mapsto x, x_3\mapsto y)} -6+3+2x+y$$

Let `A`, `B` be Σ-algebras for a signature Σ. A *homomorphism* $h : \texttt{A} \to \texttt{B}$ of Σ-algebras is a semantics-preserving function $h : \texttt{U A} \to \texttt{U B}$ between their carriers. Explicitly, for all operators $f : n$ in Σ and elements $a_1, \ldots, a_n$ in `U A`, we have: $h(\texttt{A}\,[\![f]\!](a_1,\ldots,a_n)) = \texttt{B}\,[\![f]\!](h\,a_1,\ldots,h\,a_n)$. A homomorphism between presentation algebras is a homomorphism between the underlying signature algebras. For example, the list-length function is a homomorphism from the monoid of concatenation over lists to the monoid of addition over naturals: `length` : (`ListX`, (`++`), `[]`) $\to$ ($\mathbb{N}$, (+), 0).

Let $\mathcal{T}$ be a presentation and `X` a set whose elements represent variables. We define a *$\mathcal{T}$-algebra* `a` = (`a.Model`, `Env a`) *over* `X` to be a $\mathcal{T}$-algebra `a.Model` equipped with an `X`-environment in this model, i.e. a function `e` : `X` $\to$ `U`(`a.Model`). This concept formalises the inputs to the fral simplification process from (1). The argument in the label on the left arrow is a **Additive**-term with variables from `X` := $\{x_0, \ldots, x_3\}$. The argument to the label on the right arrow is an `X`-environment in a yet-to-be-determined syntactic monoid involving concrete and abstract elements.

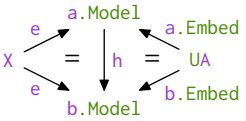


A *morphism* `h` : `a` $\to$ `b` of $\mathcal{T}$-algebras over `X` is a $\mathcal{T}$-algebra homomorphism that moreover makes the diagram on the right commute. A *free $\mathcal{T}$-algebra over* `X` is then a $\mathcal{T}$-algebra over `X` from which there is a unique such morphism to every other $\mathcal{T}$-algebra over `X`. This existence-and-uniqueness property is called the *universal property* of free $\mathcal{T}$-algebras. For example, the free commutative monoid over `X` = $\{x_0, \ldots, x_3\}$ is the **Additive**-algebra over $\mathbb{N}^4$ given by componentwise arithmetic addition, and equipped with the `X`-environment sending x_0 to $(1,0,0,0)$, etc. The unique morphism out of this algebra into any commutative monoid `A`, equipped with an environment `e`, sends $(a_0,\ldots,a_3)$ to $a_0 e(x_0)+\ldots+a_3 e(x_3)$. So the core idea behind fral simplification is to make the creativity that goes into designing the data-structure in the middle of (1) methodological and principled: it is an implementation of the fral. The universal property, which singles the fral up to a unique isomorphism of algebras over `X`, provides a checklist that organises the simplification code.

Let `A` be a $\mathcal{T}$-algebra and `X` a set whose elements represent variables. An *extension* of `A` by `X` is a triple `a` = (`a.Model`, `a.Var`, `a.Embed.H`) consisting of a $\mathcal{T}$-algebra `a.Model`, a function `a.Var` : `X` $\to$ `U`(`a.Model`), and a $\mathcal{T}$-homomorphism `a.Embed` : `A` $\to$ `a.Model`. This concept formalises the inputs to the frex simplification process from (1). This time both concrete elements of the algebra `A` and abstract variables from `X` are part of the vocabularly in the left-most term.

	free algebra		free extension	
ordinary monoid	variable lists	$yxxyx$	alternating lists in $\mathbb{M}_{2\times 2}(\texttt{Nat})[y]$	$\begin{pmatrix}1 & 3\\ 0 & 2\end{pmatrix} y \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} y$
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	$y\overline{x}xx\overline{y}yx$	alternating lists with tagged variables in $\texttt{String}[x,y]$	$\texttt{""}x\texttt{"hello"}y\texttt{"olleh"}\overline{x}\texttt{""}$

Fig. 1. Frals and frexes for varieties of monoids



A *morphism* $\texttt{h} : \texttt{a} \longrightarrow \texttt{b}$ of extensions of $\texttt{A}$ by $\texttt{X}$ is a $\mathcal{T}$-homomorphisms $\texttt{h} : \texttt{a.Model} \longrightarrow \texttt{b.Model}$ that moreover makes the two triangles in diagram on the right commute. A *free extension* of $\texttt{A}$ by $\texttt{X}$ is then an extension from which there is a unique such morphism to every other extension of $\texttt{A}$ by $\texttt{X}$. For example, the free extension of a commutative monoid $\texttt{A}$ by two variables $\texttt{X} := \{x, y\}$ is the **Additive**-algebra over $\texttt{U A} \times \mathbb{N}^2$, given componentwise, equipped with the **Additive**-homomorphism that sends $u \in \texttt{U A}$ to $(u, 0, 0)$ and the function that sends x to $(\texttt{A}\,[\![0]\!]\,, 1, 0)$ and y to $(\texttt{A}\,[\![0]\!]\,, 0, 1)$. The unique morphism to any extension $(\texttt{B}, \texttt{e}, b)$ sends (u, a_0, a_1) to $bu + a_0(\texttt{e}\,x) + a_1(\texttt{e}\,y)$.

Figure 1 summarises the frals and frexes considered in this manuscript. All but the last frex are well-known representations. Our contribution is to implement, alongside these representations, constructive proofs that they represent the fral or the frex, which in turn requires implementing the universal algebraic concepts we introduced so far, enabling other simplification modules to reuse them, and allowing users and library code to be extensible and modular.

2.3 Completeness, Proof Extraction, and Certification with Setoids

One way to design the library, which we explored in an early implementation, is to faithfully represent the concepts in Section 2.1 and Section 2.2. However, generalising the design to use setoids rather than sets enables the same abstractions to offer more flexible functionality.

A *setoid* $\texttt{X} = (\texttt{U X}, (\equiv_{\texttt{X}}))$ consists of a set $\texttt{U X}$ and an equivalence relation $(\equiv_{\texttt{X}})$ over $\texttt{U X}$. A *setoid homomorphism* $\texttt{f} : \texttt{X} \texttt{ ~> } \texttt{Y}$ is a relation-preserving function between sets $\texttt{X}$ and $\texttt{Y}$: $\texttt{f}\,x \equiv_{\texttt{Y}} \texttt{f}\,y$ whenever $x \equiv_{\texttt{X}} y$. We think of elements in $\texttt{U X}$ as representatives of the equivalence classes of $(\equiv_{\texttt{X}})$, and so every setoid homomorphism induces a (unique) function between the quotients $\texttt{X}/(\equiv_{\texttt{X}}) \longrightarrow \texttt{Y}/(\equiv_{\texttt{Y}})$. One way to construct a setoid is to equip a set $\texttt{X}$ with its equality relation to give $(\texttt{X}, (=))$, but using setoids also allows us to explicate sophisticated equivalence relations and define operations on them that are not supported by the corresponding quotients. For example, given a presentation $\mathcal{T}$ and a set $\texttt{X}$, we can define the *provability* relation $\texttt{X} \vdash - = -$ over terms $\texttt{X} \vdash t$ (cf. Fig. 2). Related elements in the setoid $(\texttt{Term X}, \texttt{X} \vdash - = -)$ represent different, but provably equal, terms. In contrast, elements in the quotient $\texttt{Term X}/(\texttt{X} \vdash - = -)$ represent equivalence classes of provably equal terms. Figure 2 also includes the evaluation axiom (EVAL), which we will use to present the frex.

Setoids and their homomorphisms form a common technique to complete an intensional type theory with extensional functions and quotients, requiring users to establish that every defined function is a setoid homomorphism. However, in FREX we make essential use of setoids that a type theory with quotients does not allow. For example, the terms-up-to-provability setoid supports operations such as $\texttt{vars} : \texttt{Term X} \longrightarrow \texttt{List X}$ which extract the list of variables appearing in a given term in-order. This function is not a setoid homomorphism. The quotient can only support such extraction following a canonisation, i.e. a function which first chooses one term representing the equivalence class, and then applies vars to it. Both flavours of function are useful in applications,

$$\frac{}{\mathtt{X} \vdash t = t}\,\textsc{Refl} \qquad \frac{\mathtt{X} \vdash s = t}{\mathtt{X} \vdash t = s}\,\textsc{Sym} \qquad \frac{\mathtt{X} \vdash t = s \quad \mathtt{X} \vdash s = r}{\mathtt{X} \vdash t = r}\,\textsc{Trans} \qquad \frac{(\mathtt{X} \vdash t = s) \in \mathcal{T}.\mathtt{Axiom}}{\mathtt{X} \vdash t = s}\,\textsc{Ax}$$

$$\frac{\mathtt{Y} \vdash t = s \quad \theta_1, \theta_2 : \mathtt{Y} \longrightarrow \mathtt{Term\ Y} \quad (\mathtt{X} \vdash \theta_1 y = \theta_2 y)_{y \in \mathtt{Y}}}{\mathtt{X} \vdash t[\theta_1] = s[\theta_2]}\,\textsc{Cong} \qquad \frac{\mathtt{UA} \vdash t}{\mathtt{X} \vdash t = \mathtt{A}\,[\![t]\!]}\,\textsc{Eval}$$

Fig. 2. Provability in (a) equational logic (unshaded) and (b) the evaluation rule

```
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 a GADT and projections

but setoids, and not quotients, support both. The setoid design supports, for example: printing terms and proofs, proof simplication, code generation, etc.

In Frex, we implement *setoid-enriched* universal algebra: carriers are setoids; algebraic operations are setoid homomorphisms; algebra homomorphisms and environments must also be setoid homomorphisms; and the unique homomorphisms in the universal properties of the fral and the frex must be setoid homomorphisms. With this generalisation, the setoid $\mathtt{Term\ X}/(\mathtt{X} \vdash - = -)$ also satisfies the fral universal property constructively. Therefore, there is a unique canonical setoid isomorphism to every other fral. Similarly, by including the evaluation equations (Eval), we obtain a setoid frex together with a setoid isomorphism to every other frex. These isomorphisms let us extract simplification proofs generically out of user-defined simplifiers.

3 Setoids in Idris2

To introduce the relevant features of Idris2, we review some relevant standard constructions in dependent types [e.g. Hu and Carette 2021; Huet and Saïbi 2000]. 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 annotation `0` preceding the definition of the field `U` is a *quantity* [Atkey 2018; McBride 2016] indicating that the field is not represented at runtime, but may be used in types. If you are 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. We define setoid homomorphisms:

```
(^) : 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. 4. Algebras and setoid algebras in Frex

```
SetoidHomomorphism : (a,b : Setoid)
  -> (f : U a -> U b) -> Type
SetoidHomomorphism a b f
  = (x,y : U a) -> a.equivalence.relation x y
  -> b.equivalence.relation (f x) (f y)
```

```
record (~>) (A,B : Setoid) where
  constructor MkSetoidHomomorphism
  H : U A -> U B
  homomorphic : SetoidHomomorphism A B H
```

The Appendix includes expanded examples for setoids of functions and quotient setoids.

This technique is affectionately dubbed 'setoid hell', since we need to prove that all our functions are setoid homomorphisms. Following Hu and Carette [2021], we manage setoid hell by structuring code categorically, organising results into homomorphisms between appropriate setoids.

4 Universal Algebra in Frex

To define an interface to algebraic simplifiers, we first specify and represent algebraic structures. We implement signatures and their operators in Frex as follows (below, left and middle):

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

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

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

The implementation uses Idris2's implicit record field for `arity`. Users define concrete instances of `Signature`, such as the signature `MkSignature Operation` for monoids, by defining an injective type family for the indexed field `OpWithArity` (above, right). Injectivity avoids projecting the arity in concrete cases, where unification extracts it automatically. Injectivity improves usability but is not otherwise necessary. Idris currently has limited support for injectivity: unification makes use of the injectivity of data-type constructors, but there is no means for a type to require a judgementally injective type-level function.

Frex represents the domain of an n-ary operation with an n-ary vector (Fig. 4). (As in Haskell, backticks turn any name into an infix operator.) For example, the additive natural numbers form an algebra for the monoid signature as follows:

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

```
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. 5. Axiomatising monoids in Frex

The code uses the smart constructor `MkAlgebra` that transfers its `Sem` argument into `MakeAlgebra`'s `Semantics` field by uncurrying each `n`-ary function into a function taking an `n`-ary vector of arguments. 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. 4).

We implement terms over a signature as follows, mirroring their mathematical definition:

```
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
```

Terms form an algebra, the *free* algebra, with symbols denoting term formers:

```
Free : (0 sig : Signature) -> (0 x : Type) -> Algebra sig
Free sig x = MakeAlgebra (Term sig x) Call
```

Terms also form a monad, with `Done` as its unit and substitution as its sequencing operation.

Turning to equations, Frex only needs equations in a finite context, and we call its cardinality the *support* of the equation. We implement equations and presentations as follows:

```
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. 5 has three axioms: left and right neutrality, and associativity. Frex defines a generic collection of axiom schemes (above, right). Its type `EqSpec sig [2]` (lines 1–2) states that it is a scheme involving a single binary operation, and its declaration involves 3 variables (`MkEquation 3` in line 5).

Frex's representation of this statement is in Fig. 6. We use Idris2's dependent pairing construct to pair an algebra with an environment in the standard entailment syntax `eq =| (a ** env)`. The following code validates the monoid axioms for our running example:

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

```
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. 6. Equational validity in an algebra

We define models for a presentation:

```
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. 5. For another example, now putting everything together, we validate the monoid structure of multiplication as follows:

```
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
```

Line 3 converts the constructed algebra into a setoid algebra, and lines 7–9 use results about the natural numbers from Idris2's standard library.

Using Frex

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 are already working abstractly with an algebraic structure, we expect that in practice users start by recognising that their concrete algebra validates the axioms of an existing simplification module—*frexlet* for short. As a concrete example, we will take computations with lists that also involve the `reverse` function. These form an *involutive* monoid: a monoid `A` equipped with a unary *involution* operator $x \mapsto \overline{x} : \mathtt{UA} \to \mathtt{UA}$ satisfying two axioms $\overline{\overline{x}} = x$ and $\overline{xy} = \overline{y}\,\overline{x}$. We then equip our type of interest, lists, with an involutive model structure as in Fig. 7. We can use this algebra and the involutive monoid to discharge equations containing list variables and concrete lists:

```
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. 7. The involutive monoids of list reversal

```
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:

```
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
```

```
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. 8. Setoid algebra homomorphisms in Frex

Lines 2–3 and 8–9 overload the infix and postfix notation using the frexlet's built-in notation suites. Concretely, the projection `Notation1` brings into scope the functions `(.*.)` and `(.inv)` when writing algebraic terms. 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. The variables `x`, `y`, `z` (bound in line 7) are implicitly used in this call. §5.2 covers the type of `solve` in more detail.

5 Free Extensions and Algebras

Before delving into the details of Frex's core, we revisit our frexlet representations using examples for elements in the fral and the frex for ordinary, commutative, and involutive monoids (see Fig. 1).

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.

5.1 Universal Properties

Frex defines homomorphisms of setoid algebras in Fig. 8, 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 setoid 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.

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

Similarly, Fig. 9 presents the declarations for algebras over a setoid and extensions. It expresses the equations in the commuting diagrams using the extensionality equivalence relation on the setoid of functions from Fig. 16b in the Appendix and the power of an algebra by a setoid (see §5.3). As Idris2 resolves names using type-directed disambiguation, we overload the record name `(~>)`.

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 will unpack it for the free commutative monoid over `Fin n`, the finite set with `n` elements.

First, we designate a commutative monoid for the model structure in the fral. This structure is the data structure our simplifier will use to represent the equivalence classes of terms. In Fig. 1, we mentioned the carrier consists of origin-intercepting linear polynomials with `Nat` coefficients $p = a_1x_1 + \ldots + a_nx_n$, which we represent with `n`-tuples of natural numbers and pointwise addition:

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

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

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

Denote the resulting `CommutativeMonoid` by `Model n`. For the `Env` component, use tabulation to define `unit n : Fin n ⟶ Carrier n`, with `1` in the argument position and `0` elsewhere:

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

where[3]:

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

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

```
1  solveVect : {0 n : Nat} -> {pres : Presentation} -> {a : Model pres} ->
2    (frex : Frex a (irrelevantCast $ Fin n)) -> (env : Vect n (U a)) ->
3    (eq : ( Term pres.signature (U a `Either` Fin n)
4          , Term pres.signature (U a `Either` Fin n))) ->
5    {auto prf : frex.Data.Model.rel
6       (frex.Sem (fst eq) (frexEnv {x = cast $ Fin n} frex).H)
7       (frex.Sem (snd eq) (frexEnv {x = cast $ Fin n} frex).H)}
8       ->
9    a.rel (a.Sem (fst eq) (either Prelude.id (flip Vect.index env)))
10         (a.Sem (snd eq) (either Prelude.id (flip Vect.index env)))
```

Fig. 10. Core frex-based simplification routine

The initiality of this structure follows from the normal form property — every origin-intersecting linear polynomial p can be represented as $p = \sum_{i=1}^{n} a_i \cdot \texttt{unit n i}$:

```
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`) ⟶ `a` is this homomorphism:

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

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.

To summarise, to implement a fral/frex simplifier, the developer follows these steps:

- Design a data-structure for the carrier of the frex/fral's algebra, e.g. for commutative monoids: `Vect n Nat` for the fral and `(U a, Vect n Nat)` for the frex.
- Equip it with a setoid algebra structure: pointwise operations with propositional equality.
- Equip it with the appropriate additional structure, e.g. the unit for the fral and the `Var`iable function and the `Embed`ding homomorphism for the frex.
- Define the function underlying the homomorphism into any other algebra over the variable setoid or extension, e.g. linear combination for commutative monoids.
- Prove that this function is a homomorphism and its uniqueness.

5.2 Solver Implementation

We can now explain how Frex implements the `solve` functions. We describe the frex-based interface in detail; the fral-based interface is similar. We implement the core functionality in the auxiliary function `solveVect` in Fig. 10.

The argument `frex` (line 2) is an implementation of a frex simplifier for some `pres`-algebra `a`, extended with `n` free variables (line 1). We erase the number of variables at runtime, and so we also erase the type `Fin n`. We cast an erased type to a setoid instead of an unerased type, i.e.:

```
irrelevantCast : (0 a : Type) -> Setoid    instead of    cast : (a : Type) -> Setoid
```

The function also takes an `env`ironment of terms to substitute for the free variables in the simplification equation (line 2). In this auxiliary function, we present the environment using an `n`-ary vector of terms over the algebra's carrier. Next comes the `eq`uation we want to discharge (line 3),

```
data Visibility = Visible | Hidden | Auto

Pi : Visibility -> (a : Type) -> (a -> Type) -> Type
Pi Visible a b = (x : a) -> b x
Pi Hidden  a b = {x : a} -> b x
Pi Auto    a b = {auto x : a} -> b x

PI : (n : Nat) -> Visibility -> (a : Type) -> (Vect n a -> Type) -> Type
PI Z     vis a b = b []
PI (S n) vis a b = Pi vis a (\ x => PI n vis a (b . (x ::)))
```

Fig. 11. Metaprogramming abstractions for curried Π-types

involving either concrete values (of type `U a`) and any of the `n` available variables. Both the `frex` and the `a`lgebra with its `env`ironment give rise to extensions in the formal sense, which we can use to give an environment for the `eq`uation in question, namely a setoid homomorphism from the joint setoid of constants and free variables to the carrier of the model underlying the extension:

```
extEnv : {a : Model pres} -> {x : Setoid} -> (ext : Extension a x) ->
  Either (cast a) x ~> cast ext.Model
extEnv ext = either ext.Embed.H ext.Var
```

where: `either : {0 a, b, c : Setoid} -> (a ~> c) -> (b ~> c) -> (a `Either` b) ~> c`

We use these environments to interpret the equation, once in the `frex` (lines 6–7) and once in the given algebra (lines 9–10). If the equation holds in all extensions, it will hold in the `frex` and in `a`, and, moreover, homomorphisms of extensions will preserve this interpretation. Interpreting this equation in the `frex` may have better decidability properties over equivalence in `a`.

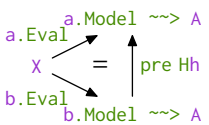
We use Idris2's `auto`-implicits mechanism to search for the equivalence of the `frex` interpretations. This mechanism will try to find terms that resolve the implicit argument `prf`, using a heuristic informed by unification, that will also attempt to apply data constructors.

Typically, Idris2's judgemental equality decides the `frex` setoid's equivalence relation, and the number of variables we extend by is known statically. This case can happen when the equivalence relation on the setoid algebra `a` is decidable by judgemental equality. Then, the type of the `prf` argument (line 5) is a propositional equality between closed terms. Judgemental equality decides this relation between the interpretations in the type of `prf`. In Idris2, the `auto`-search heuristic tries to use `Refl`, and promotes the required equation to a judgemental equality constraint. Even when the setoid relation is not decidable by judgemental equality, making `prf` an `auto`-implicit may provide more functionality in the future. We may be able to freely extend algebras whose propositional equality is only partially decidable by judgemental equality (e.g. function types in a type theory with function extensionality), or by a sophisticated decision procedure (e.g. multiset equality).

We use metaprogramming abstractions (Fig. 11) to simplify the user-facing interface. The `PI` combinator produces an `n`-ary telescope of `Visible`/`Hidden`/`Auto` arguments, packaged as an `n`-ary vector which it passes this its argument `b`. Using this abstraction to reduce `solve` (Fig. 12) to `solveVect` means unification can resolve the arguments to substitute for the free variables in the equation.

5.3 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 on the right, are an `X`-indexed collection of algebra homomorphisms `a.Eval f : a.Model ⟶ A`.



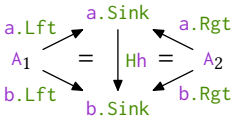
```
solve : (n : Nat) -> {pres : Presentation} -> {a : Model pres} ->
  (frex : Frex a (cast $ Fin n)) ->
  PI n Hidden (U a) $ (\ env =>
  (eq : ( Term pres.signature (U a `Either` Fin n)
        , Term pres.signature (U a `Either` Fin n))) ->
  {auto prf : frex.Data.Model.rel
     (frex.Sem (fst eq) (frexEnv frex).H)
     (frex.Sem (snd eq) (frexEnv frex).H)}
     ->
  a.rel (a.Sem (fst eq) (either Prelude.id (flip Vect.index env)))
        (a.Sem (snd eq) (either Prelude.id (flip Vect.index env))))
```

Fig. 12. User-facing frex-based simplification routine

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`.

5.4 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 in the diagram 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 cases such as commutative monoids, the coproduct is straightforward to represent: its carrier is the cartesian product of the components carriers.



The universal property of the frex `A[X]` combines those of the fral `Free 𝒯 X` and its coproduct with `A`. The fral's universality equates the left triangles:

This identification lets us 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)`.

5.5 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 𝒯 ∅` is also the initial algebra. We then calculate the frex:

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

Therefore, we may construct the fral from an initial algebra and its frex:

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

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

5.6 Reusing Frexlets

The final example demonstrates reuse of one simplifier when constructing another. Recall the presentation of involutive monoids from the end of Section 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, and have taken this strategy in Frex. However, it is also possible to phrase the result in much greater generality, and give a higher-level proof, using Jacobs's [2021] axiomatisation of involutions. This more abstract proof generalises to other notions of involutive algebras, and we plan to exploit it in the future for generic frexlet reuse. However, the more abstract proof goes beyond the scope of this manuscript, involving more abstract category theoretic notions. We include it in Appendix D for reviewing purposes.

6 Completeness and Certification

Frex uses setoids beyond a mere completion: it automatically extracts the proofs its simplifiers derive. Concretely, the fral and frex can be constructed by quotienting the term algebra with the provability equivalence relation. For the fral, the relation is provability with respect to the axioms of the theory, or a postulated equivalence in the given variables setoid. For the frex $\mathtt{A}[\mathtt{X}]$, we further postulate constants $\underline{a}$ for every element $a : \mathtt{U}\,\mathtt{A}$, 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 the rules of equational logic from Fig. 2.

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, and these frexlets are therefore *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 cannot 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, consider 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 $\mathtt{X} \uplus \mathtt{U}\,\mathtt{a} \vdash t = s$ involving variables and concrete elements in the algebra represented by terms over the disjoint union $\mathtt{X} \uplus \mathtt{U}\,\mathtt{a}$. The abstract frex in this situation is the term algebra over $\mathtt{X} \uplus \mathtt{U}\,\mathtt{a}$ 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. $\mathtt{A}[\![t]\!] = t$. Provability proofs between $\mathtt{A}[\![t]\!]$ and $\mathtt{A}[\![s]\!]$ are non-effective — all they amount to are deep embeddings of equational proofs in `a`, and `A` does not 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 $\mathtt{env} : \mathtt{X} \longrightarrow \mathtt{U}\,\mathtt{a}$, 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. 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.

At this point it is worth pausing and reflecting on this implementation. There is no need to write any proof extraction code for our simplifiers. The universal properties of the fral and the frex 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 nonetheless 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 type of linear/flat derivations that could be pretty-printed using Idris2's combinators for setoid reasoning.

We represent derivations in layers (cf. Fig. 17 in Appendix C): (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 [van der Ploeg and Kiselyov 2014] lists of steps in the closed-over relation: the target element of each element in the list is the source element of the next.
(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 equivalence, or one of the theory's axioms.
Putting these together, we get the type of derivations (cf. Fig. 17(e) in Appendix C).

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.

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

```
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. Goal extraction in (a) Idris2 Frex's elaborator reflection script; (b) the **fr**ex **Ag**da aug**ment**ation lib.

semantically irrelevant subterms that can be simplified away. Frex implements a generic proof simplifier that automatically removes all such loops. This mechanism suggests future investigation of certification modules that simplify these deeply embedded proofs further.

7 Goal Extraction via Reflection

Thus far, our examples have illustrated interaction with Frex using `solve`. The `solve` function provides a similar interface to the simplifiers in (e.g.) Agda's standard library: it takes the fral or frex simplifier, the number of free variables and the abstract syntax of a goal. However, these simplifiers additionally provide ergonomic goal extraction with Agda's *proof reflection* mechanism.

Proof reflection is a metaprogramming paradigm, available in proof assistants and dependently typed programming languages, that supports bi-directional communication between a language and its implementation. The language provides a representation of its terms, operators that construct, manipulate and destruct term representations, and primitives *quote* and *unquote* that respectively *reify* terms into the representation and *reflect* back encoded terms as ordinary terms.

Given mechanisms for querying unsolved proof obligations, proof reflection enables the implementation of verified decision procedures for automatically discharging such obligations without boilerplate [Boutin 1997; Christiansen and Brady 2016]. Coupled with the meta-theoretic properties that dependently typed implementations of decision procedures can enforce (e.g. relative soundness and completeness), reflection-driven interfaces yield easy-to-use tactics with strong guarantees. It is then natural to ask: is it possible to construct an interface to Frex that uses proof reflection to avoid the need to explicitly supply the equation to discharge?

As the example code in Fig. 13 illustrates, using proof reflection to provide such an interface to Frex is possible in both Idris2 and Agda. Rather than designing custom reflection-based drivers for individual simplifiers, we combine proof reflection with Frex's design philosophy of extensibility and common core reuse and provide a single generic metaprogram parameterized by a signature and a model of a presentation. The metaprogram can be instantiated for any algebraic simplifier, built-in or user-defined. Fig. 13 (a) shows the invocation of the Idris2 elaboration script `frexMagic`, and Fig. 13 (b) shows the Agda proof reflection macro `fragment`. Both implementations aim 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 argument ahead of time. Frex's inductive `Term` representation means that relevant abstract operator names can be extracted from the presentation. However, the process of 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.

The differences between the Idris2 and Agda implementations can be seen by considering the normalisation of arithmetic expressions such as `(x + 1) + y = x + (1 + y)`. In Idris2, the reflected syntax passed to the driver represents the normalized expression `(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 does not normalise quoted expressions before reflection, and so the Agda driver successfully finds the equation, allowing Frex to solve the example. Agda's approach is not always superior: it is possible

to construct similar examples where the Agda driver fails. The extent to which the implementation can avoid such pathologies ultimately depends on the engineering effort available to develop heuristics.

As these problems indicate, this rather naive approach to automation requires significant developer resources to deal with edge cases or construct bespoke solutions under simplifying assumptions. In large, mature ecosystems it may be possible to maintain practical heuristics despite these challenges. However, we think there might be better mechanisms for specifying algebraic contexts from which the solver can extract the required information automatically; we touch on some possible directions in Section 11.

8 Supplementary Evaluation: Usability and Interactive Development

The implementation of Frex is still in its early stages, and offers many opportuntities for further engineering work to extend its functionality, expressiveness, ergonomics, and efficiency. However, we have already carried out some small experiments to assess user experience and frexlet developer experience to establish that the approach is feasible, and to identify further directions.

8.1 Using Frex

Quantitative evaluation. Idris2 encourages interactive, type-driven development, so it is important that the checker is responsive when the user modifies the program. Following Nielsen [1993], our Idris2 implementation aims for response times under one second, and we treat a response time of over 10 seconds when type checking a modification to Frex client code as a bug.

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 14 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 10 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 (Section 9) and that the ongoing development of Idris2 may eventually eliminate the problem, bringing the type-checking time for most terms up to size 60 down to a few seconds.

Qualitative evaluation. To experience using Frex, we have reproduced Brady et al.'s [2007] dependently typed representation of binary arithmetic. Brady et al. index binary representations by the natural numbers that they represent, and so the programmer needs to give proofs for the correctness of arithmetic operations. These proofs typically interleave insightful equational reasoning steps with rote calculational steps such as the following:

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

which may be discharged by passing the equation to `solve`. We do not use our reflection capabilities since these kinds of examples, in which the binding-time analysis is challenging, are beyond their reach at the moment. With early implementations of Frex the task was arduous due to several performance bottlenecks in Idris2 that are now eliminated. The only other significant obstacle we encountered was the usual pain point involved in invoking an algebraic simplifier without a goal

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

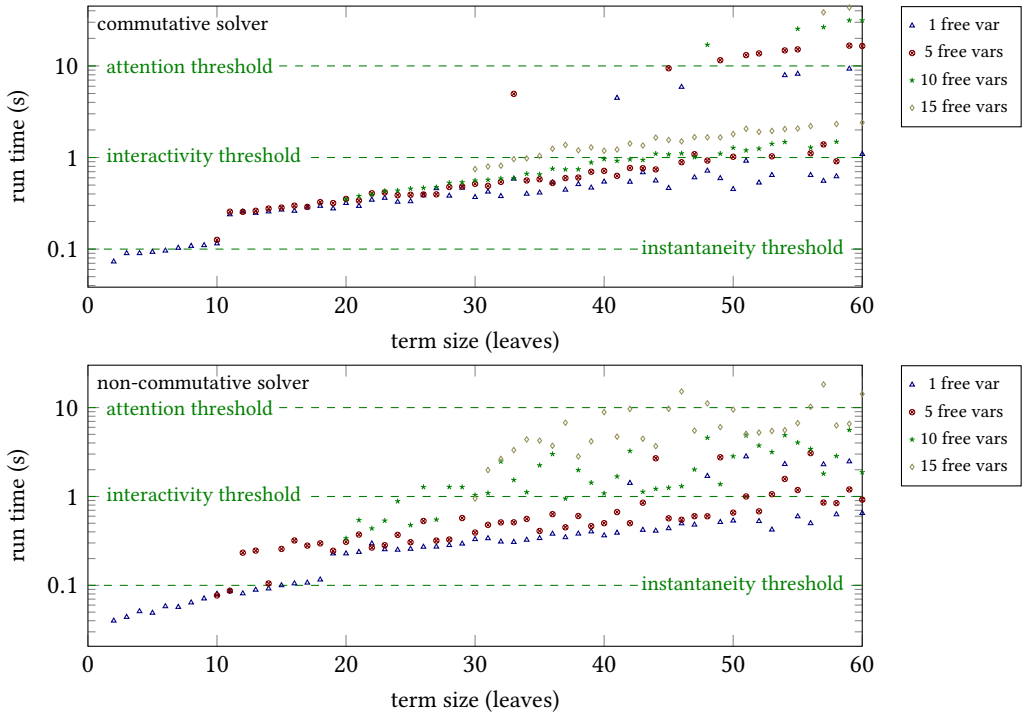


Fig. 14. Frex monoid simplifiers type-checking times

extraction mechanism: the need to repeat the equation and its relevant rewriting-context when calling Frex.

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 an 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 an 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 that a more modular construction might be possible. Within two days, the developer found Jacobs's axiomatisation of involutive algebras [Jacobs 2021], 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 five minutes or longer to type check. To work around the bottleneck, the developer proceeded with a three buffer approach, using using separate buffers for definitions that are completed, currently checked and under construction. Later, when preparing this manuscript, the developer used the `ByFrex` construct to implement the involutive monoid fral from this frex. This step took an additional half-afternoon, mostly 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.

We view the experiment as noisy due to the effect of the type-checker bottleneck (now eliminated, as described in the next section), but draw some tentative conclusions. First, a two-week development time seems acceptable for a shipping component: the involutive monoid frexlet now forms part of Frex. Second, 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. Finally, although algebraic generalization delayed the development of the component, the result of the delay is pleasing: it appears that working with the generic representations in Frex encourages algebraic thinking that can lead to modular code and even to new theorems.

9 System Design Lessons

Frex uses generic and dependently typed programming techniques extensively, requiring significant type level computation that taxes the capabilities of the host 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, in the hope that developers of other systems will find them useful.

9.1 Idris2

At the heart of the type checker is an implementation of dynamic pattern unification [Gundry 2013; Miller 1992; Reed 2009], which instantiates implicit arguments, and a conversion checker, which checks whether two terms evaluate to the same reduct. Each of these components requires an evaluator. Idris2 uses a form of normalisation by evaluation [Berger and Schwichtenberg 1991] 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.

Profiling the Idris2 executable reveals that most performance bottlenecks we have encountered in developing Frex are caused by the evaluator. We have experimented with alternative implementations of the evaluator that compile terms using Scheme's backend and a glued representation of values [Chapman et al. 2005; Coquand and Dybjer 1997] rather than interpreting terms directly. 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! We have therefore also experimented with various ways to avoid evaluating terms, including preserving subterm sharing, choosing appropriate data representation in unification, and taking advantage of the typical structure of unification and conversion problems.

Preserving Sharing Instantiating implicit arguments in dependently typed programs often 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 [2019], we preserve sharing by introducing a metavariable for *every* implicit argument, inlining only when it is guaranteed that the definition cannot break sharing. Consequently, we inline a metavariable 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 Idris2 needs to postpone a unification problem if it is blocked due to an unsolved metavariable. When the metavariable is solved, Idris2 need to re-evaluate the terms being unified. Previously, Idris2 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 checking the convertibility of subterms involves evaluation. Fortunately, terms in blocked unification problems tend to differ at the heads rather than at deeply nested subterms. Therefore, we always check the heads of the values of corresponding `xi` and `yi` first, postponing the unification problem if any are unequal. This heuristic significantly improves performance, preventing a lot of unnecessary evaluation.

Influence on Language Design and Ecosystem The development of Frex has led to the implementation of a number of desirable language features 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, which are solved by a search procedure which uses constructors and functions marked as search hints. To assist the development of Frex and improve the readability of its code we have added the ability to mark *local* functions as search hints, allowing us to restrict the scope of the hints and so avoid excessive growth of the search space. 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 two languages arise from Agda's focus on proving versus Idris2's focus on programming. Idris2 currently uses a single *universe* [Palmgren 1998], allowing `Type : Type`, and is hence inconsistent by Girard's paradox. In contrast, Agda's well-developed predicative theory of universes avoids 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 [2021], we adopt a conservative set of compiler options (`--without-K --safe`). All our definitions are universe-polymorphic. This conservativity broadens the applicability of Frex in the Agda ecosystem by guaranteeing compatibility with all of Agda's various configurations, and further assures us about the correctness of Frex itself. Corbyn [2021] discusses these ideas in greater detail.

10 Related Work

Within the Coq ecosystem, an abundance of tactics enable algebraic simplification. Boutin's [1997] ring and field tactics[5] let programmers discharge proof obligations involving (and requiring!) addition, multiplication, and division operations. Strub's [2010] CoqMT 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 ensure preservation of good meta-theoretical properties, Strub extends only term level conversion. Implementations of Hilbert's Nullstellensatz theorem (Harrison's [2007] in HOL Light and Pottier's [2008] in Coq) help users discharge proof obligations involving polynomial equalities on a commutative integral domain.

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

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 does not deal with algebraic simplification directly, it may help in 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 [2017] and Slama [2018] implement a hierarchy of rewriting procedures for algebraic structures of increasing complexity. Though the procedures' completeness is not enforced by type as 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 and Brady's), and (3) FREX is extensible, with support for sufficiently motivated users to extend the library with bespoke solvers.

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 [2013] work, 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` and prove their construction sound and complete with respect to the extended equational theory.

In Agda, Cockx's [2020] and Cockx et al.'s [2021] '`--rewriting`' flag allows users to enrich the existing reduction relation with new rules. Their implementation goes beyond Allais et al.'s: it may restart stuck computations. They leave to future work the soundness of user-provided reduction rules, i.e. ensuring rules neither introduce nontermination nor break canonicity. Unlike our approach, neither Allais et al's nor Cockx et al's technique currently supports commutativity.

Implementing FREX required formalising the fragments of universal algebra needed for its architecture. Formalising more complete fragments of the theory is an active area of research, with recent contributions by Gunther et al. [2018] and Abel [2021] in Agda. Carette et al. [2020] generate Agda code for a comprehensive collection of multi-sorted algebraic theories and their associated machinery via a pre-processing phase from a much smaller description. Fiore and Szamozvancev [2022] similarly generate definitions for the significantly more general second-order abstract syntax in Agda via a preprocessing step, while formalising more of the meta-theory as library code. We consider it an open problem in this domain to include the concise information as first-class data in the meta-language while nonetheless enjoy the full ergonomics of hand-written, inlined, definitions.

The Meta-F⋆ language [Martínez et al. 2019] provides normalisation tactics for commutative monoids and semi-rings through its metaprogramming facilities. FREX's usage resembles these tactics' usage, and we hope a FREX port to F⋆ will make use of F⋆'s metaprogramming facilities to reduce some syntactic noise during goal extraction.

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. Preliminary evaluation shows that, despite a high level of abstraction, the resulting library is responsive, with

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

comparable functionality to other libraries, in a combination of features that no single existing library provides. Frex's unique design—the frex and the fral—offer new prospects and questions.

Yallop et al.'s [2018] partial evaluators include additional frexlets (e.g. abelian groups, semirings, distributive lattices). We plan to follow suit and port the remaining simplifiers, then conduct larger evaluation and comparison studies. The main challenge is that, unlike Yallop et al., we must mechanically prove that these frexlets are complete, which is costly. One 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-based goal extraction 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. However, since software engineers for dependently typed languages are a scarce resource, we plan to explore 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.

Another promising direction is *bootstrapping* of the Frex library using simplifier certification. Bootstrapping might start with a hierarchy of inefficient simplifiers that are easy to implement. Next, these simplifiers may then be used to develop a hierarchy of more efficient simplifiers and proof-simplifiers. Finally, the certification mechanism can extract proofs to complete the bootstrap.

We would also like to extend Frex's design beyond algebraic structures. More general notions of theories abound: multisorted, second-order/parameterised, and essentially algebraic. Supporting these may allow Frex to cover much more complex situations, such as decision procedures for first order theories (e.g. Presburger arithmetic, cf. Strub's [2010] CoqMT) normalisation-by-evaluation for fusion laws [Allais et al. 2013], and equational manipulation of big-operators [Bertot et al. 2008; Lau 2017; Markert 2015]. Note that Frex can already 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 making use of a rich category theory library like Hu and Carette's [2021] `agda-categories` might lead to a sleeker and even more modular Frex implementation. More specifically, we plan to explore a general treatment of involutive algebras following Jacobs [2021], and Power's *distributive tensor* of equational theories [Hyland and Power 2006; Power 2005] for a uniform treatment of semi-ring varieties. Instantiating each of the 6 semi-group varieties makes it possible to cover each instance of the following combinations:

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

and modularly construct $(2 + 3) \times 3 = 15$ semi-ring varieties, including rings and semirings. As this example shows, this kind of modular treatment can 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

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

work appeared as part of a short-abstract in TyDe'20 [Allais et al. 2020]. We are grateful to Jacques Carette, Donovan Crichton, Joey Eremondi, Sam Lindley, Conor McBride, James McKinna, Kasia Marek, Wojciech Nawrocki, and Robert Wright for useful discussions and suggestions, and to the anonymous referees of the various iterations of this manuscript for their insistence on improving its presentation.

References

Andreas Abel. 2021. Birkhoff's Completeness Theorem for Multi-Sorted Algebras Formalized in Agda. *CoRR* abs/2111.07936 (2021). arXiv:2111.07936 https://arxiv.org/abs/2111.07936

Andreas Abel, Klaus Aehlig, and Peter Dybjer. 2007a. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electronic Notes in Theoretical Computer Science* 173 (2007), 17–39. https://doi.org/10.1016/j.entcs.2007.02.025 Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII).

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2007b. Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 3–12. https://doi.org/10.1109/LICS.2007.33

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 33:1–33:30. https://doi.org/10.1145/3110277

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).

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

Thorsten. Altenkirch, Peter. Dybjer, Martin Hofmann, and Philip Scott. 2001. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 303–310. https://doi.org/10.1109/LICS.2001.932506

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–199.

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

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

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..

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

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.

Ilya Beylin and Peter Dybjer. 1996. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 47–61.

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

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.

Jacques Carette, William M. Farmer, and Yasmine Sharoda. 2020. Leveraging the Information Contained in Theory Presentations. In *Intelligent Computer Mathematics*, Christoph Benzmüller and Bruce Miller (Eds.). Springer International

Publishing, Cham, 55–70.
James Chapman, Thorsten Altenkirch, and Conor McBride. 2005. Epigram Reloaded: A Standalone Typechecker for {ETT}. *Trends in Functional Programming* (2005).
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
Jesper Cockx. 2020. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In *25th International Conference on Types for Proofs and Programs (TYPES 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 175)*, Marc Bezem and Assia Mahboubi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:27. https://doi.org/10.4230/LIPIcs.TYPES.2019.2
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
Thierry Coquand and Peter Dybjer. 1997. Intuitionistic Model Constructions and Normalization Proofs. *Math. Struct. Comput. Sci.* 7, 1 (1997), 75–94. https://doi.org/10.1017/S0960129596002150
Nathan Corbyn. 2021. *Proof Synthesis with Free Extensions in Intensional Type Theory.* Technical Report. University of Cambridge. MEng Dissertation.
Djordje Čubrić, Peter Dybjer, and Philip Scott. 1998. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science* 8, 2 (1998), 153–192. https://doi.org/10.1017/S0960129597002508
Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498715
Benjamin Grégoire and Assia Mahboubi. 2005. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–113.
Jason Gross, Adam Chlipala, and David I. Spivak. 2014. Experience Implementing a Performant Category-Theory Library in Coq. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 275–291.
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
Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. 2018. Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science* 338 (2018), 147–166. https://doi.org/10.1016/j.entcs.2018.10.010 The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).
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
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
Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by evaluation for modal dependent type theory. *J. Funct. Program.* 33 (2023). https://doi.org/10.1017/S0956796823000060
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.
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.
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
Donnacha Oisín Kidney. 2019. *Automatically and Efficiently Illustrating Polynomial Equalities in Agda.* Technical Report. University College Cork. BSc Dissertation.
András Kovács. 2019. Fast Elaboration for Dependent Type Theories. Talk at EU Types WG meeting, 2019.
Stella Lau. 2017. Theory and implementation of a general framework for big operators in Agda. Bachelor's thesis, University of Cambridge.
Leonhard Markert. 2015. *Big operators in Agda.* Master's thesis. MSc thesis, University of Cambridge.
Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1
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

Conor McBride. 2016. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

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

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

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

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

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.

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

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

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.

Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–15. https://doi.org/10.1109/LICS52264.2021.9470719

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

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

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

- *Frex* (§4–§5.4): core definitions
  - *Signature* (§4): operations & arities
  - *Algebra* (§4,§5.1): algebraic structures and terms, homomorphisms
  - *Presentation* (§4): axioms, equational theories
  - *Axiom* (§4): common axiom schemes
  - *Model* (§4): axiom-validating algebras
  - *Powers* (§5.3): parameterised algebras
  - *Free* (§4): simplification in all algebras
    - *Definition* (§5.1): universal property
    - *Construction* (§6): a non-effective quotient construction used for extraction, printing, and certification
      - *ByFrex* (§5.5): reuse a frex simplifier to define a fral simplifier
      - *Linear* (§6.1–§6.2): generic proof simplification and printing
      - *Idris* (§6): generic certification
  - *Coproduct* (§5.4): universal property
  - *Frex* (§5.1–§5.4): universal property, reuse coproduct and fral simplifier to define a frex simplifier
    - *Construction* (§6): non-effective quotient construction used for extraction, printing, and certification
  - *Lemma* (§6): auxiliary representation for auxiliary lemmata discharged by fral simplifiers, printed, or certified
  - *Magic* (§7): generic reflection code for ergonomic invocation
- *Frexlet.Monoid*: modules concerning varieties of monoids and their simplifiers
  - *Theory* (§4): signature, axioms, pretty printing for the theory of ordinary monoids
  - *Notation* (§4): shared infix notation (additive and multiplicative) for monoid varieties
  - *Frex* (Fig. 1): frex simplifier for monoids
  - *Free* (§5.5): fral simplifier, reuses frex simplifier
  - *Nat* (§4): additive and multiplicative monoid structure of the natural numbers
  - *Pair*: types with the cartesian product as a proof-relevant monoid structure
  - *List*: monoid structure of lists with catenation
  - *Commutative*: commutative monoids modules
    - *Theory*: commutativity axiom
    - *Free* (§5.1:) fral simplifier
    - *NatSemiLinear* (§5.1): auxiliary definitions for fral simplifier
    - *Frex* (§5.4): simplifier, reuses fral via coproducts
    - *Coproduct* (§5.4): coproduct of commutative monoids
    - *Nat*: addition and multiplication of naturals
  - *Involutive*: modules concerning monoids equipped with an involution
    - *Theory* (§4): signature and axioms
    - *Free* (§8.2): simplifier, reuses frex simplifier
    - *Frex* (§5.6): simplifier, reuses monoid frex
    - *List* (§4): involutive monoid structure of list reversal

Fig. 15. Overview of the core Frex code-base and its relationship to this manuscript

A Module Structure of Frex

Fig. 15 summarises the core modules in this codebase and their relationship to this manuscript.

B Extensional Function and Quotient Setoids

Figure 16a defines the quotient of a type by a function `q`, 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 4–6, e.g.), possibly with quantities, named-argument function calls (lines 8–15, e.g.), application operator `$`, 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`. These underscores mean elaboration must fill-in the blanks uniquely using unification.

Figure 16b 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)

```
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   }
```

```
0 (.VectEquality) : (a : Setoid) -> Rel (Vect n (U a))
a.VectEquality xs ys = (i : Fin n) ->
  a.equivalence.relation (index i xs) (index i ys)

VectSetoid : (n : Nat) -> (a : Setoid) -> Setoid
VectSetoid n a = MkSetoid (Vect n (U a))
  $ MkEquivalence
  { relation   = (.VectEquality) a
  , reflexive  = \xs                    , i =>
      a.equivalence.reflexive  _
  , symmetric  = \xs,ys, prf            , i =>
      a.equivalence.symmetric  _ _   (prf  i)
  , transitive = \xs, ys, zs, prf1, prf2, i =>
      a.equivalence.transitive _ _ _ (prf1 i) (prf2 i)
  }

VectMap : {a, b : Setoid} -> (a ~~> b) ~>
  (VectSetoid n a ~~> VectSetoid n b)
VectMap = MkSetoidHomomorphism
  (\f => MkSetoidHomomorphism
    (\xs => map f.H xs)
    $ \xs, ys, prf, i  => CalcWith b $
    |~ index i (map f.H xs)
    ~~ f.H (index i xs)
                        .=.(indexNaturality _ _ _)
    ~~ f.H (index i ys) ...(f.homomorphic _ _ $ prf i)
    ~~ index i (map f.H ys)
                          .=<(indexNaturality _ _ _))
  $ \f,g,prf,xs,i => CalcWith b $
  |~ index i (map f.H xs)
  ~~ f.H (index i xs) .=.(indexNaturality _ _ _)
  ~~ g.H (index i xs) ...(prf _)
  ~~ index i (map g.H xs) .=<(indexNaturality _ _ _)
```

Fig. 16. (a) Quotient, function-space, and (b) vector setoids (top) and a higher-order homomorphism (bottom)

`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).

C Proof Printing and Certification

Figure 17 presents the layered representation of linear derivations.

Fig. 18 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

```
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

```
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

```
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

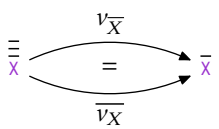
Fig. 17. Layered (a–d) representation of linear derivations (e)

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. 19 shows an automatically extracted certificate for the equation $0 + (x + 0) + 0 = x$ in a generic monoid `m` $= (\texttt{U m}, \texttt{0}, (\texttt{.+.}))$. The certificate is generated inside a module that parameterises over the generic monoid `m` and introduces the various notations and reasoning functions.

D Modularity with Involutive Algebras

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* [Baez and Dolan 1998] — an algebraic structure on an object requires a compatible structure on its category of context—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



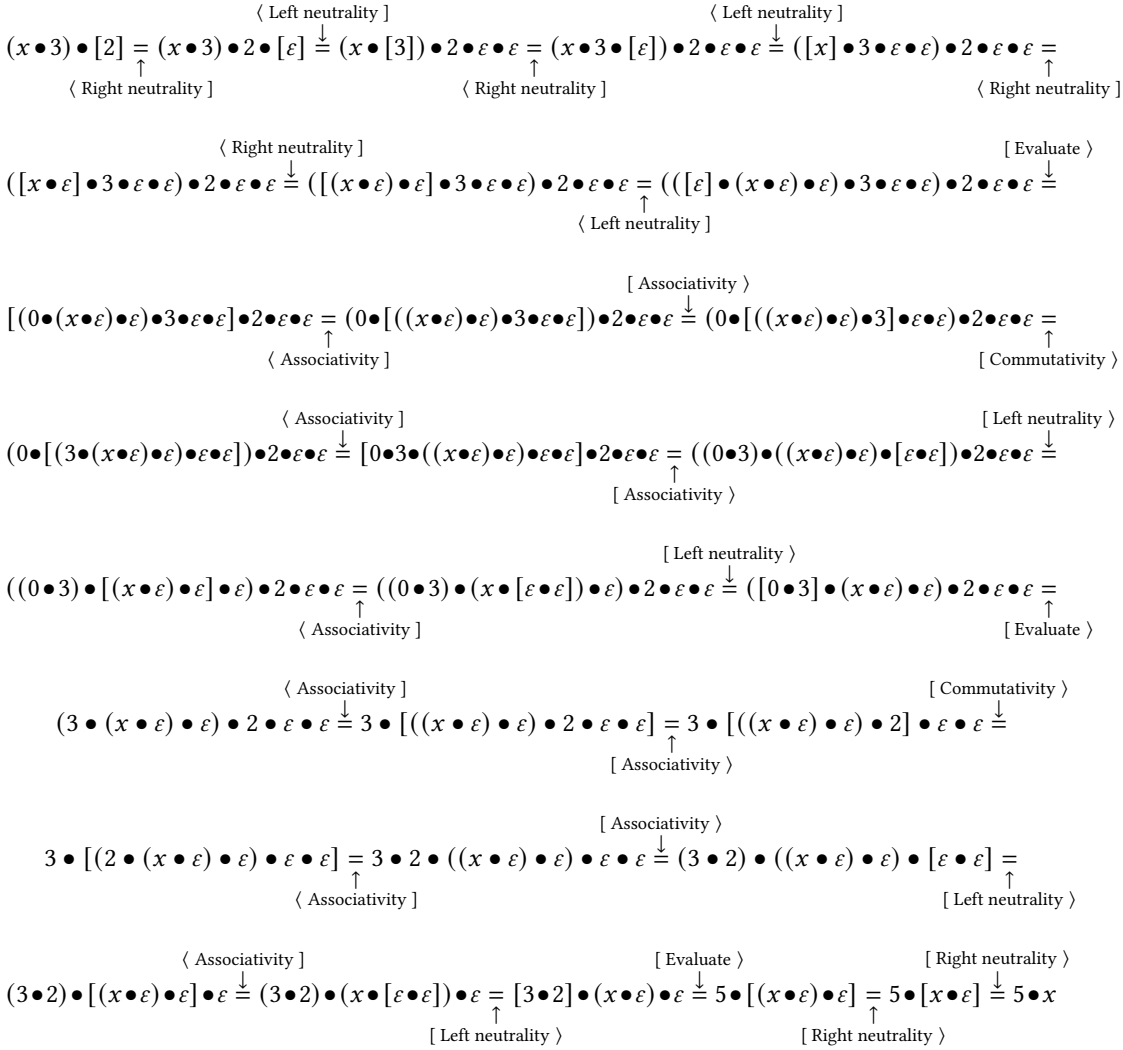


Fig. 18. Frex-extracted proof of $(x \bullet 3) \bullet 2 = 5 \bullet x$ in the additive monoid over Nat

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.

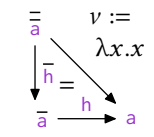
The motivating example is **Monoid**, the category of monoids. It has the following non-trivial involutive structure. Given a monoid a, construct another monoid $\overline{\mathtt{a}}$ with the operation reversed: $\overline{\mathtt{a}}\,[\![\cdot]\!]\,(x, y) := \mathtt{a}\,[\![\cdot]\!]\,(y, x)$. If $\mathtt{h} : \mathtt{a} \longrightarrow \mathtt{b}$ is a monoid homomorphism, then the same underlying function provides a monoid homomorphism $\overline{\mathtt{h}} : \overline{\mathtt{a}} \longrightarrow \overline{\mathtt{b}}$. These maps define an involution functor $\overline{(-)}$: **Monoid** $\longrightarrow$ **Monoid**. The identity function is then a monoid isomorphism $\nu := (\lambda x.x) : \overline{\overline{\mathtt{a}}} \longrightarrow \mathtt{a}$,

```
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 )
```

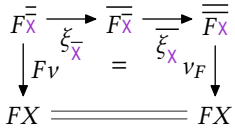
Fig. 19. Frex-certificate for the of $0 + (x + 0) + 0 = x$ in a generic monoid `m`

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 $h : U\,a \to U\,a$ makes a monoid a into an involutive monoid if and only if (1) it is a monoid homomorphism $h : \bar{a} \to a$, so $h(x \cdot y) = h\,y \cdot h\,x$, and (2) the diagram on the right commutes, so $h(h\,x) = x$. 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.

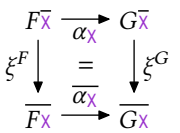


Packaging this structure, an *involutive category* $\mathcal{C} = (\mathcal{C}_o, \overline{(-)}, v)$ is an ordinary category $\mathcal{C}_o$ equipped with an involutive structure $(\overline{(-)}, v)$. An *involutive functor* $F : \mathcal{B} \to \mathcal{C}$ between involutive categories is a pair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



The canonical example is the forgetful functor $U : \mathrm{Model}\mathcal{T} \to \mathbf{Set}$ 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 $\mathrm{Model}\mathcal{T}$, 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 $\alpha : F_o \to G_o$ between their underlying ordinary functors that moreover satisfies the condition on the right.



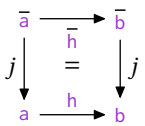
As Jacobs comments, we therefore have a 2-category **ICat** consisting of involutive categories, functors, and natural transformations, and we may derive involutive adjunctions as two involutive functors and two involutive natural transformations satisfying the triangle laws.

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

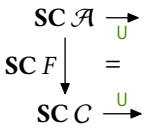
Proposition D.1. *Let $G : \mathcal{A} \to C$ be an involutive functor, and $F_o \dashv G_o$ be a left-adjoint to the ordinary functor underlying G with unit η and counit ε. Set $\xi^F_X : F_o\overline{X} \to \overline{F_oX}$ to be the mate of the composite $\overline{X} \xrightarrow{\overline{\eta}} \overline{G_oF_oX} \xrightarrow{(\xi^G)^{-1}} G_o\overline{F_oX}$. 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: $\lambda x.\eta x : \overline{X} = X \xrightarrow{\overline{\eta}=\eta} UFX \xrightarrow{\xi^U=\lambda t.t} U\overline{FX}$. 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 $\overline{FX}$, 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.

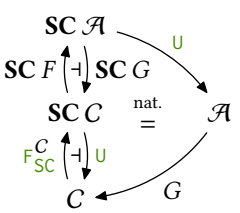
A *self-conjugate* object a in an involutive category $\mathcal{A}$ is a pair (a_{obj}, j_a) consisting of an object a_{obj} in $\mathcal{A}$, and an $\mathcal{A}$-morphism $j_A : \overline{a_{obj}} \to a_{obj}$, satisfying the triangle on the right. As we saw on p. 33, self-conjugate monoids are involutive monoids, and more generally, self-conjugate semi-groups, groups, semi-rings, rings, etc. are the involutive ones. A *homomorphism* $h : a \to b$ of self-conjugate objects is a homomorphism $h : a_{obj} \to b_{obj}$ 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 $\mathbf{SC}\,\mathcal{A}$. Jacobs shows that the forgetful functor $U : \mathbf{SC}\,\mathbf{Set} \to \mathbf{Set}$ has a left adjoint $F_{SC} : \mathbf{Set} \to \mathbf{SC}\,\mathbf{Set}$ sending each set 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 $F_{SC}X := ((\mathsf{Bool}, X), \lambda(b, x).(\neg b, x))$.



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 $\mathbf{SC} : \mathbf{ICat} \to \mathbf{ICat}$. We recall the remaining structure. The action on objects of **ICat** equips the category $\mathbf{SC}\,\mathcal{A}$ with an involutive structure, sending each self-conjugate object a to the self-conjugate object $\overline{a} := (\overline{a_{obj}}, \overline{j_a} : \overline{\overline{a_{obj}}} \to \overline{a_{obj}})$. The action on the morphisms of **ICat**, sends an involutive functor $F : \mathcal{B} \to C$ to the involutive functor $\mathbf{SC}\,F : \mathbf{SC}\,\mathcal{B} \to \mathbf{SC}\,C$ mapping each self-conjugate object a to the self-conjugate object $(F_o a_{obj}, j_{\mathbf{SC}\,Fa} : \overline{F_o a_{obj}} \xrightarrow{(\xi^F)^{-1}} F_o\overline{a_{obj}} \xrightarrow{F_o j_a})$, 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 preserves the resulting self-conjugated structure. The forgetful functor $U : \mathbf{SC}\,\mathcal{A} \to \mathcal{A}$ 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 $\mathtt{F}^{C}_{\mathtt{SC}} \dashv \mathtt{U} : \mathbf{SC}\, C \to C$, we get the free self-conjugate $\mathcal{A}$-object on $\mathtt{X}$ as the composite $F(\mathtt{F}^{C}_{\mathtt{SC}}\mathtt{X})$ completely structurally, as on the right. Applying this result to frals, we get the following generalisation of Prop. 5.1(fral):

Proposition D.2. *Let $\mathcal{T}$ be a presentation equipped with an involutive structure over* $\mathtt{Model}\mathcal{T}$ *and an involutive forgetful functor structure. Let* $(\mathtt{A}, \mathtt{Env})$ *be any free $\mathcal{T}$ model over the product* $(\mathtt{Bool}, \mathtt{X})$*. Then the following structure exhibits* $\mathtt{A}$ *as the free self-conjugate $\mathcal{T}$-model over* $\mathtt{X}$*:*

$$j_{\mathtt{A}} : \overline{\mathtt{A}} \xrightarrow{\xi^{-1}} \mathtt{A} \xrightarrow{\gg=(\neg\times\mathtt{id})} \mathtt{A} \qquad \mathtt{Env}' : \mathtt{X} \xrightarrow{\lambda\mathtt{x}.(\mathtt{False},\mathtt{x})} (\mathtt{Bool},\mathtt{X})$$

$$(\gg=' \mathtt{f}) : \mathtt{A} \xrightarrow{\gg=\lambda(\mathtt{b},\mathtt{x}).\begin{cases}\mathtt{b} = \mathtt{True} : & j_{\mathtt{a}}(\mathtt{f}\,\mathtt{x}) \\ \mathtt{b} = \mathtt{False} : & \mathtt{f}\,\mathtt{x}\end{cases}} \mathtt{a}$$

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

Proposition D.3. *Let $\mathcal{T}$ be a presentation equipped with an involutive structure over* $\mathtt{Model}\mathcal{T}$ *and an involutive forgetful functor structure, and* $\mathtt{a}$ *be a self-conjugate $\mathcal{T}$-model. Let* $(\mathtt{A}, \mathtt{Var}, \mathtt{Embed})$ *be any $\mathcal{T}$-frex of* $\mathtt{a}_{\mathtt{obj}}$ *by the product* $(\mathtt{Bool},\mathtt{X})$*. Then the following structure exhibits* $\mathtt{A}$ *as the frex of the self-conjugate $\mathcal{T}$-model* $\mathtt{a}$ *by* $\mathtt{X}$*, for* $\mathtt{h} : \mathtt{a} \to \mathtt{c}$ *involutive monoid homomorphism and function* $\mathtt{e} : \mathtt{X} \to \mathtt{c}$*:*

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

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

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

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 $\mathtt{Model}\mathcal{T}'$ 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.