Extensible Metatheory Mechanization via Family Polymorphism

ENDE JIN, University of Waterloo, Canada
NADA AMIN, Harvard University, USA
YIZHOU ZHANG, University of Waterloo, Canada

With the growing practice of mechanizing language metatheories, it has become ever more pressing that interactive theorem provers make it easy to write reusable, extensible code and proofs. This paper presents a novel language design geared towards extensible metatheory mechanization in a proof assistant. The new design achieves reuse and extensibility via a form of family polymorphism, an object-oriented idea, that allows code and proofs to be polymorphic to their enclosing families. Our development addresses technical challenges that arise from the underlying language of a proof assistant being simultaneously functional, dependently typed, a logic, and an interactive tool. Our results include (1) a prototypical implementation of the language design as a Coq plugin, (2) a dependent type theory capturing the essence of the language mechanism and its consistency and canonicity results, and (3) case studies showing how the new expressiveness naturally addresses real programming challenges in metatheory mechanization.

CCS Concepts: • **Theory of computation** → *Type theory*; *Logic*; *Object-oriented constructs*; *Functional constructs*; • **Software and its engineering** → *Language features*; *Compilers*; *Semantics*; *Software verification*.

Additional Key Words and Phrases: Proof engineering, interactive theorem proving, expression problem, inductive types, extensible frameworks, modules, mixins, reuse, late binding, dependent type theory, Coq.

ACM Reference Format:
Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Proc. ACM Program. Lang.* 7, PLDI, Article 172 (June 2023), 25 pages. https://doi.org/10.1145/3591286

1 INTRODUCTION

It is fashionable—and commendable—for programming-languages researchers to mechanize metatheories using proof assistants. However, the practice also brings to the forefront a long-standing challenge: the *expression problem* (EP) [Wadler et al. 1998].

The EP is a programming challenge that epitomizes the difficulty of writing type-safe, extensible code. To define an expression language that can be reused for future extensions, the programmer faces a fundamental tension [Reynolds 1975] between adding new constructors to a data type (e.g., new abstract syntax) and adding new functions over the data type (e.g., new compiler passes).

The EP is well studied in the setting of ordinary functional programming and object-oriented (OO) programming. Modern languages, such as Scala [Odersky and Zenger 2005], have a good supply of linguistic features that offer expressive power to solve the EP.

In contrast, proof assistants offer few linguistic solutions to the EP. But the challenge of writing extensible, type-safe code is as real as in any other language. Metatheory mechanization epitomizes

Authors' addresses: Ende Jin, Cheriton School of Computer Science, University of Waterloo, Canada, ende.jin@uwaterloo.ca; Nada Amin, Paulson School of Engineering and Applied Sciences, Harvard University, USA, namin@seas.harvard.edu; Yizhou Zhang, Cheriton School of Computer Science, University of Waterloo, Canada, yizhou@uwaterloo.ca.

ACM 2475-1421/2023/6-ART172
https://doi.org/10.1145/3591286

the difficulty: the programmer faces a tension between adding new constructors to an inductive type (e.g., new abstract syntax) and adding new functions and theorems over the inductive type (e.g., new metatheoretical results).

In the Coq proof assistant, for instance, inductive types, as well as functions and theorems over inductive types, are closed to extension. So to reuse mechanized metatheories, the common practice is still to copy code and proofs and then modify them in each extension. But having to maintain multiple copies is highly non-modular and antithetical to good software engineering. The programmer could also turn to design patterns for structuring developments and to tool support for cutting down on boilerplate [Delaware et al. 2011, 2013a; Schwaab and Siek 2013; Keuchel and Schrijvers 2013; Forster and Stark 2020]. But these solutions tend to require heavy lifting from the programmer to make code extensible and often lead to non-idiomatic programming styles.

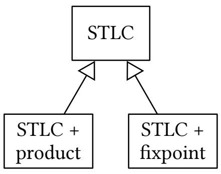
We thus set out to answer the following question: can extensible metatheory mechanization be made easier by having a proof assistant support new *linguistic features* that address the EP?

At the core of many linguistic solutions to the EP is *inheritance*. Inheritance is sometimes interpreted narrowly as a subclass' inheriting methods and instance variables from its superclass. But the language-theoretic essence of inheritance is more general: it is a linguistic approach to incrementally modifying record-like definitions by allowing *late binding* [Cook et al. 1990].

Language mechanisms including virtual classes [Madsen and Moller-Pedersen 1989; Ernst et al. 2006; Clarke et al. 2007], mixins [Bracha and Cook 1990], virtual types [Thorup 1997], and extensible cases [Blume et al. 2006] are based on this essential idea of inheritance. In particular, when a mechanism allows late-binding the meaning of nested types and terms, it is said to support *family polymorphism* [Ernst 2001]: types and terms are polymorphic to a family they are nested within.

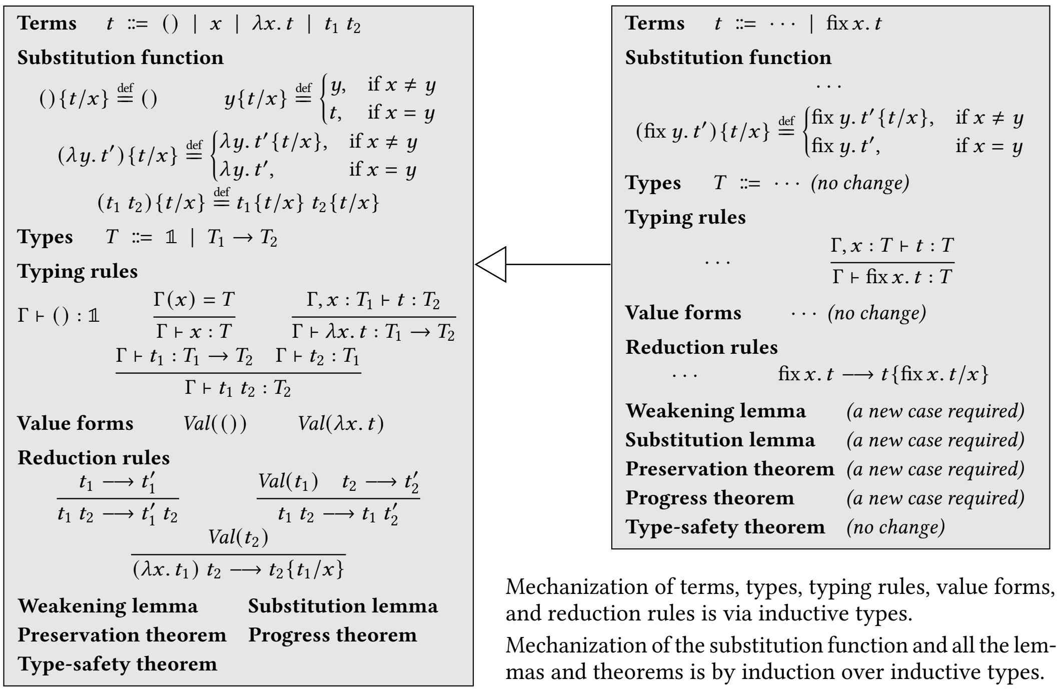
Contributions. We contribute a language design that integrates family polymorphism into a proof assistant. Because code and proofs are polymorphic to a family they are nested within, they can be inherited and reused by a derived family. Hence, family polymorphism allows for extensible metatheory mechanization.

As an example, the diagram to the right depicts an extensible mechanization of the simply typed λ-calculus (STLC), using family polymorphism. An extension of STLC with products and another with fixpoints can both inherit from the base STLC family: they reuse mechanized metatheories, from abstract syntax all the way to the type-safety theorem, only adding new constructors to inductive types and adding new cases to recursive functions and induction proofs *as needed* by an extension.



Integrating family polymorphism into a dependent type theory for logical reasoning, however, poses significant technical challenges. As we analyze, pillars of dependent type theories—including inductive types, definitional equality, and logical consistency—are all inimical to the kind of extensibility and family polymorphism found in existing OO language designs. Thus, our contributions include novel design recipes for dealing with these challenges and foundational metatheoretical guarantees on the underlying logic. Specifically, we make the following contributions.

- We present a language design that enables extensible metatheory mechanization in a higher-order, dependent type theory with inductive types (Section 3). The language design reconciles the expressiveness enabled by family polymorphism with the rigor of a proof assistant, while largely retaining an idiomatic programming style.
- We contribute a prototypical implementation of our language mechanism as a Coq plugin (Section 4). The plugin works by compiling surface-language definitions into Coq modules parameterized by explicit extensibility hooks.



Terms $t ::= () \mid x \mid \lambda x.\, t \mid t_1\ t_2$

Substitution function

$$()\{t/x\} \overset{\text{def}}{=} () \qquad y\{t/x\} \overset{\text{def}}{=} \begin{cases} y, & \text{if } x \neq y \\ t, & \text{if } x = y \end{cases}$$

$$(\lambda y.\, t')\{t/x\} \overset{\text{def}}{=} \begin{cases} \lambda y.\, t'\{t/x\}, & \text{if } x \neq y \\ \lambda y.\, t', & \text{if } x = y \end{cases}$$

$$(t_1\ t_2)\{t/x\} \overset{\text{def}}{=} t_1\{t/x\}\ t_2\{t/x\}$$

Types $T ::= \mathbb{1} \mid T_1 \to T_2$

Typing rules

$$\Gamma \vdash () : \mathbb{1} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.\, t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Value forms $Val(())$ $Val(\lambda x.\, t)$

Reduction rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{Val(t_1) \quad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{Val(t_2)}{(\lambda x.\, t_1)\ t_2 \longrightarrow t_2\{t_1/x\}}$$

Weakening lemma **Substitution lemma**
Preservation theorem **Progress theorem**
Type-safety theorem

Terms $t ::= \cdots \mid \text{fix}\, x.\, t$

Substitution function

$$\cdots$$

$$(\text{fix}\, y.\, t')\{t/x\} \overset{\text{def}}{=} \begin{cases} \text{fix}\, y.\, t'\{t/x\}, & \text{if } x \neq y \\ \text{fix}\, y.\, t', & \text{if } x = y \end{cases}$$

Types $T ::= \cdots$ *(no change)*

Typing rules

$$\cdots \qquad \frac{\Gamma, x : T \vdash t : T}{\Gamma \vdash \text{fix}\, x.\, t : T}$$

Value forms $\cdots$ *(no change)*

Reduction rules

$$\cdots \qquad \text{fix}\, x.\, t \longrightarrow t\{\text{fix}\, x.\, t/x\}$$

Weakening lemma	*(a new case required)*
Substitution lemma	*(a new case required)*
Preservation theorem	*(a new case required)*
Progress theorem	*(a new case required)*
Type-safety theorem	*(no change)*

Mechanization of terms, types, typing rules, value forms, and reduction rules is via inductive types.
Mechanization of the substitution function and all the lemmas and theorems is by induction over inductive types.

Figure 1. STLC metatheories (left) and its extension with fixpoints (right).

- We capture the essence of the new language mechanism formally by extending Martin-Löf type theory with facilities to express family polymorphism (Section 6). We prove foundational metatheoretical results including consistency and canonicity.
- We present case studies of using our Coq plugin to mechanize language metatheories (Section 7). They show how our language design naturally solves the EP and enables a good amount of reuse and extensibility for mechanizing proofs.

2 DESIGN REQUIREMENTS AND CHALLENGES

Integrating family polymorphism into a proof assistant presents challenges far beyond those found in an object-oriented setting [Nystrom et al. 2004; Aracic et al. 2006; Igarashi and Viroli 2007; Zhang and Myers 2017], as the underlying programming language of an interactive theorem prover is simultaneously functional, dependently typed, a logic, and an interactive tool.

C1. Extensible inductive types vs. exhaustive inductive reasoning. Inductive types, generalizing algebraic data types found in functional languages, are a central feature of any proof assistant in use for mechanizing language metatheories. They offer a means to define abstract syntax and inference rules. But unfortunately, inductive types are closed to extension by design.

A family-polymorphism design could potentially support extensible inductive types, by allowing a *derived family* to add new constructors to inductive types inherited from a *base family*. Such a feature would be useful for extending mechanized languages. As an example, Figure 1 shows an STLC extension, the mechanization of which would be made easier by extensible inductive types. Code would be organized into two families, with the derived family inheriting constructors from the base family (e.g., the ellipsis under "Typing rules") and adding new constructors to model the syntax and semantics of a fixpoint construct.

However, there is a tension between extensibility of inductive types and exhaustivity of inductive reasoning.[1] In Figure 1, all the lemmas and theorems, as well as the substitution function, require induction (i.e., elimination of inductive types). A language design must enforce that induction remains exhaustive in the face of the new constructors in the derived family. For modularity, the type system should do so without requiring redefinition or rechecking of those cases already handled by the base family.

C2. Late binding vs. definitional equality. Family polymorphism enables modular reuse via late binding: the code of a base family can be reused by a derived family, because fields referenced by that code have meanings polymorphic to the enclosing family.

This flexibility, however, prevents *definitional equality* that one takes for granted when programming in a proof assistant. In Figure 1, a proof assistant supporting family polymorphism cannot unfold references to the substitution function into a pattern match against four cases, as a derived family may modify the definition of the function by adding new cases. Without the ability to unfold the substitution function, how can the programmer even prove the substitution lemma? The problem is compounded by the occasional need in derived families to override fields, which is potentially at odds with being able to use equalities over the fields.

C3. Self reference vs. logical consistency. The language-theoretic essence of late binding is self reference; inheritance and family polymorphism are mechanisms for incrementally modifying self-referential definitions [Cook et al. 1990]. However, self reference could easily lead to divergence. Divergence is not a concern for the design of ordinary OO or functional languages, but it would mean logical inconsistency—and hence unsoundness!—for a language aimed at logical reasoning. A family-polymorphism design must tame self reference to guarantee consistency.

C4. User experience and system implementation. Interactive theorem proving and tactic programming are central to a typical programming experience with a proof assistant. A language design integrating family polymorphism should be compatible with these forms of programming. In particular, it should be possible in our system to incrementally navigate through vernacular commands and, moreover, construct proofs with common tactics while getting instant feedback on the proof state, even in the middle of a family definition. Last but not least, in addition to proving theorems, it should be possible for terms defined with families to possess computational content.

3 LANGUAGE DESIGN

We present the key ingredients of our design as an extension to the Coq proof assistant, though we believe the design could be adapted to other proof assistants such as Lean. We call our design and implementation FPOP (family polymorphism for a proof assistant). In this section, we focus on the language design of FPOP. Section 4 describes its implementation as a Coq plugin.

Figure 2 shows how STLC and its extension with fixpoints can be mechanized using FPOP, in a style envisioned in Figure 1. The base family `STLC` hosts the STLC metatheories, from abstract syntax to the type-safety theorem. Family `STLCFix`, derived from `STLC`, makes adjustments as needed by a fixpoints extension: it adds new constructors to the inductive types (`FInductive`) and adds new cases to the recursive functions (`FRecursion`) and induction proofs (`FInduction`). Existing constructors and cases, as well as those definitions and theorems that need no adjustments (`ty`, `env`, `empty`, `steps`, and `typesafe`), are automatically inherited and reused. In particular, executing the last command, `Check STLCFix.typesafe`, displays the type-safety theorem of the fixpoints extension.

[1]The tension reflects a duality between variants and records. With record-like language constructs (e.g., objects and families), an extension can safely add new fields: existing fields can still be projected. But variant-like constructs (e.g., inductive types) do not automatically enjoy safe, modular addition of constructors: existing pattern matches could become non-exhaustive.

```
Family STLC.                          (* The base STLC *)

FInductive tm : Set ≔                      (* Terms *)
| tm_unit : tm
| tm_var  : id → tm
| tm_abs  : id → tm → tm
| tm_app  : tm → tm → tm.

FRecursion subst on tm   (* Substitution function *)
  motive λ(_ : tm), id → tm → tm.
Case tm_unit ≔ λ x t, tm_unit.
Case tm_var ≔ λ y x t,
  if eqb x y then t else (tm_var y).
Case tm_abs ≔ λ y t' IHt' x t,
  tm_abs y (if eqb x y then t' else IHt' x t).
Case tm_app ≔ λ t1 IHt1 t2 IHt2 x t,
  tm_app (IHt1 x t) (IHt2 x t).
End subst.

FInductive ty : Set ≔                      (* Types *)
| ty_unit  : ty
| ty_arrow : ty → ty → ty.

FDefinition env : Type ≔ id → option ty.
FDefinition empty : env ≔ λ _, None.

FInductive hasty : env → tm → ty → Prop ≔
                                     (* Typing rules *)
| ht_unit : ∀ G, hasty G tm_unit ty_unit
| ht_var  : …
| ht_abs  : …
| ht_app  : ….

FInductive value : tm → Prop ≔(* Value forms *)
| v_unit : value tm_unit
| v_abs  : ∀ x t, value (tm_abs x t).

FInductive step : tm → tm → Prop ≔
| st_app1 : …                     (* Reduction rules *)
| st_app2 : …
| st_beta : ∀ x t v, value v →
  step (tm_app (tm_abs x t) v) (subst t x v).

FDefinition steps ≔ clos_refl_trans step.

FInduction weakenlem        (* Weakening lemma *)
  on hasty motive λ G t T (_ : hasty G t T),
  ∀ G', includedin G G' → hasty G' t T.
Case ht_unit. … Qed.        Case ht_var. … Qed.
Case ht_abs.  … Qed.        Case ht_app. … Qed.
End weakenlem.

FInduction substlem       (* Substitution lemma *)
  on hasty motive λ G' t T (_ : hasty G' t T),
  ∀ G x t' T', G' = extend G x T' →
  hasty empty t' T' →
  hasty G (subst t x t') T.
Case ht_unit. … Qed.        Case ht_var. … Qed.
Case ht_abs.  … Qed.        Case ht_app. … Qed.
End substlem.
```

```
FInduction preserve       (* Preservation theorem *)
  on hasty motive λ G t T (_ : hasty G t T),
  G = empty → ∀ t', step t t' →
  hasty empty t' T.
Case ht_unit. … Qed.        Case ht_var. … Qed.
Case ht_abs.  … Qed.        Case ht_app. … Qed.
End preserve.

FInduction progress           (* Progress theorem *)
  on hasty motive λ G t T (_ : hasty G t T),
  G = empty → value t ∨ ∃ t', step t t'.
Case ht_unit. … Qed.        Case ht_var. … Qed.
Case ht_abs.  … Qed.        Case ht_app. … Qed.
End progress.

FTheorem typesafe :        (* Type-safety theorem *)
  ∀ t t' T, steps t t' → hasty empty t T →
  value t' ∨ ∃ t'', step t' t''.
Proof. … Qed.

End STLC.
```

```
Family STLCFix extends STLC.
                        (* STLC extended with fixpoints *)
FInductive tm : Set +=
| tm_fix : id → tm → tm.

FRecursion subst on tm motive λ_, id→tm→tm.
Case tm_fix ≔ λ y t' IHt' x t, ….
End subst.

FInductive hasty : env → tm → ty → Prop +=
| ht_fix : ∀ G x t T,
  hasty (extend G x T) t T →
  hasty G (tm_fix x t) T.

FInductive step : tm → tm → Prop +=
| st_fix : ∀ x t,
  step (tm_fix x t) (subst t x (tm_fix x t)).

FInduction weakenlem on hasty motive ….
Case ht_fix. … Qed.
End weakenlem.

FInduction substlem on hasty motive ….
Case ht_fix. … Qed.
End substlem.

FInduction preserve on hasty motive ….
Case ht_fix. … Qed.
End preserve.

FInduction progress on hasty motive ….
Case ht_fix. … Qed.
End progress.

End STLCFix.
```

```
Check STLCFix.typesafe.
```

Figure 2. Using FPOP to mechanize STLC and the fixpoints extension, as envisioned in Figure 1.

3.1 Extensible Inductive Types and Exhaustive Recursion/Induction

Extending inductive types. In family `STLCFix`, the `FInductive tm` *further binds* the `tm` type in family `STLC`. It has five constructors: four inherited from `STLC` and a fifth called `tm_fix`.

Crucially, the meanings of `tm` and its constructors are *late bound*, depending on the family in which they are referenced. Consider `tm_app`. It is defined in family `STLC` and is thus unaware of `tm_fix`. Yet in family `STLCFix`, we can use `tm_app` to construct applications of fixpoints, as in `tm_app (tm_fix "f"` t_1`)` t_2. This use is justified by `tm_app`'s type, `tm → tm → tm`. It allows `tm_app` to be applied to anything of type `tm`, which in family `STLCFix` include those constructed from `tm_fix`.

Ensuring exhaustivity of induction. Ordinarily, an inductive type is not extensible: it is exhaustively generated by its constructors and has no more inhabitants beyond those they construct. This idea is captured by the eliminator (aka *recursor*) associated with an inductive type. For example, if `tm` were defined as an ordinary inductive type, then its eliminator would have the following type:

```
tm_rect : ∀ (P : tm → Type), P tm_unit → (∀ x, P (tm_var x)) →
                (∀ x t, P t → P (tm_abs x t)) →
                (∀ t1, P t1 → ∀ t2, P t2 → P (tm_app t1 t2)) → ∀ t, P t
```

The eliminator would enable function definitions by recursion and proofs by induction, over `tm`, that exhaustively handle the four cases corresponding to each constructor. The (dependent) return type `P` is called the *motive* of the recursion.

With inductive types made extensible, exhaustivity of induction is now in question, however. In particular, the recursor `tm_rect` should no longer be allowed, because its type mentions `tm`, whose meaning is late bound, yet `tm_rect` purports to claim `∀ t, P t` given only four case handlers.

To reconcile the tension without requiring redefinition or rechecking of case handlers (C1), our design introduces the `FRecursion` and `FInduction` commands. The key idea is to allow case handlers to be added retroactively, should inductive types be extended, and to allow recursion and induction (which are defined in terms of case handlers) to be late bound.

As an example, consider the substitution function `subst`, defined using `FRecursion`. The `on` clause specifies that recursion is over `tm`. The `motive` clause suggests that the recursive function being defined has type `tm → id → tm → tm`. The `subst` function in `STLC` is further bound by the `subst` in `STLCFix`: the four cases from `STLC` are automatically inherited and reused, with `STLCFix` adding a fifth case retroactively to form a new `subst` function. For exhaustivity, it is a static error if the programmer fails to further bind `subst` and define this fifth case. The type system does this check in family `STLCFix` by examining if the inductive type `tm`, over which `subst` is recursively defined, is further bound in the same family.

The `FInduction` command is similar to `FRecursion` but allows cases to be defined in proof mode. Consider `weakenlem` as an example. It is proven by induction on the typing relation `hasty`. Its `motive` clause shows that the lemma reads as `∀ G t T, hasty G t T → ∀ G', includedin G G' → hasty G' t T`. Upon entering case `ht_abs`, for instance, Coq enters proof mode with the goal

```
∀ G', includedin G G' → hasty G' (tm_abs x t) (ty_arrow T1 T2)
```

and with the induction hypothesis `∀ G', includedin (extend G x T1) G' → hasty G' t T2`. The programmer can use tactic programming to discharge the goal. Because the lemma is by `FInduction` on `hasty` and because `STLCFix` adds a constructor `ht_fix` to `hasty`, the programmer is required in family `STLCFix` to extend the proof of `weakenlem` to handle this extra case.

3.2 Late Binding and Equalities

Late binding of nested names. OO inheritance allows the late binding of method names. Family polymorphism generalizes the power of OO inheritance by allowing the late binding of all names nested within families, including those referring to types. We have seen that late binding of `tm` allows the `tm` constructors in `STLC` to be reused in `STLCFix` to construct terms containing `tm_fix`.

As another example, consider the type of `st_beta` in family `STLC`. It refers to `subst`, whose meaning is late bound. When `st_beta` is inherited into family `STLCFix`, `st_beta` has a type that now refers to the `subst` function in `STLCFix`, where `subst` is defined by handling all the five cases known to that family. Thus, late binding of `subst` allows the derived family to reuse `st_beta` as the β-reduction rule for applications possibly constructed from `tm_fix`.

Importantly, a name is late bound only within a family that defines or further binds it. Outside such families, the name can be accessed only by explicitly specifying a family that contains it. For example, the last line of Figure 2 accesses `typesafe` with a qualifier `STLCFix`. The command prints

```
STLCFix.typesafe: ∀ t t' T, STLCFix.steps t t' → STLCFix.hasty STLCFix.empty t T →
                                STLCFix.value t' ∨ ∃ t'', STLCFix.step t' t''.
```

In this type, all references to nested names are qualified by `STLCFix`, as desired.

Equality on late bound names. Consider proving the `ht_unit` case of the substitution lemma, `substlem`. The goal is seemingly trivial: the programmer is asked to prove

```
∀ G' G x t' T', G' = extend G x T' → hasty empty t' T' → hasty G (subst tm_unit x t') ty_unit.
```

If `subst` were an ordinary Coq function, then the programmer could discharge the goal with `intros; simpl subst; apply ht_unit`. The Ltac term `simpl subst` unfolds `subst` using its definition and simplifies `subst tm_unit x t'` to `tm_unit`.

But with `subst` being late bound, `subst` cannot and should not be unfolded (C2): the definition of `subst`, as a recursive function, varies across families, yet a derived family should be able to reuse the proof of the `ht_unit` case even when it has to modify the definition of `subst`. Without the ability to unfold `subst`, how can the programmer make progress in this proof, then?

A key observation is that although late binding prevents *definitional equality* on `subst`, it does not affect *propositional equality*. That is, `∀ x t, subst tm_unit x t = tm_unit`, as a proposition (`Prop`) about the computational behavior of `subst`, should still hold. After all, how `subst` is defined on `tm_unit` does not vary from a base family to a derived family; what can vary is `subst` itself as a recursive function combining the case handlers.

Based on this insight, FPOP automatically generates a propositional equality for each case handler defined within `FRecursion`, making the equalities and the recursive function available as axioms for use by the rest of the current family. FPOP also provides a tactic `fsimpl` that enables, for instance, simplifying `subst tm_unit x t'` to `tm_unit`. `fsimpl` works by rewriting applications of the axiomatized recursive function using the axiomatized equalities about its computational behaviors.

Note that the definitional equality on `subst` is available *outside* those families that contain `subst`. Within those families, the meaning of `subst` is late bound (i.e., polymorphic to the enclosing family), so only propositional equality is available. In contrast, outside those families, `subst` is always referenced by specifying a family that contains it, as in `STLC.subst` and `STLCFix.subst`. As far as the type checker is concerned, `tm_unit` and `STLCFix.subst tm_unit x t'` are the same thing—the type checker equates them definitionally by unfolding `STLCFix.subst` and performing normalization.

3.3 Overriding

In an OO language, a subclass can override methods of a superclass. Similar expressivity is useful for mechanizing proofs, too. For example, in a derived family, rather than adding new cases to an induction proof, the programmer may prefer overriding the proof entirely, as we observe in our case studies (Section 7).

Overriding is potentially incompatible with having equalities on late bound names, however. Coq distinguishes *opaque* definitions from *transparent* ones. FPOP supports the overriding of opaque

```
Family STLCIsorec extends STLC.
              (* STLC extended with iso-recursive types *)
FInductive tm : Set +=
| tm_fold : tm → tm | tm_unfold : tm → tm.

FRecursion subst on tm motive .... ... End subst.

FInductive ty : Set +=
| ty_var : id → ty | ty_rec : id → ty → ty.

FRecursion tysubst on ty(* Type-level substitution *)
  motive λ(_ : ty), id → ty → ty.
Case ty_unit  ≔ ....   Case ty_arrow ≔ ....
Case ty_var   ≔ ....   Case ty_rec   ≔ ....
End tysubst.

FInductive hasty : env → tm → ty → Prop +=
| ht_fold : ∀ G t α T,
    hasty G t (tysubst T α (ty_rec α T)) →
    hasty G (tm_fold t) (ty_rec α T)
| ht_unfold : ....
...                              (* Other adjustments *)
End STLCIsorec.
```

```
Family STLCFixIsorec extends STLC
using STLCFix, STLCIsorec.         (* STLC extended
                    with fixpoints and iso-recursive types *)
End STLCFixIsorec.
```

```
Family STLCProd extends STLC.
                         (* STLC extended with products *)
FInductive tm : Set +=
| tm_pair : tm → tm → tm
| tm_fst : tm → tm | tm_snd : tm → tm.

FRecursion subst on tm motive .... ... End subst.

FInductive ty : Set +=
| ty_prod : ty → ty → ty.

...                              (* Other adjustments *)

End STLCProd.
```

```
Family STLCProdIsorec extends STLC
using STLCProd, STLCIsorec.      (* STLC extended
                   with products and iso-recursive types *)
FRecursion tysubst on ty
  motive λ(_ : ty), id → ty → ty.
Case ty_prod ≔ ....(* Substitution on product types *)
End tysubst.

End STLCProdIsorec.
```

```
Family STLCFixProdIsorec extends STLC
using STLCFix, STLCProdIsorec.  (* STLC extended
         with fixpoints, products, and iso-recursive types *)
End STLCFixProdIsorec.
```

Figure 3. Composing extensions of STLC.

definitions, which include most proofs. It is safe to override opaque definitions, because the type checker will never attempt to unfold them. For transparent definitions, the common case is that the programmer does not want to override them. In Figure 2, `env`, `empty`, `subst`, and `steps` are transparent—that is, they are not defined with `Qed`. FPOP treats transparent definitions as non-overridable by default. Thus, the definitional equalities on `env`, `empty`, and `steps`, as well as the propositional equality on `subst`, are available to the type checker for type-checking the families.

FPOP does allow the overriding of transparent definitions explicitly marked as `Overridable` by the programmer. Overriding is made safe by requiring that when an overridable field is overridden, code whose type checking involves unfolding that field should be overridden too. We expect this feature to be used occasionally.

3.4 Sound Logical Reasoning

In Figure 2, just because the `typesafe` theorem in `STLC` is inherited and reused by `STLCFix`, it does not follow that in `STLCFix` the programmer can use `typesafe` to prove `progress`. If the programmer did, then they would be committing the logical fallacy of circular reasoning: the proof of `progress` would depend on `typesafe`, yet the proof of `typesafe` depends on `progress`. Such circularity would easily lead to logical inconsistency. Consider the following two families, where `B` extends `A`:

```
Family A.
FLemma f : False. Proof. Admitted.
FLemma g : False. Proof. apply f. Qed.
End A.
```

```
Family B extends A.
FLemma f : False. Proof. apply g. Qed.
End B.
```

`B` overrides lemma `f` by proving it using `g`. Lemma `g` is in turn inherited from family `A`, where it is proven using a late bound reference to lemma `f`. Circularity between `f` and `g` allows proving `False`!

To ensure the soundness of logical reasoning (C3), the type system requires that in a derived family, the context in which a field is defined be preserved from the base family. In `STLC`, `progress` is in the context of typesafe. Per the requirement, this relationship must be preserved into STLCFix, which prevents the proof of progress from depending on typesafe in STLCFix.

Note that the requirement still allows a derived family to introduce new declarations into the context of a field. For example, the left column of Figure 3 shows an extension of STLC with iso-recursive types, where tysubst is introduced into the context of hasty. In the event that FPOP cannot infer where the programmer intends to place a new field, an annotation is required.

3.5 Composing Families as Mixins

Families can be readily reused to construct larger extensions that *mix in* [Bracha and Cook 1990] the functionalities of the individual families. A family like STLCFix can be viewed as a family-to-family functor—and hence a mixin, in the sense of Flatt et al. [1998]—that transforms any family providing the base STLC functionalities (i.e., STLC or a derived family there of) into a new family additionally supporting fixpoints.

In Figure 3, STLCFixIsorec is declared as an STLC extension that mixes in STLCFix and STLCIsorec. The family is declared with minimal verbiage, yet STLCFixIsorec.typesafe is automatically a proof of the type-safety theorem of an STLC equipped with fixpoints and iso-recursive types.

Mixin composition is a form of multiple inheritance, which may cause name conflicts in general. FPOP requires the programmer to resolve conflicts by overriding conflicted overridable fields.

In the presence of extensible inductive types, mixin composition may also create an obligation to retrofit the mixins with new case handlers. In Figure 3, family STLCProdIsorec is composed of two mixins: STLCProd, which extends ty with a new constructor ty_prod, and STLCIsorec, which introduces a function tysubst recursively defined on ty. Hence, for exhaustivity, it is required that a composition of STLCProd and STLCIsorec should additionally handle the ty_prod case in tysubst.

3.6 Injectivity and Disjointness of Constructors via Partial Recursors

Tactics support for constructors. Coq provides tactics for proving injectivity and disjointness of constructors (i.e., injection and discriminate). The proof terms generated by the tactics involve exhaustively matching on the constructors of an inductive type, so they do not work for extensible inductive types (C1). In principle, the programmer could use FInduction to prove injectivity and disjointness. But this workaround is unsatisfying: it is tedious, it forces the programmer to revisit the induction proofs every time an inductive type is extended, and above all, why should a property like ¬(tm_var "x" = tm_abs "y" t) have anything to do with tm_fix?

To provide a streamlined programming experience (C4), FPOP offers two tactics, finjection and fdiscriminate. For example, in a proof state that contains a manifestly false assumption H : tm_var "x" = tm_abs "y" t, the programmer can use fdiscriminate H to obtain False and thus discharge the current goal, just as they would with discriminate H if tm were not extensible.

Partial recursors. We make the observation that injectivity and disjointness of existing constructors ought to hold regardless of future addition of constructors. This insight motivates the design of *partial recursors*, which power the finjection and fdiscriminate tactics. Partial recursors can be generated for inductive types defined with FInductive.

As analyzed earlier, ordinary recursors, such as tm_rect, are impossible within a family in which the name of the inductive type is late bound. However, a key observation is that extensible inductive types still admit a weakened elimination principle where the motive is an option type. For example, within family STLC, the partial recursor for tm has the following type:

```
tm_prect_STLC : ∀ (P : tm → Type), option (P tm_unit) → (∀ x, option (P (tm_var x))) →
  (∀ x t, option (P t) → option (P (tm_abs x t))) →
  (∀ t1, option (P t1) → ∀ t2, option (P t2) → option (P (tm_app t1 t2))) → ∀ t, option (P t)
```

As expected, `tm_prect_STLC` is axiomatized along with four equalities describing its computational behaviors, one for each constructor. For instance, the equality for constructor `tm_abs` is as follows:

```
tm_abs_eq_STLC: ∀ x t P H1 H2 H3 H4,
                tm_prect_STLC P H1 H2 H3 H4 (tm_abs x t) = H3 x t (tm_prect_STLC P H1 H2 H3 H4 t)
```

Importantly, unlike the standard `tm_rect` recursor, the partial recursor `tm_prect_STLC` is compatible with the late binding of `tm` in its type. When `tm_prect_STLC` is inherited into family `STLCFix`, all the previous four equalities still hold, and a trivial, fifth equality is made available:

```
tm_fix_eq_STLC : ∀ x t P H1 H2 H3 H4, tm_prect_STLC P H1 H2 H3 H4 (tm_fix x t) = None
```

Partial recursors appear weaker than ordinary recursors, but there is power in restraint. In particular, they offer a principled, uniform way to derive injectivity and disjointness of constructors, while supporting future extension: they enable injective mappings from a late bound inductive type, like `tm`, to an ordinary inductive type, like ℕ, the injectivity and disjointness of whose constructors are readily available. For example, in a proof state with the assumption `H : tm_var "x" = tm_abs "y" t`, running `fdiscriminate H` first applies an injective mapping to both sides of `H`, obtaining

```
tm_prect_STLC (λ_, ℕ) (λ_, Some 1) (Some 2) (λ___, Some 3) (λ____, Some 4) (tm_var "x") =
tm_prect_STLC (λ_, ℕ) (λ_, Some 1) (Some 2) (λ___, Some 3) (λ____, Some 4) (tm_abs "y" t)
```

and then rewrites the above equality using the axiomatized computational behaviors of `tm_prect_STLC`, obtaining `Some 1 = Some 3`, from which `False` easily follows. Note that the proof term generated by `fdiscriminate H` in `STLC` is reusable by family `STLCFix`, because the partial recursor and its computational behaviors are compatible with the late binding of `tm`.

Within family `STLCFix`, a second partial recursor (called `tm_prect_STLCFix`) and its computational behaviors are automatically axiomatized, which allows properties of `tm_fix`, such as `tm_fix x t1 = tm_fix x t2 → t1 = t2`, to be proved.

4 COMPILING FAMILY POLYMORPHISM TO PARAMETERIZED MODULES

We implement our language design as a Coq plugin. It works by translating programs in FPOP syntax into programs that can be checked and evaluated by Coq. The translation is compatible with interactive theorem proving (C4), in that a family is translated piece by piece, allowing each field to be defined and checked separately. The translation is modular and efficient, in that code compiled for fields of a base family can be shared with derived families without having to be rechecked.

Explicit self parameterization. The spirit of the translation is to take "family polymorphism" literally: every field is translated into a Coq definition that is polymorphic to (i.e., universally quantified over) a representation of its enclosing family. While this universal quantification has been implicit with the FPOP syntax, it has to be made explicit in the translated Coq code.

Figures 4 and 5 illustrate the translation of the `STLC` and `STLCFix` families from Figure 2. Fields of a family are translated into parameterized Coq modules (or parameterized module types).

Take `env` in family `STLC` for example. It is translated into a top-level module named `STLC°env`. This module has a `self` parameter representing the enclosing family: fields of the current family in the context of `env` can be referenced through `self`. In particular, `env` is defined as `id → option ty`, where `ty` is a late-bound reference to the `ty` field of the enclosing family. Hence, this reference to `ty` is translated to `self.ty`, which is manifestly polymorphic to the enclosing family. This translation of the `env` field can be shared with a derived family even if it extends `ty` (e.g., `STLCProd`)—no recompilation is needed because `self.ty` is not tied to any concrete definition of `ty`.

The type of `STLC°env`'s `self` parameter is `STLC°env°Ctx`, a module type constructed from `STLC°ty` (i.e., the translation of the field before `env`) and its context `STLC°ty°Ctx`. In turn, `STLC°ty°Ctx` (not shown in Figure 4) is constructed from `STLC°subst`, the translation of the field before `ty`, and its

```
(* Code emitted upon definition of tm in family STLC *)
Module Type STLC°tm°Ctx.
End STLC°tm°Ctx.

Module Type STLC°tm (self : STLC°tm°Ctx).
 Axiom tm : Set.
 Axiom tm_unit : tm.     Axiom tm_var : id → tm.
 Axiom tm_abs : id → tm → tm.
 Axiom tm_app : tm → tm → tm.
 Axiom tm_prect_STLC : ….
 Axiom tm_unit_eq_STLC : …. Axiom tm_abs_eq_STLC : ….
 Axiom tm_abs_eq_STLC : ….  Axiom tm_app_eq_STLC : ….
End STLC°tm.

(* Code emitted for definition of subst in family STLC *)
Module Type STLC°subst°Cases°Ctx.
 Include STLC°tm°Ctx.  Include STLC°tm.
End STLC°subst°Cases°Ctx.

Module STLC°subst°Cases (self : STLC°subst°Cases°Ctx).
 Def subst°tm_unit ≔ (* emitted upon definition of case *)
   λ (x : id) (t : self.tm), self.tm_unit.
 Def subst°tm_var ≔ ….
 Def subst°tm_abs ≔ ….  Def subst°tm_app ≔ ….
End STLC°subst°Cases.

Module Type STLC°subst°Ctx.
 Include STLC°subst°Cases°Ctx.
 Include STLC°subst°Cases.
End STLC°subst°Ctx.

Module Type STLC°subst (self : STLC°subst°Ctx).
 Axiom subst : self.tm → id → self.tm → self.tm.
 Axiom subst_tm_unit_eq :
   ∀ x t, self.subst (self.tm_unit) x t =
   self.subst°tm_unit x t.
 Axiom subst_tm_var_eq : ….
 Axiom …
End STLC°subst.

(* Code emitted upon definition of ty in family STLC *)  …
```

```
(* Code emitted upon definition of env in family STLC *)
Module Type STLC°env°Ctx.
 Include STLC°ty°Ctx.  Include STLC°ty.
End STLC°env°Ctx.

Module STLC°env (self : STLC°env°Ctx).
 Def env : Type ≔ id → option self.ty.
End STLC°env.

(* Code emitted for other fields defined in family STLC *)
 …

(* Code emitted upon conclusion of family STLC *)
Module STLC.

 (* Instantiate tm & its constructors *)
 Inductive tm : Set ≔
 | tm_unit : tm | tm_var : id → tm
 | tm_abs : id → tm → tm
 | tm_app : tm → tm → tm.

 (* Instantiate tm partial recursor & its comp. behaviors *)
 Def tm_prect_STLC ≔
   λ P, tm_rect (λ t, option (P t)).
 Fact tm_unit_eq_STLC : …. reflexivity. Qed.
 Fact …

 Include STLC°subst°Cases.
 (* Instantiate subst & its computational behaviors *)
 Def subst ≔ tm_rect _ subst°tm_unit
   subst°tm_var subst°tm_abs subst°tm_app.
 Fact subst_tm_unit_eq : …. reflexivity. Qed.
 Fact …

 (* Instantiate ty, its constructors, partial recusor, etc. *)  …

 (* Include env *)
 Include STLC°env.

 (* Include/Instantiate other fields of family STLC *)  …

End STLC.
```

Figure 4. Compilation of family STLC (Figure 2).

context `STLC°subst°Ctx`. Thus, the `self` parameter can be used to reference those and only those fields in the current field's typing context, which echoes the discussion in Section 3.4.

Translating extensible inductive types. An `FInductive` definition is translated to a parameterized module type. Consider the inductive type `tm`. In Figure 4, it is translated to a top-level module type `STLC°tm` that declares a `tm` type, four functions standing for the constructors, a partial recursor (`tm_prect_STLC`), and the computational behaviors of the partial recursor (e.g., `tm_abs_eq_STLC`).

Importantly, `STLC°tm` merely declares the existence of these names and their types; it does not specify their definitions. Having these names and their types available through the context parameters (`self`) suffices for the translations of the subsequent fields to be type-checked by Coq. Leaving the definitions unspecified enables `STLC` and `STLCFix` to instantiate `tm` differently upon `End STLC` and upon `End STLCFix`. In particular, non-exhaustive pattern matching is prevented because an ordinary recursor like `tm_rect` is not available.

The command `FInductive tm : Set += tm_fix : …` in family `STLCFix` is again translated to a module type `STLCFix°tm` (Figure 5). It includes all the names declared by `STLC°tm` via command `Include STLC°tm(self)`, and additionally declares `tm_fix`, a partial recursor, and related equalities.

```
(* Code emitted upon definition of tm in STLCFix *)
Module Type STLCFix°tm°Ctx.
End STLCFix°tm°Ctx.

Module Type STLCFix°tm (self : STLCFix°tm°Ctx).
 Include STLC°tm(self).
 Axiom tm_fix : id → tm → tm.
 Axiom tm_fix_eq_STLC : ∀ …, tm_prect_STLC … = None.
 Axiom tm_prect_STLCFix : ….
 Axiom tm_fix_eq_STLCFix : ….
End STLCFix°tm.

(* Code emitted upon definition of subst in STLCFix *)
Module Type STLCFix°subst°Cases°Ctx.
 Include STLCFix°tm°Ctx. Include STLCFix°tm.
End STLCFix°subst°Cases°Ctx.

Module STLCFix°subst°Cases
(self : STLCFix°subst°Cases°Ctx).
 Include STLC°subst°Cases(self).          (* reuse *)
 Def subst°tm_fix ≔ ….    (* translation of new case *)
End STLCFix°subst°Cases.

Module Type STLCFix°subst°Ctx.
 Include STLCFix°subst°Cases°Ctx.
 Include STLCFix°subst°Cases.
End STLCFix°subst°Ctx.

Module Type STLCFix°subst (self : STLCFix°subst°Ctx).
 Include STLC°subst(self).
 Axiom subst_tm_fix_eq : ….
End STLCFix°subst.

(* Code emitted for other fields defined in STLCFix *) …
```

```
(* Code emitted upon conclusion of STLCFix *)
Module STLCFix.

 (* Instantiate tm & its constructors *)
 Inductive tm : Set ≔
 | tm_unit : tm | tm_var : id → tm
 | tm_abs : id → tm → tm
 | tm_app : tm → tm → tm
 | tm_fix : id → tm → tm.

 (* Instantiate tm partial recursors & their comp. behaviors *)
 …

 Include STLCFix°subst°Cases.
 (* Instantiate subst & its computational behaviors *)
 Def subst ≔ tm_rect _ subst°tm_unit
 subst°tm_var subst°tm_abs subst°tm_app  (* reuse *)
 subst°tm_fix.
 Fact subst_tm_unit_eq : …. reflexivity. Qed.
 Fact …

 (* Include ty, its constructors, partial recursor, etc. *) …

 Include STLC°env.                        (* reuse *)

 (* Include/Instantiate other fields of STLCFix *) …

 Include STLC°typesafe.                   (* reuse *)

End STLCFix.

(* Code emitted upon command Check STLCFix.typesafe *)
Check STLCFix.typesafe.
```

Figure 5. Compilation of family STLCFix and the final Check command (Figure 2).

Translating recursion and induction. An FRecursion definition is translated in two parts: first a module containing the definitions of all the case handlers, and then a module type declaring the existence of the recursive function as well as its computational behaviors.

Consider the translation of subst in family STLC. First, a module named STLC°subst°Cases is generated on the fly. Importantly, programming remains interactive, as the programmer need not wait until the entire FRecursion definition is completed to have a Case command type-checked.

Upon End subst, a module type named STLC°subst is generated. As discussed in Section 3.2, subst can be further bound, so its definition is not exposed to the fields that come after it. Accordingly, the translation STLC°subst merely declares the types of subst and the equalities about its computational behaviors, leaving subst undefined and the equalities unproven. The equalities are stated in terms of the case handlers, whose definitions *are* available through the self parameter. So Coq can simplify, for example, the type of subst_tm_unit_eq to ∀ x t, self.subst self.tm_unit x t = self.tm_unit. These equalities about the computational behaviors of subst will be included and available for use in the translations of the subsequent fields through their self parameters.

Importantly, code generated for the case handlers is shared with derived families. In Figure 5, module STLCFix°subst°Cases reuses—without rechecking—all the case handlers in STLC°subst°Cases via command Include STLC°subst°Cases(self).

The translation of FInduction is similar, except that there is no need to register computational behaviors, as FInduction proofs are considered opaque.

Translation of further-bindables vs. non-further-bindables. In family STLC, field env and the case handlers for subst are not further-bindable by derived families. In contrast, tm, subst, and the related equalities can be further bound. The distinction is reflected in the translations. The further-bindable fields are translated to module types that export only types of the fields. The non-further-bindable fields are translated to modules that export definitional equalities on the fields. Opaque fields in FPOP can be further bound (Section 3.3); they are translated to Coq modules that export opaque fields.

Eliminating `self` by aggregation. Upon the conclusion of a family definition, a representation of the family is created. For example, module `STLC` in Figure 4 is generated upon `End STLC`. This module can be viewed as the "fixed point" of the `self`-parameterized translations. The "fixed point" is taken step by step, by adding the translation of each field to this module in the same order as they appear in the family definition.

For the non-further-bindables, the translated modules are directly included (e.g., `Include STLC°env` and `Include STLC°subst°Cases` in Figure 4). The instantiation of `self`s for these modules is implicit, thanks to a Coq nicety: when including a higher-order module, Coq automatically instantiates its parameter with the current interactive module environment. For instance, command `Include STLC°subst°Cases` is successfully executed, because Coq automatically instantiates the `self` parameter using the current module environment, which by construction contains all the fields required by `STLC°subst°Cases°Ctx`.

For the further-bindables, `Axiom`s declared in the module types must be instantiated.

- In Figure 4, an inductive type `tm` is generated, instantiating the axiomatized `tm` type and its constructors. The partial recursor `tm_prect_STLC` is defined with the help of `tm_rect`, the recursor Coq generates for `tm`. The computational behaviors of `tm_prect_STLC` are immediate, by `reflexivity`.
- Similarly, `subst` is instantiated by applying the recursor `tm_rect` to the (already included) case handlers. The computational behaviors of `subst` are then immediate, by `reflexivity`.

Module `STLCFix` in Figure 5 is emitted upon `End STLCFix`, in the same way as described above for `STLC`. The translation makes sharing evident. In particular, case handlers compiled for `STLC` are reused to instantiate `subst`, `substlem`, and alike. `STLC°env` and `STLC°typesafe` are also reused in the construction of module `STLCFix`. One may argue that since the first four constructors of `tm` are repeated in `STLCFix`, the translation does not satisfy the modular compilation requirement. We could address this concern by using wrapper types, but we consider restating constructors a small price to pay in return for the clarity and concision of implementation. We emphasize that compiled case handlers, such as `subst°tm_abs`, are entirely reusable without rechecking, even with restated constructors. Finally, the reference `STLCFix.typesafe` (where `STLCFix` is a family) can simply be translated to `STLCFix.typesafe` (where `STLCFix` is a Coq module), as the last line of Figure 5 shows.

Trusted base. Rather than modifying the Coq kernel to extend its core theory, a translation to Coq conveniently reduces the trusted base of any development using FPOP to Coq. In particular, once a family is closed, `Print Assumptions` can be used to verify that a theorem does not depend on any lingering assumptions possibly introduced by the translation. The ramifications of possible bugs in the FPOP implementation are limited to the usability of the plugin.

5 LIMITATIONS

The FPOP implementation currently does not yet bring extensibility to Coq's full facility for inductive types. Mutually inductive types and parameterized inductive types are not yet extensible, though indexed inductive types are supported and can be used to encode parameterized ones. Partial recursors can be generated only for non-indexed inductive types. These features are not exercised by our case studies (Section 7) but may be useful for modeling other languages. We believe that they do not pose conceptual challenges and can be addressed with more engineering effort on the same level as the current FPOP implementation.

What seems to require more thought from a language-design perspective is the restriction that recursion and induction (on extensible types) cannot be nested. A possible solution is to make the plugin generate proof obligations for nested pattern matches when the inductive types acquire new constructors. Also interesting for future research is the support for automatically converting terms to propositionally equally typed forms using generated propositional equalities. The lack of this automation currently may cause inconveniences for developments using intrinsically typed syntax. Finally, work on nested inheritance [Nystrom et al. 2004; Zhang and Myers 2017] points to a direction to further increase the expressive power of our language design.

6 FMLTT: A CORE DEPENDENT TYPE THEORY

We contribute FMLTT, a core type theory that extends Martin-Löf dependent type theory (MLTT) with facilities to express family polymorphism while maintaining consistency and canonicity.

For accessibility of the main text, Section 6 presents FMLTT using named variables and meta-level substitution. For clarity of proof details, a technical appendix [Jin et al. 2023b] supplementing Section 6 instead uses de Bruijn indices and explicit substitutions. We acknowledge that the presentation is dense for an audience without intimate knowledge of MLTT, so we summarize the salient points first.

Summary. FMLTT is intended as a foundational model. So unlike our programmer-facing plugin, fields automatically axiomatized by the plugin require explicit definitions in FMLTT. FMLTT provides MLTT-style constructs that can be used to express families and family polymorphism. Most notably, it extends MLTT with what we call *linkages*. Linkages are a namesake of the theoretical device through which Zhang and Myers [2017] model family polymorphism in an OO setting, but the technical details differ significantly from the prior work.

- Linkages model families, so they are like tuples composed of fields (with field names represented by variable bindings). But there is a twist: linkages support late binding. Unlike dependent tuples where a later component is *existentially* quantified over the earlier ones, a linkage component is *universally* quantified over—and thus polymorphic to—the components preceding it.
- FMLTT features *linkage transformers*, which model how a family can be inductively constructed by inheriting fields from another family, adding new fields, and overriding existing fields.
- Inductive types are modeled as W-types [Martin-Löf 1984] and their extension as overriding.
- Consistency and canonicity of FMLTT are proved by giving semantic interpretations to the syntactic typing judgments.

Brief review of MLTT. Figure 6 presents the syntax and selected typing rules of MLTT (and Figure 7 FMLTT). Dependent function types $\Pi(x : A).B$, dependent pair types $\Sigma(x : A).B$, and identity types $\mathsf{Eq}(t_1, t_2)$ are standard. So are their introduction and elimination forms. We use based path induction $\mathsf{J}(\cdot, t)$ as the elimination principle for a term t of an identity type [UFP 2013]. Capture-avoiding substitution is notated $\bullet[\bullet/x]$. We use x and *self* to denote variables. A singleton type $\mathbb{S}(t)$ helps expose the definition of a term t in its type (rule TM/S) [Aspinall 1995; Stone 2000]. Definitional equalities have the forms $\Gamma_1 \equiv \Gamma_2 \vdash$, $\Gamma \vdash T_1 \equiv T_2$, $\Gamma \vdash t_1 \equiv t_2 : T$, etc. Following Altenkirch and Kaposi [2016], we regard our (intrinsically typed) syntax as being quotiented by these equalities. Quotienting facilitates coercion along equalities—given $\Gamma_1 \equiv \Gamma_2 \vdash$ and $\Gamma_1 \vdash T_1 \equiv T_2$, the derivation of $\Gamma_1 \vdash t : T_1$ is considered definitionally equal to a derivation of $\Gamma_2 \vdash t : T_2$.

We use universes à la Coquand [2013], following Kaposi et al. [2019]. Unlike Russell-style ones, these universes are not inhabited by types directly, but rather by the *codes* of types, and arguably

$$\begin{array}{rrcl} \text{Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, A \\ \text{Types} & A, B, T & ::= & \mathbb{U} \mid \mathbb{B} \mid \bot \mid \top \mid \Pi(x:A).B \mid \Sigma(x:A).B \mid \mathtt{Eq}(t_1, t_2) \mid \mathbb{S}(t) \mid \mathtt{El}(t) \\ \text{Terms} & t & ::= & x \mid \mathtt{c}(T) \mid () \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(t_1, t_2, t_3) \mid \lambda x.t \mid \mathtt{app}(t_1, t_2) \mid \\ & & & (t_1, t_2) \mid \mathtt{fst}\ t \mid \mathtt{snd}\ t \mid \mathtt{refl}(t) \mid \mathtt{J}(t_1, t_2) \end{array}$$

$$\boxed{\Gamma \vdash} \qquad \boxed{\Gamma \vdash T} \qquad \boxed{\Gamma \vdash T_1 \equiv T_2} \qquad \boxed{\Gamma \vdash t : T} \qquad \boxed{\Gamma \vdash t_1 \equiv t_2 : T}$$

$$\text{(TM/LAM)}\ \frac{\Gamma \vdash A \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x:A).B} \qquad \text{(TM/APP)}\ \frac{\Gamma \vdash t : \Pi(x:A).B \quad \Gamma \vdash t' : A}{\Gamma \vdash \mathtt{app}(t, t') : B\big[t'/x\big]} \qquad \text{(TM/PAIR)}\ \frac{\Gamma, x : A \vdash B \quad \Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B[t_1/x]}{\Gamma \vdash (t_1, t_2) : \Sigma(x:A).B}$$

$$\text{(TM/FST)}\ \frac{\Gamma \vdash t : \Sigma(x:A).B}{\Gamma \vdash \mathtt{fst}\ t : A} \qquad \text{(TM/SND)}\ \frac{\Gamma \vdash t : \Sigma(x:A).B}{\Gamma \vdash \mathtt{snd}\ t : B[\mathtt{fst}\ t/x]} \qquad \text{(TY/EL)}\ \frac{\Gamma \vdash t : \mathbb{U}}{\Gamma \vdash \mathtt{El}(t)} \qquad \text{(TM/C)}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathtt{c}(T) : \mathbb{U}} \qquad \text{(TMEQ/C)}\ \frac{\Gamma \vdash t : \mathbb{U}}{\Gamma \vdash \mathtt{c}(\mathtt{El}(t)) \equiv t : \mathbb{U}}$$

$$\text{(TYEQ/EL)}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathtt{El}(\mathtt{c}(T)) \equiv T} \qquad \text{(TY/S)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbb{S}(t)} \qquad \text{(TM/S)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash t : \mathbb{S}(t)} \qquad \text{(TMEQ/S/ETA)}\ \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \mathbb{S}(t_1)}{\Gamma \vdash t_1 \equiv t_2 : T}$$

Figure 6. Syntax and selected typing rules of MLTT, named variables and meta-level substitution

behave better due to its closeness to Tarski-style universes [Luo 2012]. The term $\mathtt{c}(T)$ encodes type T (TM/C) and thus inhabits universe $\mathbb{U}$. The type $\mathtt{El}(t)$ decodes term t (TY/EL). There is an infinite hierarchy of universes; we omit universe levels in the presentation to avoid clutter.

For concision, typing rules omit obvious premises required for well-formedness. For example, TM/LAM implicitly requires $\Gamma \vdash$ (i.e., that the context be well-typed).

Introducing and eliminating inductive types. W-types [Martin-Löf 1984] are a succinct way to model inductive types in MLTT. Together with the identity type, they can express a whole host of inductive types [Hugunin 2020], including those with multiple constructors.

Our formulation of W-types differs from previous ones in that it is straightforward to identify constructors from what we call *W-type signatures*. A signature $\Gamma \vdash \tau\ \mathtt{WSig}^n$ is composed of n pairs of types (WSIG/EMPTY and WSIG/ADD), each modeling a constructor of the inductive type. The i-th pair, projected from the signature τ using the forms $\Gamma \vdash \mathtt{w}\pi_1^i(\tau)$ and $\Gamma \vdash \mathtt{w}\pi_2^i(\tau) : \mathtt{w}\pi_1^i(\tau) \to \mathbb{U}$, defines the i-th constructor. That is, given two arguments $\Gamma \vdash t_1 : \mathtt{w}\pi_1^i(\tau)$ and $\Gamma, x : \mathtt{El}(\mathtt{w}\pi_2^i(\tau)(t_1)) \vdash t_2 : \mathtt{El}(\mathtt{W}(\tau))$, one can construct a term $\mathtt{Wsup}_i(\tau, t_1, x.t_2)$ of the W-type $\mathtt{El}(\mathtt{W}(\tau))$ (TM/WSUP). $\mathtt{W}(\tau)$ gives the code of the W-type (TM/W).

For each pair of types identifying a constructor, the first type models the non-inductive arguments of the constructor, and the second type models the *arity* of the inductive arguments. For example, the signature $\tau_{\mathtt{tm}}$ of the W-type modeling the inductive type tm (Figure 2) is constructed as follows, where $\mathbb{0}$ is the bottom type $\bot$, $\mathbb{1}$ the unit type $\top$, $\mathbb{2}$ the boolean type, and $T_{\mathtt{id}}$ a type encoding id:

tm_unit : tm	tm_var : id → tm	tm_abs : id → tm → tm	tm_app : tm → tm → tm
$\tau^0_{\mathtt{tm}} := \mathtt{w}^+(\mathtt{w}^\bullet, \mathbb{1}, \lambda\_.\mathbb{0})$	$\tau^1_{\mathtt{tm}} := \mathtt{w}^+(\tau^0_{\mathtt{tm}}, T_{\mathtt{id}}, \lambda\_.\mathbb{0})$	$\tau^2_{\mathtt{tm}} := \mathtt{w}^+(\tau^1_{\mathtt{tm}}, T_{\mathtt{id}}, \lambda\_.\mathbb{1})$	$\tau_{\mathtt{tm}} := \mathtt{w}^+(\tau^2_{\mathtt{tm}}, \mathbb{1}, \lambda\_.\mathbb{2})$

While tm_unit and tm_var have no inductive arguments, tm_abs has one and tm_app has two. The encoding of tm_abs has type $T_{\mathtt{id}} \to (\mathbb{1} \to \mathtt{El}(\mathtt{W}(\tau_{\mathtt{tm}}))) \to \mathtt{El}(\mathtt{W}(\tau_{\mathtt{tm}}))$, and that of tm_app has type $\mathbb{1} \to (\mathbb{2} \to \mathtt{El}(\mathtt{W}(\tau_{\mathtt{tm}}))) \to \mathtt{El}(\mathtt{W}(\tau_{\mathtt{tm}}))$. These types are strictly positive by construction.

W-types are eliminated with the form $\mathtt{Wrec}(\tau, \ell, t)$, where t is of a W-type $\mathtt{El}(\mathtt{W}(\tau))$, and ℓ is essentially an n-tuple of case handlers for the n constructors in τ (TM/WREC). Each case handler has a type of the form $\mathtt{CaseTy}(A, B, T)$, where T is the motive of the recursion (TYEQ/CASETY); for

$$\begin{array}{rcll}
\text{Types} & A, B, T & ::= & \ldots \mid \mathtt{w}\pi_1^i(\tau) \mid \mathtt{w}\pi_2^i(\tau) \mid \mathbb{L}(\sigma) \mid \mathbb{P}(\sigma) \mid \nu\pi_2(\sigma) \mid \mathtt{CaseTy}(A, B, T) \\
\text{Terms} & t, s, \ell & ::= & \ldots \mid \mathtt{W}(\tau) \mid \mathtt{Wsup}_i(\tau, t_1, x.t_2) \mid \mu^\bullet \mid \mu^+(\ell, x.t) \mid \mathtt{inh}(h, \ell) \mid \mathtt{Wrec}(\tau, \ell, t) \mid \\
 & & & \mu\pi_1(\ell) \mid \mu\pi_2(\ell) \mid \nu\pi_{\mathrm{s}}(\sigma) \mid \mathbb{P}(\ell) \mid \mathrm{R}\pi^i(\ell) \\
\text{W-type signatures} & \tau & ::= & \mathtt{w}^\bullet \mid \mathtt{w}^+(\tau, A, B) \mid \mathtt{w}^-(\tau) \\
\text{Linkage signatures} & \sigma & ::= & \nu^\bullet \mid \nu^+(\sigma, x.s, \mathit{self}.T) \mid \nu\pi_1(\sigma) \mid \mathtt{RecSig}(\tau, T) \\
\text{Linkage transformers} & h & ::= & \mathtt{Identity} \mid \mathtt{Extend}(h, \mathit{self}.t) \mid \mathtt{Override}(h, \mathit{self}.t) \mid \mathtt{Inherit}(h) \mid \mathtt{Nest}(h, h')
\end{array}$$

$$\boxed{\Gamma \vdash T_1 \equiv T_2} \quad \boxed{\Gamma \vdash t : T} \quad \boxed{\Gamma \vdash t_1 \equiv t_2 : T} \quad \boxed{\Gamma \vdash \tau \ \mathtt{WSig}^n} \quad \boxed{\Gamma \vdash \sigma \ \mathtt{LSig}^n} \quad \boxed{\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2}$$

(TM/W)
$$\frac{\Gamma \vdash \tau \ \mathtt{WSig}^n}{\Gamma \vdash \mathtt{W}(\tau) : \mathbb{U}}$$

(WSIG/EMPTY)
$$\frac{}{\Gamma \vdash \mathtt{w}^\bullet \ \mathtt{WSig}^0}$$

(WSIG/ADD)
$$\frac{\Gamma \vdash \tau \ \mathtt{WSig}^n \qquad \Gamma \vdash A \qquad \Gamma \vdash B : A \to \mathbb{U}}{\Gamma \vdash \mathtt{w}^+(\tau, A, B) \ \mathtt{WSig}^{n+1}}$$

(TM/WSUP)
$$\frac{\Gamma \vdash \tau \ \mathtt{WSig}^n \qquad \Gamma \vdash t_1 : \mathtt{w}\pi_1^i(\tau) \qquad \Gamma, x : \mathtt{El}(\mathtt{app}(\mathtt{w}\pi_2^i(\tau), t_1)) \vdash t_2 : \mathtt{El}(\mathtt{W}(\tau))}{\Gamma \vdash \mathtt{Wsup}_i(\tau, t_1, x.t_2) : \mathtt{El}(\mathtt{W}(\tau))}$$

(TM/WREC)
$$\frac{\Gamma \vdash \ell : \mathbb{L}(\mathtt{RecSig}(\tau, T)) \qquad \Gamma \vdash t : \mathtt{El}(\mathtt{W}(\tau))}{\Gamma \vdash \mathtt{Wrec}(\tau, \ell, t) : T}$$

(TYEQ/CASETY)
$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B : A \to \mathbb{U} \qquad \Gamma \vdash T}{\Gamma \vdash \mathtt{CaseTy}(A, B, T) \equiv \Pi(x : A).(\mathtt{El}(\mathtt{app}(B, x)) \to T) \to T}$$

(LSIG/EMPTY)
$$\frac{}{\Gamma \vdash \nu^\bullet \ \mathtt{LSig}^0}$$

(LSIG/ADD)
$$\frac{\Gamma \vdash \sigma \ \mathtt{LSig}^n \qquad \Gamma, \mathit{self} : A \vdash T \qquad \Gamma, x : \mathbb{P}(\sigma) \vdash s : A}{\Gamma \vdash \nu^+(\sigma, x.s, \mathit{self}.T) \ \mathtt{LSig}^{n+1}}$$

(L/EMPTY)
$$\frac{}{\Gamma \vdash \mu^\bullet : \mathbb{L}(\nu^\bullet)}$$

(L/ADD)
$$\frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \qquad \Gamma, \mathit{self} : A \vdash t : T \qquad \Gamma, x : \mathbb{P}(\sigma) \vdash s : A}{\Gamma \vdash \mu^+(\ell, \mathit{self}.t) : \mathbb{L}(\nu^+(\sigma, x.s, \mathit{self}.T))}$$

(TYEQ/PK/ADD)
$$\frac{\Gamma \vdash \sigma \ \mathtt{LSig}^n \qquad \Gamma, \mathit{self} : A \vdash T \qquad \Gamma, x : \mathbb{P}(\sigma) \vdash s : A}{\Gamma \vdash \mathbb{P}(\nu^+(\sigma, x.s, \mathit{self}.T)) \equiv \Sigma(x : \mathbb{P}(\sigma)).T[s/\mathit{self}]}$$

(TMEQ/PK/ADD)
$$\frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \qquad \Gamma, \mathit{self} : A \vdash t : T \qquad \Gamma, x : \mathbb{P}(\sigma) \vdash s : A}{\Gamma \vdash \mathbb{P}(\mu^+(\ell, \mathit{self}.t)) \equiv (\mathbb{P}(\ell), t[s[\mathbb{P}(\ell)/x]/\mathit{self}]) : \mathbb{P}(\nu^+(\sigma, s, T))}$$

(TM/INH)
$$\frac{\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2 \qquad \Gamma \vdash \ell : \mathbb{L}(\sigma_1)}{\Gamma \vdash \mathtt{inh}(h, \ell) : \mathbb{L}(\sigma_2)}$$

(TMEQ/OV/BETA)
$$\frac{\begin{array}{c}\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2 \qquad \Gamma \vdash \ell : \mathbb{L}(\sigma_1) \\ \Gamma, \mathit{self}_1 : A_1 \vdash t_1 : T_1 \qquad \Gamma, x_1 : \mathbb{P}(\sigma_1) \vdash s_1 : A_1 \qquad \Gamma, \mathit{self}_2 : A_2 \vdash t_2 : T_2 \qquad \Gamma, x_2 : \mathbb{P}(\sigma_2) \vdash s_2 : A_2\end{array}}{\Gamma \vdash \mathtt{inh}(\mathtt{Override}(h, \mathit{self}_2.t_2), \mu^+(\ell, \mathit{self}_1.t_1)) \equiv \mu^+(\mathtt{inh}(h, \ell), \mathit{self}_2.t_2) : \mathbb{L}(\nu^+(\sigma_2, x_2.s_2, \mathit{self}_2.T_2))}$$

Figure 7. Syntax and selected typing rules of FMLTT.

simplicity, we model only non-dependent motives. The collection of case handlers ℓ encodes those defined and inherited by an FRecursion command in our plugin. We choose to type it with a linkage type $\mathbb{L}(\mathtt{RecSig}(\tau, T))$ to avoid introducing non-dependent n-tuples, which linkages generalize.

The rules TM/WSUP and TM/WREC require access to τ: the W-type is exhaustively generated by its constructors, and its elimination must exhaustively handle all the constructors in its signature.

In contrast, τ should be hidden from the typing context of any term that does not invoke $\mathtt{Wsup}(\tau, \bullet, \bullet)$ or $\mathtt{Wrec}(\tau, \bullet, \bullet)$, so that the term can be reused—without being rechecked—for a different W-type signature τ' that extends τ with additional constructors. Moreover, the typing of the term should be made parametric to the definitions of those fields that invoke Wsup or Wrec, so that the term can be reused—without being rechecked—when those fields are overridden to support the extended signature τ'. Such abstraction required by family polymorphism is supported via linkages, which we discuss next.

Family polymorphism via linkages. Family polymorphism requires late binding. In FMLTT, families are expressed through linkages, and late binding of field references is achieved by requiring that typing be polymorphic to the definition of that field.

Linkage signatures have the judgment form $\Gamma \vdash \sigma\ \texttt{LSig}^n$. A linkage signature is a list of n types (LSIG/EMPTY and LSIG/ADD). *Linkages* have the judgment form $\Gamma \vdash \ell : \mathbb{L}(\sigma)$, where $\mathbb{L}(\sigma)$ is a type formed by σ. A linkage is a list of n terms, each representing a field of the family modeled by the linkage (L/EMPTY and L/ADD).

In the rules L/ADD and LSIG/ADD, the second premise $\Gamma, \mathit{self} : A \vdash t : T$ is responsible for late binding. Here, A abstracts the context of the current field t, controlling how the types of the fields prior to t are exposed to the typing of t. As discussed later, the third premise $\Gamma, x : \mathbb{P}(\sigma) \vdash s : A$ is responsible for creating the context type A that possibly hides W-type signatures in σ, which records the types of the prior fields.

Crucially, the premise $\Gamma, \mathit{self} : A \vdash t : T$ makes clear that the typing of t in L/ADD is *universally* quantified—rather than *existentially* quantified as is in TM/PAIR—over how the fields in t's context are defined. Late binding enables reuse. A different linkage ℓ' that overrides fields in t's context (and thus models a derived family) can reuse t—without retyping it—by first projecting t from $\mu^+(\ell, \mathit{self}.t)$ and then appending it to ℓ'.

The type A in L/ADD and LSIG/ADD, abstracting the types of the prior fields, does not necessarily contain the same types as those recorded by $\mathbb{L}(\sigma)$, because a field defined as the code $\texttt{W}(\tau)$ of a W-type has to expose different types to different fields that come after it. Later fields that invoke $\texttt{Wsup}(\tau, \bullet, \bullet)$ or $\texttt{Wrec}(\tau, \bullet, \bullet)$ should see the concrete signature τ, as the rules TM/WSUP and TM/WREC stipulate. By contrast, τ should be hidden from all other fields, so that they can be reused in a different context where the W-type signature τ is replaced by an extended one τ'.

We use Figure 8 to illustrate. On the left is code excerpted from Figure 2, and on the right is how the corresponding fields are modeled and typed in FMLTT. ℓ_i is the linkage that adds the i-th field t_i with the typing $\mathit{self} : A_i \vdash t_i : T_i$. The context types A_i are a dependent tuple type, but for readability, we write them as dependent record types that give labels to the fields. Field accesses are notated, for example, as $\mathit{self}_{\triangleright}\texttt{tm}$.

- The first field t_1 is defined as $\texttt{W}(\tau_{\texttt{tm}})$ and given the singleton type $\mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))$, where $\tau_{\texttt{tm}}$ is the W-type signature constructed earlier for the extensible inductive type tm. This typing is recorded by σ_1 and is thus available in all σ_i's.
- The next four fields model the four constructors of tm. Constructor tm_unit is modeled as $\texttt{Wsup}(\tau_{\texttt{tm}}, \top, x.\bot)$ and has type $\texttt{El}(\mathit{self}_{\triangleright}\texttt{tm})$, where *self* stands for the typing context containing the first field tm. The W-type signature $\tau_{\texttt{tm}}$ is exposed in this typing context; that is, $\mathit{self}_{\triangleright}\texttt{tm}$ has type $\mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))$. So $\texttt{El}(\mathit{self}_{\triangleright}\texttt{tm})$ and $\texttt{El}(\texttt{W}(\tau_{\texttt{tm}}))$ can be equated, as required by rule TM/WSUP.
- Likewise, $\tau_{\texttt{tm}}$ is exposed in the typing context of t_{10}, which models the recursive function subst by invoking the recursor $\texttt{Wrec}(\tau_{\texttt{tm}}, \bullet, \bullet)$. Like subst, partial recursors in FPOP are axiomatized by the plugin (Sections 3.2 and 3.6), and they can similarly be defined in FMLTT using Wrec.
- By contrast, $\tau_{\texttt{tm}}$ is hidden from the typing of all other fields. Their typing should depend on the knowledge that tm has type $\mathbb{U}$, rather than $\mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))$, so that they can be reused in a context where tm is defined as $\texttt{W}(\tau'_{\texttt{tm}})$, where $\tau'_{\texttt{tm}}$ extends $\tau_{\texttt{tm}}$ with additional constructors. For example, in Figure 8, the typing of the case handlers of subst (e.g., t_6) is oblivious to the definition of tm—it sees only $\texttt{tm} : \mathbb{U}$—so the case handlers can be reused by a linkage modeling STLCFix.

In L/ADD and LSIG/ADD, the third premise $\Gamma, x : \mathbb{P}(\sigma) \vdash s : A$ is responsible for hiding W-type signatures. Here, $\mathbb{P}(\sigma)$ packages σ—recall that σ contains the (self-parameterized) types of all the

$x : \mathbb{P}(\sigma_{i-1}) \vdash s_i : A_i$	$\sigma_i \coloneqq \nu^+(\sigma_{i-1}, x.s_i, \mathit{self}.\,T_i)$	$\mathbb{P}(\sigma_i) \equiv \Sigma(x : \mathbb{P}(\sigma_{i-1})).T_i[s_i/\mathit{self}]$
$\mathit{self} : A_i \vdash t_i : T_i$	$\ell_i \coloneqq \mu^+(\ell_{i-1}, \mathit{self}.\,t_i)$	$\mathbb{P}(\ell_i) \equiv \big(\mathbb{P}(\ell_{i-1}), t_i[s_i[\mathbb{P}(\ell_{i-1})/x]/\mathit{self}]\big)$
shown below for $i \in \{1, 2, 6, 10\}$	$\cdot \vdash \ell_i : \mathbb{L}(\sigma_i)$	$\cdot \vdash \mathbb{P}(\ell_i) : \mathbb{P}(\sigma_i)$

`Family STLC.`	$\sigma_0 \coloneqq \nu^\bullet \qquad \ell_0 \coloneqq \mu^\bullet$
`FInductive tm ≔`	$x : [\,] \vdash s_1 : [\,]$ $\mathit{self} : [\,] \vdash \texttt{W}(\tau_{\texttt{tm}}) : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))$ $\sigma_1 \coloneqq \nu^+(\sigma_0, x.s_1, \mathit{self}.\,T_1) \qquad \ell_1 \coloneqq \mu^+(\ell_0, \mathit{self}.\,t_1)$
`| tm_unit : tm`	$x : [\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))] \vdash s_2 : [\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))]$ $\mathit{self} : [\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))] \vdash \texttt{Wsup}(\tau_{\texttt{tm}}, \top, x.\bot) : \texttt{El}(\mathit{self}_{\triangleright}\texttt{tm})$ $\sigma_2 \coloneqq \nu^+(\sigma_1, x.s_2, \mathit{self}.\,T_2) \qquad \ell_2 \coloneqq \mu^+(\ell_1, \mathit{self}.\,t_2)$
`| …	…	….`	…
`FRecursion subst ….`	*$\texttt{W}(\tau_{\texttt{tm}})$ is abstracted into $\mathbb{U}$ in the typing context (i.e., A_6) of the case handler below:*
`Case tm_unit ≔ ….`	$x : [\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}})),\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots] \vdash s_6 : [\texttt{tm} : \mathbb{U},\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots]$ $\mathit{self} : [\texttt{tm} : \mathbb{U},\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots] \vdash \ldots : \texttt{CaseTy}\left(\top, \bot, \begin{matrix}\texttt{El}(\mathit{self}_{\triangleright}\texttt{tm}) \to \\ \mathit{id} \to \texttt{El}(\mathit{self}_{\triangleright}\texttt{tm})\end{matrix}\right)$ $\sigma_6 \coloneqq \nu^+(\sigma_5, x.s_6, \mathit{self}.\,T_6) \qquad \ell_6 \coloneqq \mu^+(\ell_5, \mathit{self}.\,t_6)$
`Case … Case … Case …`	…
`End subst.`	$x : \begin{bmatrix}\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}})),\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots, \\ \texttt{subst\_tm\_unit} : \texttt{CaseTy}(\top, \bot, \ldots),\ \ldots\end{bmatrix} \vdash s_{10} : \begin{bmatrix}\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}})),\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots, \\ \texttt{subst\_tm\_unit} : \texttt{CaseTy}(\top, \bot, \ldots),\ \ldots\end{bmatrix}$ $\mathit{self} : \begin{bmatrix}\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}})),\ \texttt{tm\_unit} : \texttt{El(tm)},\ \ldots, \\ \texttt{subst\_tm\_unit} : \texttt{CaseTy}(\top, \bot, \ldots),\ \ldots\end{bmatrix} \vdash \lambda x.\texttt{Wrec}(\tau_{\texttt{tm}}, \ldots, x) : \begin{matrix}\texttt{El}(\mathit{self}_{\triangleright}\texttt{tm}) \to \\ \texttt{El}(\mathit{self}_{\triangleright}\texttt{tm}) \to \\ \mathit{id} \to \texttt{El}(\mathit{self}_{\triangleright}\texttt{tm})\end{matrix}$ $\sigma_{10} \coloneqq \nu^+(\sigma_9, x.s_{10}, \mathit{self}.\,T_{10}) \qquad \ell_{10} \coloneqq \mu^+(\ell_9, \mathit{self}.\,t_{10})$
…	…
`End STLC.`	$\cdot \vdash \mathbb{P}(\ell_{53}) : \mathbb{P}(\sigma_{53})$

Figure 8. FMLTT encoding of the STLC family from Figure 2. Each row $\mathit{self} : A_i \vdash t_i : T_i$ types a field. The type A_i controls how the typing of field t_i sees the types of the fields prior to t_i.

fields preceding the current field t—into a dependent tuple type (TYEQ/PK/ADD). The term s turns a tuple of type $\mathbb{P}(\sigma)$ into a new tuple of type A that hides W-type signatures behind $\mathbb{U}$, if necessary.[2]

It is straightforward to find the s that fits the bill, though this process is not automated in FMLTT. In particular, when no hiding is needed (that is, when the field being checked invokes a W-type constructor or eliminator), s is simply x, and A is $\mathbb{P}(\sigma)$. In Figure 8, s_2 and s_{10} are x. Otherwise, it is needed to hide W-type signatures. In Figure 8, s_6 hides $\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))$ as $\texttt{tm} : \mathbb{U}$ in A_6, so that t_6, typed under $\mathit{self} : A_6$, is oblivious to the concrete signature $\tau_{\texttt{tm}}$ and therefore can be reused.

When a family is concluded (e.g., `End STLC`), a linkage ℓ containing all the fields is available (e.g., ℓ_{53} in Figure 8). Fields of the family can then be accessed by projecting them out of the tuple $\mathbb{P}(\ell)$. As TMEQ/PK/ADD indicates, $\mathbb{P}(\ell)$ ties the recursive knot: it packages the linkage ℓ into a dependent tuple of type $\mathbb{P}(\sigma)$, by instantiating the *self* parameters.

[2]In the presence of hiding, it is important that the context type A be a dependent tuple type rather than a linkage type; we expand on this point in an appendix.

Linkage transformers. Inheritance and code reuse can already be expressed through the projection of fields out of linkages and their inclusion into new linkages. To make common patterns of linkage manipulations more convenient, FMLTT provides a "library" of *linkage transformers*, whose well-formedness judgments have form $\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2$. The idea is that applying h to a linkage of type $\mathbb{L}(\sigma_1)$ yields a linkage of type $\mathbb{L}(\sigma_2)$ (TM/INH).

Derived families can be modeled as linkage transformers inductively constructed from the introduction forms $\texttt{Identity}$, $\texttt{Extend}(h, \mathit{self}.t)$, $\texttt{Override}(h, \mathit{self}.t)$, etc. Figure 7 shows the β-rule of an example transformer, TMEQ/OV/BETA. It states that applying the transformer $\texttt{Override}(h, \mathit{self}_2.t_2)$ to a linkage of form $\mu^+(\ell, \mathit{self}_1.t_1)$ overrides the linkage's last field t_1 with t_2. For instance, the construction below shows that $\texttt{Override}(\texttt{Identity}, \mathit{self}'.\texttt{W}(\tau'_{\texttt{tm}}))$ is used as the first step in creating a linkage transformer modeling a derived family that overrides $\tau_{\texttt{tm}}$ with an extended signature $\tau'_{\texttt{tm}}$:

`Family STLCFix extends STLC.`	$h_0 \coloneqq \texttt{Identity}$	$\cdot \vdash h_0 : \nu^\bullet \twoheadrightarrow \nu^\bullet$
`FInductive tm += ...`	$h_1 \coloneqq \texttt{Override}(h_0, \mathit{self}.\texttt{W}(\tau'_{\texttt{tm}}))$	$\cdot \vdash h_1 : \begin{array}{l} \nu^+(\nu^\bullet, x.s_1, \mathit{self}.\mathbb{S}(\texttt{W}(\tau_{\texttt{tm}}))) \twoheadrightarrow \\ \nu^+(\nu^\bullet, x'.s'_1, \mathit{self}'.\mathbb{S}(\texttt{W}(\tau'_{\texttt{tm}}))) \end{array}$

A supplemental appendix sketches how the other introduction forms of linkage transformers can be used to model the construction of a derived family as a linkage transformer.

The complete formalization. The definitive version containing all the rules in FMLTT is stated in a technical appendix [Jin et al. 2023b]. This definitive version uses de Bruijn indices and explicit substitutions [Abadi et al. 1989], for exactness of technical details.

Consistency and canonicity. One of the most fundamental properties of a dependent type theory is consistency.

THEOREM 6.1 (CONSISTENCY). *The typing judgment* $\cdot \vdash t : \bot$ *is not derivable for any term* t.

Consistency says that the type $\bot$ is not inhabited. Thus, it is safe to use the type theory for logical reasoning, as not every proposition is trivially provable.

A second property we prove is canonicity.

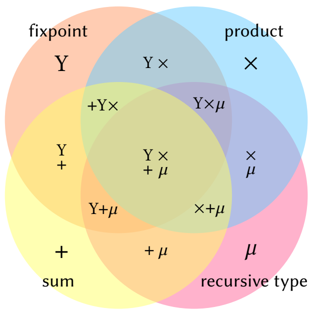
THEOREM 6.2 (CANONICITY). *If* $\cdot \vdash t : \mathbb{B}$, *then either* $\cdot \vdash t \equiv \texttt{tt} : \mathbb{B}$ *or* $\cdot \vdash t \equiv \texttt{ff} : \mathbb{B}$.

This canonicity theorem says that every closed term of the ground type $\mathbb{B}$ is convertible to one of the canonical forms `tt` and `ff`. When the canonicity theorem is proved in a constructive metalogic, its proof amounts to a normalization function for closed terms of the ground type. So canonicity serves to justify the computational nature of the type theory.

We prove Theorems 6.1 and 6.2 by constructing a logical-relations model for the well-formedness rules, following prior approaches [Coquand 2019; Kaposi et al. 2019; Sterling 2019]. The model interprets a linkage $\mu^+(\ell, t)$ as a pair where the second component is a function (modeling late binding), as rule L/ADD indicates. The model interprets the bottom type $\bot$ as an empty set, from which Theorem 6.1 follows. A closed, well-formed type $\cdot \vdash T$ is interpreted as a logical predicate on closed terms: the predicate includes all closed, "reducible" terms of type T. Theorem 6.2 follows from this interpretation. The construction of the logical-relations model is available in an appendix.

7 CASE STUDIES

Type safety of STLCs. The first case study is the mechanization of the type safety theorem of STLC and those of its extensions, which has been occurring in the examples in this paper. The code base is ported from Software Foundations [Pierce et al. 2022]. The linguistic nature of our approach allows us to retain a programming style similar to the original proofs in Software Foundations.

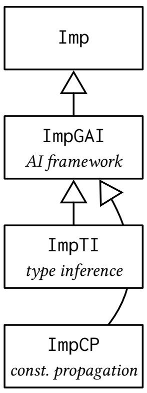


The base STLC family consists of ~360 LOC, about the same as an STLC development not using FPOP. Lines of code in each of the four derived families (Y, ×, +, and μ in the Venn diagram) vary from 100 to 250, largely depending on how many constructors they add to the inductive types. Without FPOP, the same STLC code would have to be duplicated for each feature.

Using individual families to organize the mechanization of individual language features leads to a modular design that also facilitates code reuse. Individually developed features can be easily composed (as mixins) to form new STLC variants (e.g., Y+μ). Such a feature composition often requires only a few lines of code.

Composing features can lead to *feature interactions* [Batory et al. 2011]: features working correctly in isolation may require coordination when composed. For example, composing × and μ (Figure 3) creates an obligation to extend `tysubst` to handle `ty_prod`, which the type-checker enforces. Composing contradictory features (e.g., a fixpoints construct and a strong normalization theorem) would lead to unprovable proof obligations.

Elimination of inductive types defined via `FInductive` is mostly via the `FRecursion` and `FInduction` commands. An exception is a handful of trivial "inversion lemmas". For example, consider the lemma `∀ t, ¬ step tm_true t` stating that `tm_true` is irreducible. If `step` were an ordinary inductive type, then it could be proved in Coq simply by `intros t H; inversion H`. But `step` is extensible. So one way to prove the lemma is by `FInduction` on `step` and verifying that a derived family does not accidentally make `tm_true` reducible. We observe that it is lighter-weight to use overriding (Section 3.3) instead: the programmer can specify that the proof of the lemma should be overridden in any derived family that further binds `step`, and in return, they are permitted to treat `step` as an ordinary inductive type in the proof and thus use `inversion` to prove it. The plugin then automatically tries the same proof script in a derived family to override the proof. Although proof scripts rather than proof terms are reused, this practice seems justified by the triviality of the lemmas and the terseness of the proof scripts.



Abstract interpreters for imperative languages. Our second case study is a mechanization of abstract interpreters for simple imperative languages. In addition to a soundness proof, this case study produces abstract interpreters that are directly ready for program extraction.

The code is organized into four families. A base family `Imp` (~200 LOC) defines via `FInductive` the abstract syntax of a while-language with pure expressions and impure statements. The semantics is given by an interpreter defined as a CEK-style abstract machine [Felleisen and Friedman 1986] and parameterized by a fuel value. Family `Imp` defines the interpreter via `FRecursion`.

A second family `ImpGAI` (~550 LOC) extends `Imp`. It exports a generic framework for deriving abstract interpreters with partial-correctness guarantees. Soundness of the abstract interpreter, `analyze`, is stated with respect to the interpreter, `eval`, inherited from `Imp`. The theorem says that the concretization relation `RState` over a concrete state `S` and an abstract state `absS` is preserved by the analysis:

```
∀ stmt fuel S absS, RState S absS → RState (eval fuel stmt S) (analyze fuel stmt absS)
```

`analyze` is defined via `FRecursion`, and the soundness theorem is proved via `FInduction`. This family leaves fields representing the abstract domain, the concretization relation, monotonicity of transfer functions, etc. largely unspecified or unproven—a derived family can further bind these "parameters" by overriding appropriate fields (and also possibly extend the abstract syntax), to create a sound, runnable abstract interpreter for a (possibly extended) while-language.

The next two families both extend `ImpGAI`. Family `ImpTI` (~200 LOC) is an abstract interpreter doing type inference [Cousot 1997]. Family `ImpCP` (~300 LOC) extends the abstract syntax with natural-number arithmetic, and further binds the generic abstract interpreter to perform constant propagation.

Our implementation of family polymorphism is compatible with Coq's program extraction feature. We extract the two verified abstract interpreters to OCaml. Testing the extracted program over simple queries returns expected results.

In addition to the two case studies above, we also use extensible inductive types for modeling extensible context-free grammars and derive decision procedures for language membership.

8 RELATED WORK

Approaches to modular mechanization or proof reuse exist, with different focuses and trade-offs.

Encodings based on product lines or DTC. Delaware et al. [2011] engineer product lines of theorems and proofs built from feature modules. Feature composition is manual, which seems to have motivated later approaches based on *data types à la carte* (DTC) [Swierstra 2008].

The original DTC encoding requires type-level general recursion that fails the strict positivity check imposed by proof assistants including Coq and Agda. Delaware et al. [2013a] introduce *metatheory à la carte* (MTC): it overcomes the problem by using Church encodings for data types and using Mendler-style folds for evaluation, though it requires `Set` impredicativity. Feature composition is automated through heavy use of type classes. The framework is implemented as a Coq library. Schwaab and Siek [2013] adapt DTC to Agda by considering a restricted class of functors that admit least fixed points. Keuchel and Schrijvers [2013] use datatype-generic programming techniques for the underlying representation of type-level fixed points and avoid `Set` impredicativity.

All of these approaches are largely extralinguistic, in that they work within the confine of the language offered by a proof assistant, which comes with trade-offs. On the one hand, they can be conveniently distributed as libraries, and the encoding can be more easily adapted for new purposes. For example, MTC has been applied to implementing composable program adverbs [Li and Weirich 2022] and been adapted to allow feature extensions, such as reference cells and exceptions, that require type changes [Delaware et al. 2013b; van der Rest et al. 2022].

On the other hand, the extralinguistic nature of the approaches tends to lead to non-idiomatic code and offset their user-friendliness. In particular, because data types have to be encoded (rather than expressed through natively supported inductive types), the resulting code can be obtuse at first blush, making the programming style inaccessible to non-experts. In addition, extra programmer effort may be required, such as having to manually prove additional well-formedness conditions.

Forster and Stark [2020] introduce *Coq à la carte*. It still follows DTC, but rather than embracing DTC's use of generic fixed points, it considers specific instantiations instead. The resulting mechanism appears more streamlined than prior *à la carte* approaches particularly for its extensive tool support for generating boilerplate code. But even with the tool support, components (e.g., `substlem`) of individually developed feature extensions have to be composed separately by invoking the tool.

Our approach addresses the expression problem by extending the linguistic facilities offered by a proof assistant. Families, in particular, offer an organizational advantage. They allow grouping and coevolving related types, functions, and proofs without explicit parameterization; all further-bindable fields are automatically extensibility hooks. Because family polymorphism does not require explicit parameterization or complex encodings, the resulting programming experience and code are accessible to the working Coq programmer. The more OO aspects of family polymorphism, such as the ability to use families as mixins and the ability to grow a series of mechanized languages in integral increments, also facilitate extensibility and reuse with minimal programmer effort.

Proof reuse and proof repair. Boite [2004] addresses proof reuse specifically in response to inductive types extended with new constructors. Proof reuse is via a tactic that adapts the original proof to the extended inductive type while generating proof obligations, so rechecking of proof terms is entailed. The design requires distinct names for a base inductive type and its extensions (including distinct names for constructors), while FPOP allows names to be late bound.

Mulhern [2006] introduces a heuristic approach that allows proofs for multiple small languages to be combined to yield proofs for composite languages, as long as the proof structure follows the same pattern. Johnsen and Lüth [2004] enable proof reuse in Isabelle by adapting theorems from one setting for reuse in another: proof terms are transformed by first explicitly stating all assumptions, and then abstracting over function symbols and type constants.

Pumpkin Pi [Ringer et al. 2021] is a Coq plugin that helps repair proofs broken by changes in type definitions. Its decompiler from proof terms to proof scripts prioritizes suggesting useful tactics over soundness. While Pumpkin Pi focuses on refactoring existing proofs in response to changed definitions, our solution can be viewed as an effort to preempt refactoring by enabling the programmer to write code that has built-in hooks for future extensions.

The expression problem. Solutions abound. Almost all involve some form of either explicit or implicit parameterization as extensibility hooks. Our approach is the first that applies family polymorphism [Ernst 2001], mostly seen in OO languages, to the context of mechanized proofs.

Blume et al. [2006] address the expression problem for a core subset of Standard ML by combining explicitly coded open recursion with a design that allows pattern-matching cases to be defined separately and combined later. Our `FRecursion` and `FInduction` commands achieve a similar functionality, with families making open recursion implicit and bestowing organizational power.

ML-style modules, like families, are a modularity mechanism, but with a focus on abstraction rather than extensibility. Both FMLTT and the module system of Stone and Harper [2000] use singletons to model and control the propagation of definitions. MixML [Rossberg and Dreyer 2013] integrates mixins into ML and handles the idiosyncrasies of ML modules, while our work supports mixins in the presence of extensible inductive types.

9 CONCLUSION

It is hard to write modular, extensible code and proofs. We have presented a solution that equips a proof assistant with linguistic facilities for family polymorphism. The language design ensures that the expressive power brought by family polymorphism is in harmony with the strictness of a proof assistant, while incurring low cognitive overhead and allowing an idiomatic programming experience. We implement the design via a translation to Coq and demonstrate its applicability using case studies. A novel dependent type theory formalizes the essence of the language mechanism and is shown to enjoy consistency and canonicity. Future work can explore ways to further improve the practicality and expressivity of the language mechanism.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their many useful comments. We thank Benjamin Delaware, Anastasiya Kravchuk-Kirilyuk, Yao Li, Andrew Myers, and Zhixuan Yang for valuable discussions and feedback. This work was supported in part by a gift from Amazon. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any funding agency.

DATA AVAILABILITY STATEMENT

The FPOP implementation is available through Jin et al. [2023a].

REFERENCES

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1989. Explicit substitutions. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/96709.96712

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837638

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*. https://doi.org/10.1007/11687061_5

David Aspinall. 1995. Subtyping with singleton types. In *Int'l Workshop on Computer Science Logic*. https://doi.org/10.1007/BFb0022243

Don Batory, Peter Höfner, and Jongwook Kim. 2011. Feature interactions, products, and composition. In *ACM Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. https://doi.org/10.1145/2047862.2047867

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2006. Extensible programming with first-class cases. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/1159803.1159836

Olivier Boite. 2004. Proof reuse with extended inductive types. In *Int'l Conf. on Theorem Proving in Higher Order Logics*.

Gilad Bracha and William Cook. 1990. Mixin-based inheritance. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/97945.97982

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: A simple virtual class calculus. In *International Conference on Aspect-Oriented Software Development (AOSD)*. https://doi.org/10.1145/1218563.1218578

William R. Cook, Walter L. Hill, and Peter S. Canning. 1990. Inheritance is not subtyping. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/96709.96721

Coq [n.d.]. The Coq proof assistant. https://coq.inria.fr.

Thierry Coquand. 2013. Presheaf model of type theory. (2013). https://www.cse.chalmers.se/~coquand/presheaf.pdf

Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777, C (2019). https://doi.org/10.1016/j.tcs.2019.01.015 arXiv:1810.09367

Patrick Cousot. 1997. Types as abstract interpretations. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263744

Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75, 5 (1972). https://doi.org/10.1016/1385-7258(72)90034-0

Benjamin Delaware, William Cook, and Don Batory. 2011. Product lines of theorems. *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/2076021.2048113

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013a. Meta-theory à la carte. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2429069.2429094

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. 2013b. Modular monadic meta-theory. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/2500365.2500587

Erik Ernst. 2001. Family polymorphism. In *European Conf. on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/3-540-45337-7_17

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1111037.1111062

Matthias Felleisen and Daniel P. Friedman. 1986. *Control Operators, the SECD-machine, and the λ-calculus.* Technical Report. Computer Science Department, Indiana University.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/268946.268961

Yannick Forster and Kathrin Stark. 2020. Coq à la carte: A practical approach to modular syntax with binders. In *ACM SIGPLAN Conf. on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3372885.3373817

Jasper Hugunin. 2020. Why not W?. In *Int'l Conf. on Types for Proofs and Programs (TYPES)*. https://doi.org/10.4230/LIPIcs.TYPES.2020.8

Atsushi Igarashi and Mirko Viroli. 2007. Variant path types for scalable extensibility. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/1297027.1297037

Ende Jin, Nada Amin, and Yizhou Zhang. 2023a. Extensible metatheory mechanization via family polymorphism: Artifact. https://doi.org/10.5281/zenodo.7800226

Ende Jin, Nada Amin, and Yizhou Zhang. 2023b. *Extensible Metatheory Mechanization via Family Polymorphism: Technical Report.* Technical Report CS-2023-01. School of Computer Science, University of Waterloo. https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2023-01.pdf

Einar Broch Johnsen and Christoph Lüth. 2004. Theorem reuse by proof term transformation. In *Int'l Conf. on Theorem Proving in Higher Order Logics.* https://doi.org/10.1007/978-3-540-30142-4_12

Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for type theory. In *Int'l Conf. on Formal Structures for Computation and Deduction (FSCD).* https://doi.org/10.4230/LIPIcs.FSCD.2019.25

Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *9th ACM SIGPLAN Workshop on Generic Programming.*

Yao Li and Stephanie Weirich. 2022. Program adverbs and Tlön embeddings. *Proc. of the ACM on Programming Languages (PACMPL)* 6, ICFP (2022). https://doi.org/10.1145/3547632 arXiv:2207.05227

Zhaohui Luo. 2012. Notes on universes in type theory. http://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf Notes for a talk at the Institute for Advanced Study in Princeton.

Ole L. Madsen and Birger Moller-Pedersen. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/74877.74919

Per Martin-Löf. 1982. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI.* Studies in Logic and the Foundations of Mathematics, Vol. 104. https://doi.org/10.1016/S0049-237X(09)70189-2

Per Martin-Löf. 1984. *Intuitionistic Type Theory: Notes by Giovanni Sambin of a Series of Lectures Given in Padua, June 1980 (Studies in Proof Theory).*

Anne Mulhern. 2006. Proof weaving. In *1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory.*

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable extensibility via nested inheritance. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/1028976.1028986

Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/1094811.1094815

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2022. Software foundations: Volume 2 (programming language foundations). (2022). Version 6.2.

John C. Reynolds. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages.* https://doi.org/10.1007/978-1-4612-6315-9_22

Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3453483.3454033

Andreas Rossberg and Derek Dreyer. 2013. Mixin' up the ML module system. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 35, 1 (April 2013). https://doi.org/10.1145/2450136.2450137

Christopher Schwaab and Jeremy G. Siek. 2013. Modular type-safety proofs in Agda. In *7th Workshop on Programming Languages Meets Program Verification.* https://doi.org/10.1145/2428116.2428120

Jonathan Sterling. 2019. Algebraic type theory and universe hierarchies. (2019). arXiv:1902.08848

Christopher A. Stone. 2000. *Singleton kinds and singleton types.* Ph.D. Dissertation. Carnegie Mellon University.

Christopher A. Stone and Robert Harper. 2000. Deciding type equivalence in a language with singleton kinds. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL).* https://doi.org/10.1145/325694.325724

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming (JFP)* 18, 4 (2008). https://doi.org/10.1017/S0956796808006758

Kresten Krab Thorup. 1997. Genericity in Java with virtual types. In *European Conf. on Object-Oriented Programming (ECOOP).* https://doi.org/10.1007/BFb0053390

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book. arXiv:1308.0729

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. of the ACM on Programming Languages (PACMPL)* 6, OOPSLA2 (2022). https://doi.org/10.1145/3563355

Philip Wadler et al. 1998. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt Discussion on the Java-genericity mailing list.

Yizhou Zhang and Andrew C. Myers. 2017. Familia: Unifying interfaces, type classes, and family polymorphism. *Proc. of the ACM on Programming Languages (PACMPL)* 1, OOPSLA (Oct. 2017). https://doi.org/10.1145/3133894

Received 2022-11-10; accepted 2023-03-31