# Frex: indexing modulo equations with free extensions

Guillaume Allais, Edwin Brady
{gxa1,ecb10}@st-andrews.ac.uk
University of St Andrews

Ohad Kammar
ohad.kammar@ed.ac.uk
University of Edinburgh

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
University of Cambridge

We report about the ongoing development of a library for dependently-typed programming with computations in index positions. Such indexing leads to notoriously difficult unification problems. Here we combine the established 'fording' technique (§2) with our work on *free extensions (frex)* developed for staged optimisation in OCaml and Haskell [22].

In brief, fording improves the judgmental-propositional communication channel for equations while Frex provides an extensible collection of algebraic solvers for discharging these equations. We present our design in Idris2; we would like to pursue similar development in other type theories.

## 1 Indexing with computations: the cons

To see how computations in indices go wrong, consider `Alt`, a datatype of lists of values of alternating types, indexed by:

- `even`,`odd`: the two alternating types at each parity
- `start`: the parity of the first element
- `parity`: the parity of the list's length

```
1 data Alt : (even,odd : Type)
2         -> (start, parity : Fin 2) -> Type where
3   Nil  : Alt even odd start 0
4   (::) : (x  : Choose even odd start)
5       -> (xs : Alt even odd (1 + start) parity)
6       -> Alt even odd start (1 + parity)
```

Here `Fin 2` is the finite type with two values (`0`, `1`), and `Choose` chooses one of two types depending on a parity bit:

```
Choose : (even,odd : Type) -> Fin 2 -> Type
Choose even odd 0 = even
Choose even odd 1 = odd
```

Here is a value in `Alt` with `Bool` and `String` elements:

```
Example1 : Alt Bool String 0 0
Example1 = [True, "TyDe", False, "Idris2"]
```

To complete `Alt`'s definition, we need to define `+` on `Fin 2`:

```
-- binary modular         mod2       Z     = 0
-- addition               mod2 (  S Z )  = 1
mod2 : Nat -> Fin 2       mod2 (S(S n)) = mod2 n

(+) : Fin 2 -> Fin 2 -> Fin 2
(+) x y = mod2 ((finToNat x) + (finToNat y))
```

Defining `(+)` this way, our choice to use `(+)` on lines 5 and 6 in `Alt`'s definition has well-known disastrous consequences. The main cause is using an open term such as `(1 + start)` for indices. This term reduces to `(mod2 (S (finToNat y)))`, a stuck computation and not an open value. The definition of `(+)` is unnecessarily complicated here, but in general we expect complicated functions as indices.

The problems start when we use `Alt`, e.g. in concatenation:

```
(++) : Alt even odd start left
    -> Alt even odd (start + left)   right
    -> Alt even odd  start   (left + right)
```

It would be natural to define `(++)` inductively:

```
[]        ++ ys =              ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

but these clauses are ill-typed. In the recursive call `(xs ++ ?a)`, the type of the hole `?a` and the actual type of `ys` don't unify:

```
ys: Alt even odd (start + (1 + parity)) right
?a: Alt even odd ((1 + start) + parity) right
```

We must therefore prove:

```
lemma: (x,y : Fin 2) -> x+(1+y) = (1+x)+y
```

and use rewriting mechanisms to inform the type-checker of it. As we will see, fording (§2) makes this rewriting more systematic. The full definition is in Fig. 2 in the appendix, and it also uses the four axioms of commutative monoids:

```
lftNeutral : (x : Fin 2) -> 0 + x = x
rgtNeutral : (x : Fin 2) -> x + 0 = x
associative: (x,y,z : Fin 2)-> (x+y)+z = x+(y+z)
commutative: (x,y   : Fin 2)-> x + y = y + x
```

What is vexing is that `lemma` easily follows from these four axioms, but still requires explicit proof. In general, we expect many more proof obligations like `lemma`, and we will need to prove them separately. Our contribution is a library (Frex) to discharge auxiliary equations like `lemma` immediately.

## 2 Fording

The standard technique *fording*[1] replaces a computational index `f x` by a fresh variable `y` and a propositional equality `y = f x`. For example, fording `Alt` gives:

```
(::) : forall e, o, start, parity, p, q.
 Choose e o start -> Alt e o p parity
 -> {auto 0 prf1: p=1+start}
 -> {auto 0 prf2: q=1+parity} -> Alt e o start q
```

Fording tells the type-checker not to bother discharging the equation judgmentally but instead ask the programmer for it. The Idris2 keyword `auto` gives the programmer a chance to punt this question back to the type-checker, which will try to insert `Refl` and resolve the equation judgmentally. The annotation `0` is a *quantity* annotation [2, 13], telling Idris to erase this argument at runtime. So fording in Idris2 has

[1] McBride [12, §3.5] names fording after Henry Ford's quote: 'any color so long as it's black' [8].

lower finger-typing cost and no run time costs compared to languages without implicit proof search and quantities.

Pattern-matching in a forded type introduces adverse 'noise' when judgmental equality can inform the type-checker through unification. In this case, the programmer inserts reflection manually (or transport in high dimensional type theories). Punting arbitrary equations back to the type-checker could encode arbitrary word problems. Therefore, any hypothetical fully automated solution would require careful analysis of the equations fording produces. We're interested in reducing this fording noise nonetheless.

## 3 Frex: free extensions of algebras

We want to show the type-checker that two terms, such as `start + (1 + parity)` and `(1 + start) + parity`, are equal. The type-checker's automatic *judgmental* equality is too crude (§1): it is unaware of the equations governing `(+)`. As we saw, fording (§2) turns judgmental checks into *propositional* obligations that can be discharged manually, making it possible to use those equations. We now show how to discharge the propositional obligations uniformly by encoding a third notion of equality: equality in a freely extended algebra.

To stay concrete, we discuss only commutative monoids, i.e. types with a binary operation `(+)` and a constant `0` satisfying analogues of `lftNeutral`, `rgtNeutral`, `associative`, and `commutative`. Our library deals with arbitrary such finite *presentations* (finitely many operations with finite arities and equations between them). Given an algebra `a` (i.e., commutative monoid), its *free extension* by `x`, written `a[x]` is the algebra resulting by freely adjoining `x` elements to `a`. For commutative monoids, the free extension `a[Fin n]` can be given by the product `(a, Vect n Nat)`, using `(v, [k1,...,kn])` to represent `v+k1*x1+...+kn*xn`.

We've implemented *Core Frex*, a formalisation of universal algebra (presentations, algebras, homomorphisms) and free extensions, together with supporting definitions that make it easier to define, and prove the universal property of, concrete presentations, algebras, and free extensions, which we call *frexlets*. We've only implemented the commutative monoids frexlet in full, but plan to add frexlets for other presentations we previously designed [22], including commutative rings, semirings, abelian groups, and distributive lattices.

Frex makes substantial use of type-level computation, which is supported efficiently by the nascent *Idris2* compiler. Frex is one of the first substantial Idris2 programs (around 4.3KLoC) alongside Idris2 itself, which is self-hosted.

In universal algebraic terms, we can present the free extension by: (1) taking as generators the elements of the concrete algebra and the adjoined elements (variables); and (2) taking as equations the presentation together with the evaluation equations. For example, the free extension `Bool[Fin 2]`, resulting from extending the Booleans with logical conjunction `(&&)` by adjoining two elements, has as generators `True`, `False`, `0`, `1`, and as equations the commutative monoid axioms together with `True && False = False`, etc. So we can see Frex as a normalisation-by-evaluation technique for algebraic theories. Abstracting over free extensions, instead of presentations, lets us treat uniformly *all* algebras.

## 4 Indexing modulo equations

To use Frex for indexing modulo equations, the programmer fords their computational indices. When they need derivable equations such as `lemma`, they invoke the `Frexify` function (Fig. 1 in the appendix) with the appropriate frexlet to discharge these equations. The `auto` argument punts the proof that the two sides of the equation have the same frexlet interpretation back to the type-checker. For example:

```
(++) xs ys {prf1 =
  Frexify (frex _) [start, parity]
   (var 0 :+: (sta 1  :+: var 1) =-=
   (sta 1 :+:  var 0) :+: var 1)}
```

Idris2 `auto` finds the `(1, [1, 1])=(1, [1, 1])` argument, representing the shared normal form $1 + 1 \cdot x_0 + 1 \cdot x_1$. We include the full code for `(++)` in Fig. 3 in the appendix. With Frex, programmers could focus on algebraic axioms for their computations of interest, like the commutative monoid axioms, and discharge derivable equations with low cost.

One alternative to indexing modulo equations is to calculate an inductive representation of the quotient datatype. Appendix 5 has a more thorough survey of existing approaches. A promising difference Frex offers is that we only use new operations and equations when we need them when defining operations on the datatype. As a consequence, we can establish the equations as they are needed, and use only the frexlet for the subset of operations we need to discharge each equation. Were we to represent the quotient inductively, we would need multiple representations and coercions between them, or a combined monolithic representation accounting for all possible operations and equations.

## 5 Prospects

As a first step, we plan to extend Frex with the full set of frexlets from our previous work [22], and use them to index datatypes and operations on them like matrix manipulation libraries. We expect many auxiliary equational results are needed for such libraries, and hope Frex can ease writing them. Next, we would like to investigate how to use Frex to directly inform unification, so that, for example, the terms `(?x + 1) + 1` and `(?y + ?z) + 3` unify to give `?x = S (?y + ?z)`. Finally, we are interested in providing Frex in other dependently-typed languages (Agda, Coq, Lean, F★, etc.), and we hope a presentation in TyDe could help us find collaborators for this purpose.

```
Frexify : {n : Nat} -> {pres : Presentation} -> {a : Model pres}
  -> (frex : Frex pres a (Fin n)) -> (env : Vect n (U a))
  -> (eq : (Term (Sig pres) (Either (U a) (Fin n))
           ,Term (Sig pres) (Either (U a) (Fin n))))
  -> {auto prf : frexSem frex     (fst eq) = frexSem frex     (snd eq)}
  ->           ( algSem frex env (fst eq) =  algSem frex env (snd eq))
```

**Figure 1.** API to the frex algebraic solver

```
(++) {right} {start} [] ys
  = rewrite sym (rgtNeutral start) in
    rewrite      lftNeutral right  in ys

(++) {even=e} {odd=o} {start} {right}
     ((::) {parity} x xs) ys = vs
where
  zs : Alt e o ((1 + start) + parity) right
  zs = rewrite sym (lemma start parity) in ys
  ws : Alt e o start (1 + (parity + right))
  ws = x :: (xs ++ zs)
  vs : Alt e o start  ((1 + parity) + right)
  vs = rewrite associative 1 parity right in ws
```

**Figure 2.** Concatenation with naive indexing by computations

## Appendix: Existing approaches

Existing approaches either avoid indexing by computations, discharge equations judgmentally, or propositionally.

***Slime avoidance.*** McBride calls indexing by computations 'green slime' [14], as his preferred colour scheme for user-defined functions is green, and indexing by computations saturates the program with more green proofs about these indices. Instead, McBride advocates finding inductive representations approximating these computations-modulo-equations, and index by these inductively defined values. To bridge the gap between the inductive indices and the true quotient, one uses McBride-McKinna views [15] to get open-terms unstuck. The resulting design is extremely elegant and appealing, and plays seamlessly with the type-checker, unifier, and interactive editing tools, enabling the so-called 'banzai programming', where one repeatedly, blindly, and satisfyingly assaults function definitions with repeated automatic pattern-matching, refinement, and proof-search.

The main challenge slime avoiding design poses is that it's difficult to get right. The designer can spend years working out exactly what to index by. Since the computation-indexed program is exactly what we are trying to avoid, it is difficult to know in advance what we will need to quotient by. A secondary challenge is that bespoke indexing hinders code-reuse, as we need to re-implemented existing functions for our special-purpose inductive index types. Ornamentation[2] [5–7] with its many applications [10, 20, 21] can help overcome some of this challenge.

***Enriched judgmental equality.*** Allais et al. [1] demonstrate 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 laws for list as well as the fusion laws describing the interactions of `fold`, `map`, and `append`. They prove their construction sound and complete with respect to the extended equational theory.

Cockx's extension of Agda with the '`--rewriting`' flag [4] allows users to enrich the existing reduction relation with new rules. This work goes beyond Allais', since Cockx may restart stuck computations. The question of guaranteeing the soundness of user-provided reduction rules by ensuring they neither introduce non-termination nor break canonicity is left to future work. Concretely comparing both Allais et al. and Cockx's techniques to our proposed technique, neither currently deals with commutativity.

Strub's CoqMT [19] extends Coq's Calculus of Inductive Constructions, allowing users to extend the conversion rule with arbitrary decision procedures for first order theories (e.g. Presburger arithmetic). To guarantee that this extension preserves good meta-theoretical properties, Strub only extends term level conversion. This seems incompatible with our preferred approach to systematically index data and perform type-level conversion.

***Algebraic solvers.*** The other approach is to bite the bullet, write out the many proofs resulting from indexing by computations, using automation to ease the task whenever is possible. These tend to be bespoke to the project at hand, but also include some general reusable libraries.

Within the Coq ecosystem, a plethora of tactics provide such automation. Boutin's ring [3] and field tactics[3] let programmers discharge proof obligations involving (and requiring!) addition, multiplication, and division operations. Implementations of Hilbert's Nullstellensatz theorem (Harrison's

[2] Conor McBride, *Ornamental algebras, algebraic ornaments*, unpublished.
[3] See the Coq documentation:
https://coq.inria.fr/distrib/current/refman/addendum/ring.html

```
data Alt : (even,odd : Type)
       -> (start, parity : Fin 2) -> Type where
  Nil  : forall even, odd, start, p .
         Alt even odd start FZ
  (::) : forall even, odd, start, parity, p, q.
         Choose even odd start
      -> Alt even odd p parity
      -> {auto 0 prf1 : p = 1 + start}
      -> {auto 0 prf2 : q = 1 + parity}
      -> Alt even odd start q
```

```
(++) : forall even, odd, start, parity_left,
         parity_right, p, q.
     Alt even odd start parity_left
  -> Alt even odd p parity_right
  -> {auto 0 prf1 : p = start + parity_left}
  -> {auto 0 prf2 : q = parity_left + parity_right}
  -> Alt even odd start q
```

**(a)** fording with runtime-irrelevant Idris2 **auto**-implicits

```
(++) {parity_right} {start} [] ys {prf1 = Refl} {prf2 = Refl} =
  replace2 {p = Alt _ _}
         (Frexify (frex 1) [start       ] (var 0 :+: sta 0 =-=           var 0))
         (Frexify (frex 1) [parity_right] (var 0           =-= sta 0 :+: var 0))
         ys

(++) {start} {parity_right}
     ((::) {parity} x xs {prf1 = Refl} {prf2 = Refl}) ys
     {prf1 = Refl}
     {prf2 = Refl}
  = (::) x ((++) xs ys
             {prf1 = Frexify (frex _)
                        [start, parity] $
                        var 0 :+: (sta 1 :+: var 1)   =-=   (sta 1 :+: var 0) :+: var 1})
        {prf2 = Frexify (frex _)
                          [parity, parity_right] $
                          (sta 1 :+: var 0) :+: var 1 =-= sta 1 :+: (var 0 :+: var 1)}
```

**(b)** commutative monoids frexlet in action

**Figure 3.** indexing modulo equations with Frex

in HOL Light [9] and Pottier's in Coq [16]) help users discharge proofs obligations involving equalities of polynomials on a commutative ring with no zero divisor.

In Idris, Slama and Brady [17, 18] implement a hierarchy of rewriting procedures for algebraic structures of increasing complexity. We follow this last approach, and additionally: (1) our procedures are complete by construction, (2) our procedures are based on normalisation-by-evaluation (like Boutin's tactic, and unlike Slama-Brady), and (3) our library is extensible, where sufficiently motivated users can extend the library with bespoke solvers, and we provide some support for them to do so.

The Meta-F★ language [11] provides normalisation tactics for commutative monoids and semi-rings through its metaprogramming facilities. The way we use Frex resembles how Meta-F★ uses these tactics. We hope to see whether Frex can (1) use the metaprogramming facilities to reduce the fording noise, and (2) can help in their verification efforts.

## Acknowledgments

Supported by a Royal Society University Research Fellowship, an Alan-Turing Institute seed funding grant, and a Facebook Research Award. We are grateful to James McKinna and the #idris channel for many conversations and much encouragement. We thank Conor McBride for extensive comments on this manuscript.

## References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In Stephanie Weirich, editor, *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, pages 13–24. ACM, 2013.

[2] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[3] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical*

*Aspects of Computer Software*, pages 515–529, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

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

[5] Pierre-Évariste Dagand. *A Cosmology of Datatypes: Reusability and Dependent Types*. PhD thesis, 2013.

[6] Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27:e9, 2017.

[7] Pierre-Évariste Dagand and Conor Thomas McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 2014.

[8] Henry Ford and Samuel Crowther. *My life and work*. William Heinemann Ltd., 1922.

[9] John Harrison. Automating elementary number-theoretic proofs using gröbner bases. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2007.

[10] Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27:e2, 2016.

[11] 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. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019.

[12] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, 1999.

[13] Conor McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Springer International Publishing, Cham, 2016.

[14] Conor Thomas McBride. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 297–309, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Conor Thomas McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[16] Loic Pottier. Connecting Gröbner bases programs with coq to do proofs in algebra, geometry and arithmetics. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *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*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[17] Franck Slama. *Automatic generation of proof terms in dependently typed programming languages*. PhD thesis, 2018.

[18] Franck Slama and Edwin Brady. Automatically proving equivalence by type-safe reflection. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, pages 40–55, Cham, 2017. Springer International Publishing.

[19] Pierre-Yves Strub. Coq modulo theory. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2010.

[20] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, page 15–24, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Thomas Williams and Didier Rémy. A principled approach to ornamentation in ml. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[22] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.