# Extensible Metatheory Mechanization via Family Polymorphism

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, a seemingly 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.

## 1 INTRODUCTION

There is a growing trend among programming languages researchers to use proof assistants to mechanize metatheories. However, the programmer runs into an old problem in the new setting of proof engineering: the Expression Problem (EP) [44].

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 [37] 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 conventional setting of functional programming and object-oriented (OO) programming. Modern languages, such as Scala [35], have a good supply of linguistic features that offer expressive power to solve the EP.

In contrast, proof assistants offer few linguistic solutions that address the EP. Yet, the challenge of writing extensible, type-safe code is as real, especially for metatheory mechanization. The programmer faces a tension between adding new constructors to an inductive data type (e.g., new abstract syntax) and adding new functions and theorems over the data type (e.g., new meta-theoretical results).

In the Coq proof assistant [9], 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 and encodings [13, 14, 39, 26, 20]. But they tend to require heavy lifting from the programmer to make code extensible and often lead to non-idiomatic programming styles.

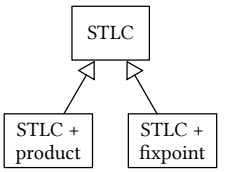
We thus set out to answer the following question: can extensible metatheory mechanization be made easier by having a proof assistant support new *linguistic features* that address the EP?

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* [8].

Language mechanisms including mixins [7], virtual classes [28, 17], virtual types [43], extensible cases [5], etc. are all based on this essential idea of inheritance. In particular, when a language mechanism allows late binding of the meaning of nested types and terms, it is said to support *family polymorphism* [16]: 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 $\lambda$-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 strong 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.
- We capture the essence of the new language mechanism formally by extending Martin–Löf type theory with facilities to express family polymorphism (Section 5). We prove foundational metatheoretical results including consistency and canonicity.
- We present case studies of using our Coq plugin to mechanize language metatheories (Section 6). 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 [34, 17, 45], because the underlying programming language is simultaneously functional, dependently typed, a logic, and an interactive tool.

**C1. Extensible inductive types vs. exhaustive induction 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.

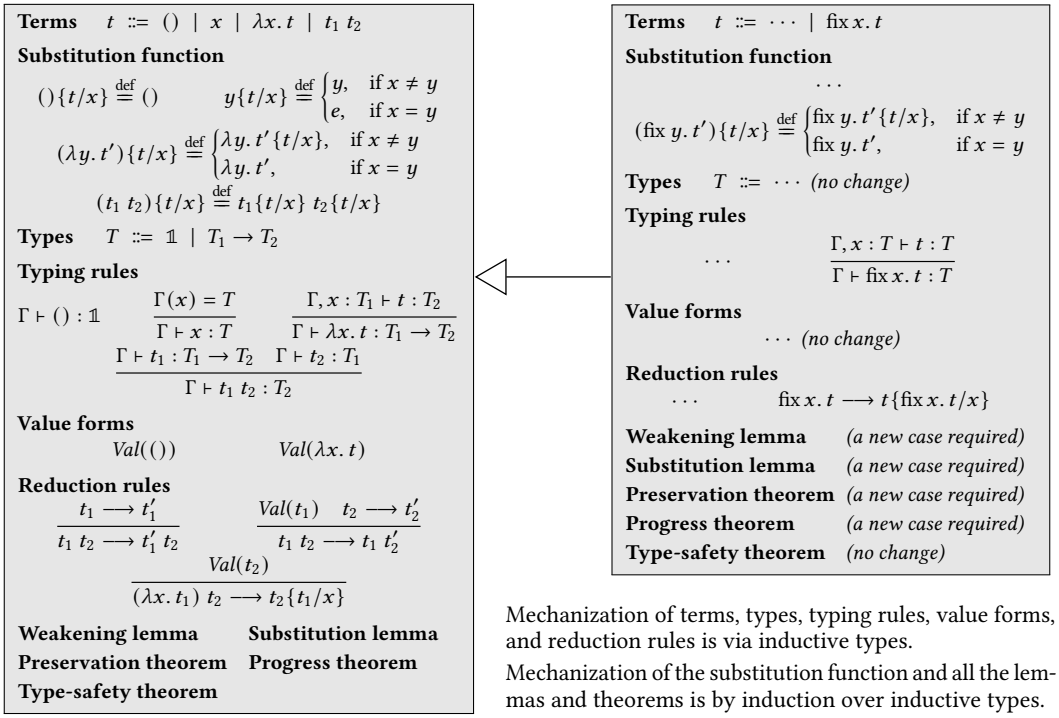



Figure 1. STLC metatheories (left) and its extension with fixpoints (right).

However, there is a tension between extensibility of inductive types and exhaustivity of induction 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: 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 equate references to the substitution function and its definition (i.e., 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

[1] The tension reflects a duality between variants and records. For record-like language constructs (e.g., classes and objects), it is safe for an extension to add more 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 potentially become non-exhaustive.

self-referential definitions [8]. 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 producing 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.

### 3.1 Extensible Inductive Types and Exhaustive Recursion/Induction

**Extending inductive types.** In family `STLCFix`, inductive type `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" t1) t2`. 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 becomes in jeopardy, however. In particular, 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.

```
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 :=
| ht_unit : ∀ G, hasty G tm_unit ty_unit
| ht_var  : …                          (* Typing rules *)
| 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 on hasty (* Weaken. lemma *)
  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 on hasty    (* Subst. lemma *)
  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 on hasty  (* Preserv. theorem *)
  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 on hasty (* Progress theorem *)
  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.

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 retroactively adding a fifth case to form the 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 substlem 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 $\beta$-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 6).

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 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`!

```
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.

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, annotation is required.

### 3.5 Composing Families as Mixins

Families can be readily reused to construct larger extensions that *mix in* [7] 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 [19]—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 the inductive type `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 are automatically generated for all 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. Rather than modifying the Coq kernel, a prototypical implementation as a plugin allows a clearly defined trusted base and improved compatibility with different versions of Coq.

The implementation 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).

As an example, consider field `env` in family `STLC`. It is translated into a top-level module named `STLC°env`. This module has a parameter called `self` 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 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`.

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.

**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 interactively: every time the programmer completes a `Case`, a case handler (e.g., `Def subst°tm_unit`) is generated and added to the module.

```
(* 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 :=(* emmited 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 computational 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 STLC°env.                    (* Include env *)

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

End STLC.
```

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

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 bahaviors 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

```
(* 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).

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 selfs 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, Axioms 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. 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. 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.

## 5 FMLTT: A CORE DEPENDENT TYPE THEORY

We contribute FMLTT, a core type theory that extends Martin-Löf dependent type theory (MLTT) [29] with facilities to express family polymorphism while maintaining consistency and canonicity. We acknowledge that the presentation herein is dense for an audience without prior exposure to MLTT, so we summarize the salient points first before delving into the technical details.

**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 prior work [45] models family polymorphism in an OO setting, but the technical details differ significantly from the prior work.)

- Linkages model families; they are like tuples composed of the 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 also features *linkage transformers* modeling the inductive construction of families from other families. They allow a linkage to be created by inheriting or overriding components of an existing linkage, or by extending an existing linkage with new components.
- Inductive types are modeled as W-types [30] 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). As is the current trend [10, 2, 21] with MLTT presentations, we use *explicit substitutions* [31, 1]: substitutions $\gamma$ and their applications (e.g., $T[\gamma]$) are part of the syntax rather than meta-operations. Variables are in the form of de Bruijn indices: $\mathtt{var}_n$ is the variable bound by the $n$-th closest enclosing binder. For example, $\lambda x.\, \lambda y.\, x$ is $\lambda(\lambda(\mathtt{var}_1))$. Substitutions are typed with the form $\Gamma \vdash \gamma : \Delta$. The idea is that applying $\gamma$ to terms valid in the context $\Delta$ yields terms valid in $\Gamma$ (TM/SUB and TY/SUB). The two main forms of substitutions are weakening (SUB/WK) and extension (SUB/EXT): $t[\mathsf{p}^n]$ introduces $n$ free variables into the context of $t$, and $t[\gamma, t']$ substitutes $t'$ for $\mathtt{var}_0$ in $t$ and then applies $\gamma$. For example, rule TM/SND states that if $t$ is a dependent pair that has type $\Sigma(A, B)$, then $\mathtt{snd}\ t$ has type $B\big[\mathsf{p}^0, \mathtt{fst}\ t\big]$, where $\mathsf{p}^0$ is the identity substitution (SUB/ID).

We use Tarski-style universes [22]: a universe $\mathbb{U}$ is inhabited by the *codes* of types, with $\mathtt{El}(t)$ decoding $t$ (rule TY/EL) and $\mathtt{c}(T)$ encoding type $T$ (rule TM/C). MLTT has an infinite hierarchy of universes; we omit universe levels in the presentation for conciseness.

A singleton type $\mathbb{S}(t)$ helps expose the definition of a term $t$ in its type (rule TM/S) [3, 41].

$$
\begin{array}{rrcl}
\text{Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, A \\
\text{Substitutions} & \gamma & ::= & \mathsf{p}^n \mid \gamma, t \mid \gamma_1 \circ \gamma_2 \\
\text{Types} & A, B, T & ::= & T[\gamma] \mid \mathbb{U} \mid \mathbb{B} \mid \bot \mid \top \mid \Pi(A, B) \mid \Sigma(A, B) \mid \mathsf{Eq}(t_1, t_2) \mid \mathbb{S}(t) \mid \mathsf{El}(t) \\
\text{Terms} & t & ::= & t[\gamma] \mid \mathsf{var}_n \mid \mathsf{c}(T) \mid () \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(t_1, t_2, t_3) \mid \lambda(t) \mid \mathsf{app}(t) \mid \\
& & & (t_1, t_2) \mid \mathtt{fst}\ t \mid \mathtt{snd}\ t \mid \mathtt{refl}(t) \mid \mathtt{J}(t_1, t_2)
\end{array}
$$

$$
\boxed{\Gamma \vdash} \quad \boxed{\Gamma \vdash \gamma : \Delta} \quad \boxed{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \Delta} \quad \boxed{\Gamma \vdash T} \quad \boxed{\Gamma \vdash T_1 \equiv T_2} \quad \boxed{\Gamma \vdash t : T} \quad \boxed{\Gamma \vdash t_1 \equiv t_2 : T}
$$

$$
\textsc{(sub/id)}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{p}^0 : \Gamma} \qquad \textsc{(sub/wk)}\ \frac{\Gamma \vdash \mathsf{p}^n : \Delta \quad \Gamma \vdash A}{\Gamma, A \vdash \mathsf{p}^{n+1} : \Delta} \qquad \textsc{(sub/ext)}\ \frac{\Gamma \vdash \gamma : \Delta \quad \Delta \vdash A \quad \Gamma \vdash t : A[\gamma]}{\Gamma \vdash \gamma, t : \Delta, A}
$$

$$
\textsc{(ty/sub)}\ \frac{\Delta \vdash T \quad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash T[\gamma]} \qquad \textsc{(tm/sub)}\ \frac{\Delta \vdash t : T \quad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash t[\gamma] : T[\gamma]}
$$

$$
\textsc{(tmeq/sub/id)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash t\left[\mathsf{p}^0\right] \equiv t : T} \qquad \textsc{(tm/var)}\ \frac{}{\Gamma, A_n, ..., A_1, A_0 \vdash \mathsf{var}_n : A_n\left[\mathsf{p}^{n+1}\right]}
$$

$$
\textsc{(tm/lam)}\ \frac{\Gamma \vdash A \quad \Gamma, A \vdash t : B}{\Gamma \vdash \lambda(t) : \Pi(A, B)} \qquad \textsc{(tm/app)}\ \frac{\Gamma \vdash t : \Pi(A, B)}{\Gamma, A \vdash \mathsf{app}(t) : B}
$$

$$
\textsc{(tm/pair)}\ \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B\left[\mathsf{p}^0, t_1\right]}{\Gamma \vdash (t_1, t_2) : \Sigma(A, B)} \qquad \textsc{(tm/snd)}\ \frac{\Gamma \vdash t : \Sigma(A, B)}{\Gamma \vdash \mathsf{snd}\ t : B\left[\mathsf{p}^0, \mathtt{fst}\ t\right]}
$$

$$
\textsc{(ty/el)}\ \frac{\Gamma \vdash t : \mathbb{U}}{\Gamma \vdash \mathsf{El}(t)} \qquad \textsc{(tm/c)}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{c}(T) : \mathbb{U}}
$$

$$
\textsc{(tmeq/c)}\ \frac{\Gamma \vdash t : \mathbb{U}}{\Gamma \vdash \mathsf{c}(\mathsf{El}(t)) \equiv t : \mathbb{U}} \qquad \textsc{(tyeq/el)}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{El}(\mathsf{c}(T)) \equiv T} \qquad \textsc{(ty/s)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbb{S}(t)}
$$

$$
\textsc{(tm/s)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash t : \mathbb{S}(t)} \qquad \textsc{(tmeq/s/eta)}\ \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \mathbb{S}(t_1)}{\Gamma \vdash t_1 \equiv t_2 : T}
$$

Figure 6. Syntax and selected typing rules of MLTT.

**Introducing and eliminating inductive types.** W-types [30] are a succinct way to model inductive types in MLTT. Together with the identity type $\mathsf{Eq}(\bullet, \bullet)$, they can express a whole host of inductive types [23]. A constructor is given by a pair of types $\Gamma \vdash A$ and $\Gamma, A \vdash B$ (rule WSIG/ADD): given two arguments $\Gamma \vdash t_1 : A$ and $\Gamma, B\left[\mathsf{p}^0, t_1\right] \vdash t_2 : T$, $\mathsf{Wsup}(\tau, t_1, t_2)$ constructs a term of type $T$ (rule TM/WSUP). As the rules suggest, our formulation additionally supports *W-type signatures*. A well formed W-type signature $\Gamma \vdash \tau\ \mathsf{WSig}^n$ is composed of $n$ pairs of types, each of which represents a constructor (rules WSIG/EMPTY and WSIG/ADD), with $\mathsf{w}\pi_1^i(\tau)$ and $\mathsf{w}\pi_2^i(\tau)$ giving the $i$-th pair in $\tau$, and $\mathsf{W}(\tau)$ giving the code of the W-type (TM/W).

W-types are eliminated with the form $\mathsf{Wrec}(\tau, \ell, t)$, where $t$ is of a W-type $\mathsf{El}(\mathsf{W}(\tau))$, and $\ell$ is essentially an $n$-tuple of case handlers for the $n$ constructors in $\tau$ (TM/WREC). Each case handler has a type of form $\mathsf{CaseTy}(A, B, T)$, where $T$ is the motive of the recursion (TYEQ/CASETY); for simplicity, we model non-dependent motives for recursion. The collection of case handlers $\ell$ encodes those defined and inherited by an `FRecursion` command in our plugin. We choose to type it with a linkage type $\mathbb{L}(\mathsf{RecSig}(\tau, T))$ to avoid introducing non-dependent $n$-tuples, which linkages generalize.

Rules TM/WSUP and TM/WREC require access to $\tau$: the W-type is exhaustively generated by its constructors, and its elimination must exhaustively handle all the constructors in its signature.

In contrast, $\tau$ should be hidden from the typing context of any term that does not invoke $\mathsf{Wsup}(\tau, \bullet, \bullet)$ or $\mathsf{Wrec}(\tau, \bullet, \bullet)$, so that the term can be reused—without being rechecked—for a different W-type signature $\tau'$ that extends $\tau$ 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 $\tau'$. Such abstraction required by family polymorphism is supported via linkages, which we discuss next.

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

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

$$\textsc{(tm/w)}\ \frac{\Gamma \vdash \tau\ \mathtt{WSig}^n}{\Gamma \vdash \mathtt{W}(\tau) : \mathbb{U}} \qquad \textsc{(wsig/empty)}\ \frac{}{\Gamma \vdash \mathtt{w}^\bullet\ \mathtt{WSig}^0} \qquad \textsc{(wsig/add)}\ \frac{\Gamma \vdash \tau\ \mathtt{WSig}^n \quad \Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash \mathtt{w}^+(\tau, A, B)\ \mathtt{WSig}^{n+1}}$$

$$\textsc{(tm/wsup)}\ \frac{\Gamma \vdash \tau\ \mathtt{WSig}^n \quad \Gamma \vdash t_1 : \mathtt{w}\pi_1^i(\tau) \quad \Gamma, \mathtt{w}\pi_2^i(\tau)\left[\mathsf{p}^0, t_1\right] \vdash t_2 : \mathtt{El}(\mathtt{W}(\tau))}{\Gamma \vdash \mathtt{Wsup}(\tau, t_1, t_2) : \mathtt{El}(\mathtt{W}(\tau))}$$

$$\textsc{(tm/wrec)}\ \frac{\Gamma \vdash \ell : \mathbb{L}(\mathtt{RecSig}(\tau, T)) \quad \Gamma \vdash t : \mathtt{El}(\mathtt{W}(\tau))}{\Gamma \vdash \mathtt{Wrec}(\tau, \ell, t) : T} \qquad \textsc{(tyeq/casety)}\ \frac{\Gamma \vdash A \quad \Gamma, A \vdash B \quad \Gamma \vdash T}{\Gamma \vdash \mathtt{CaseTy}(A, B, R) \equiv \Pi(A, \Pi(\Pi(B, T\left[\mathsf{p}^2\right]), T\left[\mathsf{p}^2\right]))}$$

$$\textsc{(lsig/empty)}\ \frac{}{\Gamma \vdash \nu^\bullet\ \mathtt{LSig}^0} \qquad \textsc{(lsig/add)}\ \frac{\Gamma \vdash \sigma\ \mathtt{LSig}^n \quad \Gamma, A \vdash T \quad \Gamma, \mathbb{P}(\sigma) \vdash s : A\left[\mathsf{p}^1\right]}{\Gamma \vdash \nu^+(\sigma, s, T)\ \mathtt{LSig}^{n+1}}$$

$$\textsc{(l/empty)}\ \frac{}{\Gamma \vdash \mu^\bullet : \mathbb{L}(\nu^\bullet)} \qquad \textsc{(l/add)}\ \frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \quad \Gamma, A \vdash t : T \quad \Gamma, \mathbb{P}(\sigma) \vdash s : A\left[\mathsf{p}^1\right]}{\Gamma \vdash \mu^+(\ell, t) : \mathbb{L}(\nu^+(\sigma, s, T))} \qquad \textsc{(tm/inh)}\ \frac{\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash \ell : \mathbb{L}(\sigma_1)}{\Gamma \vdash \mathtt{inh}(h, \ell) : \mathbb{L}(\sigma_2)}$$

$$\textsc{(tyeq/pk/add)}\ \frac{\Gamma \vdash \sigma\ \mathtt{LSig}^n \quad \Gamma, \mathbb{P}(\sigma) \vdash s : A\left[\mathsf{p}^1\right] \quad \Gamma, A \vdash T}{\Gamma \vdash \mathbb{P}(\nu^+(\sigma, s, T)) \equiv \Sigma(\mathbb{P}(\sigma), T\left[\mathsf{p}^1, s\right])} \qquad \textsc{(tmeq/pk/add)}\ \frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \quad \Gamma, \mathbb{P}(\sigma) \vdash s : A\left[\mathsf{p}^1\right] \quad \Gamma, A \vdash t : T}{\Gamma \vdash \mathbb{P}(\mu^+(\ell, t)) \equiv (\mathbb{P}(\ell), t\left[\mathsf{p}^1, s\right]\left[\mathsf{p}^0, \mathbb{P}(\ell)\right]) : \mathbb{P}(\nu^+(\sigma, s, T))}$$

$$\textsc{(tmeq/ov/beta)}\ \frac{\begin{array}{c}\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash \ell : \mathbb{L}(\sigma_1) \\ \Gamma, A_1 \vdash t_1 : T_1 \quad \Gamma, \mathbb{P}(\sigma_1) \vdash s_1 : A_1\left[\mathsf{p}^1\right] \quad \Gamma, A_2 \vdash t_2 : T_2 \quad \Gamma, \mathbb{P}(\sigma_2) \vdash s_2 : A_2\left[\mathsf{p}^1\right]\end{array}}{\Gamma \vdash \mathtt{inh}(\mathtt{Override}(h, t_2), \mu^+(\ell, t_1)) \equiv \mu^+(\mathtt{inh}(h, \ell), t_2) : \mathbb{L}(\nu^+(\sigma_2, s_2, T_2))}$$

Figure 7. Syntax and selected typing rules of FMLTT.

**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.

Well-formedness judgments of *linkage signatures* have form $\Gamma \vdash \sigma\ \mathtt{LSig}^n$. As rules LSIG/EMPTY and LSIG/ADD show, a linkage signature is a list of $n$ types. Well-formedness judgments of *linkages* have form $\Gamma \vdash \ell : \mathbb{L}(\sigma)$, where $\mathbb{L}(\sigma)$ is a type formed by $\sigma$. As rules L/EMPTY and L/ADD show, a linkage is a list of $n$ terms, each representing a field of the family modeled by the linkage.

In rule L/ADD, the second premise $\Gamma, A \vdash t : T$ 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 $t$. Crucially, the premise $\Gamma, A \vdash t : T$ 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 $\ell'$ that overrides fields in $t$'s context (and thus models a derived family) can reuse $t$—without retyping it—by first projecting $t$ from $\mu^+(\ell, t)$ and then appending it to $\ell'$.

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 $\mathtt{W}(\tau)$ of a

| | |
|---|---|
| `Family STLC.` | $\sigma_0 \coloneqq v^\bullet$ $\quad \ell_0 \coloneqq \mu^\bullet$ $\quad \cdot \vdash \ell_0 : \mathbb{L}(\sigma_0)$ |
| `FInductive tm :=` | $\sigma_1 \coloneqq v^+(\sigma_0, s_1, T_1)$ $\quad \ell_1 \coloneqq \mu^+(\ell_0, t_1)$ $\quad \cdot \vdash \ell_1 : \mathbb{L}(\sigma_1)$ $\quad \mathbb{P}(\sigma_0) \vdash s_1 : A_1[\mathsf{p}^1]$<br>$self : [] \vdash \mathtt{W}(\tau_{\mathtt{tm}}) : \mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))$ |
| `\| tm_unit : tm` | $\sigma_2 \coloneqq v^+(\sigma_1, s_2, T_2)$ $\quad \ell_2 \coloneqq \mu^+(\ell_1, t_2)$ $\quad \cdot \vdash \ell_2 : \mathbb{L}(\sigma_2)$ $\quad \mathbb{P}(\sigma_1) \vdash s_2 : A_2[\mathsf{p}^1]$<br>$self : [\mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))] \vdash \mathtt{Wsup}(\tau_{\mathtt{tm}}, \top, \bot) : \mathtt{El}(self.\mathtt{tm})$ |
| `\| … \| … \| ….` | … |
| `FRecursion subst ….` | $\mathtt{W}(\tau_{\mathtt{tm}})$ *is abstracted into* $\mathbb{U}$ *in the typing context (i.e.,* $A_6$*) of the case handler below:* |
| `Case tm_unit := ….` | $\sigma_6 \coloneqq v^+(\sigma_5, s_6, T_6)$ $\quad \ell_6 \coloneqq \mu^+(\ell_5, t_6)$ $\quad \cdot \vdash \ell_6 : \mathbb{L}(\sigma_6)$ $\quad \mathbb{P}(\sigma_5) \vdash s_6 : A_6[\mathsf{p}^1]$<br>$self : [\mathtt{tm} : \mathbb{U},\ \mathtt{tm\_unit} : \mathtt{El}(\mathtt{tm}),\ \ldots] \vdash \ldots : \mathtt{CaseTy}(\top, \bot, \mathtt{El}(self.\mathtt{tm}) \to id \to \mathtt{El}(self.\mathtt{tm}))$ |
| `Case … Case … Case …` | … |
| `End subst.` | $\sigma_{10} \coloneqq v^+(\sigma_9, s_{10}, T_{10})$ $\quad \ell_{10} \coloneqq \mu^+(\ell_9, t_{10})$ $\quad \cdot \vdash \ell_{10} : \mathbb{L}(\sigma_{10})$ $\quad \mathbb{P}(\sigma_9) \vdash s_{10} : A_{10}[\mathsf{p}^1]$<br>$self : \begin{bmatrix} \mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}})),\ \mathtt{tm\_unit} : \mathtt{El}(\mathtt{tm}),\ \ldots, \\ \mathtt{subst\_tm\_unit} : \mathtt{CaseTy}(\top, \bot, \ldots),\ \ldots \end{bmatrix} \vdash \lambda t.\, \mathtt{Wrec}(\tau_{\mathtt{tm}}, \ldots, t) : \begin{matrix} \mathtt{El}(self.\mathtt{tm}) \to \mathtt{El}(self.\mathtt{tm}) \\ \to id \to \mathtt{El}(self.\mathtt{tm}) \end{matrix}$ |
| … | … |
| `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 $self : A_i \vdash t_i : T_i$ types a field. The type $A_i$ controls how the typing of field $t_i$ sees the types of the fields prior to $t_i$.

W-type has to expose different types to different fields that come after it. Later fields that invoke $\mathtt{Wsup}(\tau, \bullet, \bullet)$ or $\mathtt{Wrec}(\tau, \bullet, \bullet)$ should see the concrete signature $\tau$, as rules TM/WSUP and TM/WREC stipulate. By contrast, $\tau$ should be hidden from all other fields, so that they can be reused in a different context where the W-type signature $\tau$ is replaced by an extended one $\tau'$.

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. $\ell_i$ is the linkage that adds the $i$-th field $t_i$ with the typing $A_i \vdash t_i : T_i$. For readability, we feel free to use a notation with explicit names in this typing judgment: we name the context as *self* (cf. Figure 4) and feel free to write its type $A_i$, which is a dependent tuple type, as a dependent record type with labels.

- The first field $t_1$ is defined as $\mathtt{W}(\tau_{\mathtt{tm}})$ and given the singleton type $\mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))$, where $\tau_{\mathtt{tm}}$ is the W-type signature for `tm`. This typing is recorded by $\sigma_1$ and is thus available in all $\sigma_i$'s.
- The next four fields model the four constructors of `tm`. Constructor `tm_unit` is modeled as $\mathtt{Wsup}(\tau_{\mathtt{tm}}, \top, \bot)$ and typed with $\mathtt{El}(self.\mathtt{tm})$, where *self* stands for the typing context containing the first field `tm`. The W-type signature $\tau_{\mathtt{tm}}$ is exposed in this typing context; that is, `tm` has type $\mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))$. So $\mathtt{El}(self.\mathtt{tm})$ and $\mathtt{El}(\mathtt{W}(\tau_{\mathtt{tm}}))$ can be equated, as required by rule TM/WSUP.
- Similarly, $\tau_{\mathtt{tm}}$ is exposed in the typing context of $t_{10}$, which models the recursive function `subst` by invoking the recursor $\mathtt{Wrec}(\tau_{\mathtt{tm}}, \bullet, \bullet)$. 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, $\tau_{\mathtt{tm}}$ is hidden from the typing of all other fields. Their typing should depend on the knowledge that `tm` has type $\mathbb{U}$, rather than $\mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))$, so that they can be reused in a context where `tm` is defined as $\mathtt{W}(\tau'_{\mathtt{tm}})$, where $\tau'_{\mathtt{tm}}$ extends $\tau_{\mathtt{tm}}$ 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 $\mathtt{tm} : \mathbb{U}$—so the case handlers can be reused by a linkage modeling `STLCFix`.

The third premise $\Gamma, \mathbb{P}(\sigma) \vdash s : A[\mathsf{p}^1]$ in rules L/ADD and LSIG/ADD is responsible for hiding W-type signatures. Here, $\mathbb{P}(\sigma)$ is a dependent tuple type (rule TYEQ/PK/ADD) that packages the types of all fields preceding the current field $t$, and $s$ is the term turning the tuple $\mathbb{P}(\ell)$ of type $\mathbb{P}(\sigma)$ (rule TMEQ/PK/ADD) into a new tuple of type $A$ that hides W-type signatures behind $\mathbb{U}$, if necessary.

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, $s$ is $\mathtt{var}_0$ and $A$ is $\mathbb{P}(\sigma)$. In Figure 8, $s_1$, $s_2$, and $s_{10}$ are $\mathtt{var}_0$, and $s_6$ is from $[\mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}})),\ \mathtt{tm\_unit} : \mathtt{El}(\mathtt{tm}),\ \ldots]$ to $[\mathtt{tm} : \mathbb{U},\ \mathtt{tm\_unit} : \mathtt{El}(\mathtt{tm}),\ \ldots]$.

When a family is concluded (e.g., `End STLC`), a linkage $\ell$ containing all the fields is available (e.g., $\ell_{53}$ in Figure 8). Fields of the family can then be accessed by projecting them out of the tuple $\mathbb{P}(\ell)$.

**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 $\mathtt{Identity}$, $\mathtt{Extend}(h, t)$, $\mathtt{Override}(h, t)$, $\mathtt{Inherit}(h)$, etc. Figure 7 shows the $\beta$-rule of an exemplary transformer, TMEQ/OV/BETA. It states that applying the transformer $\mathtt{Override}(h, t_2)$ to a linkage of form $\mu^+(\ell, t_1)$ overrides the linkage's last field $t_1$ with $t_2$. For instance, the construction below shows that $\mathtt{Override}(\mathtt{Identity}, \mathtt{W}(\tau'_{\mathtt{tm}}))$ is used as the first step in creating a linkage transformer modeling a derived family that overrides $\tau_{\mathtt{tm}}$ with an extended signature $\tau'_{\mathtt{tm}}$:

```
Family STLCFix extends STLC.
FInductive tm += ...
```

$h_0 \coloneqq \mathtt{Identity} \qquad \cdot \vdash h_0 : \nu^\bullet \twoheadrightarrow \nu^\bullet$

$h_1 \coloneqq \mathtt{Override}(h_0, \mathtt{W}(\tau'_{\mathtt{tm}})) \quad \cdot \vdash h_1 : \nu^+(\nu^\bullet, s_1, \mathbb{S}(\mathtt{W}(\tau_{\mathtt{tm}}))) \twoheadrightarrow \nu^+(\nu^\bullet, s'_1, \mathbb{S}(\mathtt{W}(\tau'_{\mathtt{tm}})))$

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 terms of a metalanguage supporting quotient inductive-inductive types (QIITs) [33, 2]. QIITs support equality constructors, which facilitate the expression of conversion rules (e.g., TMEQ/SUB/ID). The formalization is available as a supplemental file in Agda syntax, though it is not checked by Agda because Agda does not support QIITs natively. An effort to mechanize the formalization is underway, using Agda's `REWRITE` pragma to replace the limited uses of equality reflection. We also typeset a more complete set of rules than the space allows and present it in a supplemental appendix.

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

Theorem 5.1 (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.

Theorem 5.1 is a consequence of a metacircular interpretation of all the well-formedness rules, which, in particular, interprets $\bot$ in FMLTT as an uninhabited type in the metalanguage. The construction of this interpretation follows the standard model of Altenkirch and Kaposi [2], extending it to handle linkages. A linkage $\mu^+(\ell, t)$ can be interpreted into a non-dependent pair where the second component is a dependent function (modeling late binding), as rule L/ADD indicates. We refer the reader to a supplemental file in Agda syntax for the complete proof.

**Canonicity.** A second property we prove is canonicity.

Theorem 5.2 (Canonicity). *If $\cdot \vdash t : \mathbb{B}$, then either $\cdot \vdash t \equiv \mathtt{tt} : \mathbb{B}$ or $\cdot \vdash t \equiv \mathtt{ff} : \mathbb{B}$.*

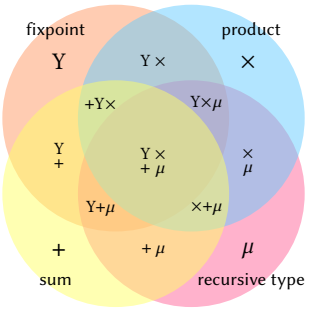
This canonicity theorem says that every closed term of the ground type $\mathbb{B}$ is convertible to one of the canonical forms $\mathtt{tt}$ and $\mathtt{ff}$. When the canonicity theorem is proved in a constructive metalogic, its proof amounts to a normalization function for closed terms of the ground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

We prove Theorem 5.2 by constructing a logical-relations model for the well-formedness rules, following prior approaches [10, 40, 25]. In particular, a closed, well formed type $\cdot \vdash T$ is interpreted into a logical predicate on closed terms: the predicate includes all closed, "reducible" terms of type $T$. The proof is available as a supplemental file in Agda syntax.

## 6 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 [36]. The base STLC family consists of about 400 LOC. Lines of code in each of the four derived families (Y, ×, +, and $\mu$ in the Venn diagram) vary from 100 to 250, largely depending on how many constructors they add to the inductive types. The linguistic nature of our approach allows us to retain a programming style similar to the original proofs in Software Foundations.

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.




Composing features can lead to *feature interactions* [4]: features working correctly in isolation may require coordination when composed. For example, composing `STLCIsorec` and `STLCProd` (Figure 3) creates a need to extend `tysubst` to handle `ty_prod`, which the typechecker enforces.

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.

The current plugin implementation does not yet support mutually inductive types or mutual induction, which are useful for modeling languages much more complex than STLCs. However, we believe accommodating mutually inductive types does not pose theoretical challenges.

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

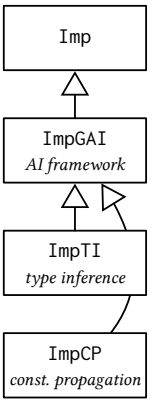
The code is organized into four families. A base family, `Imp`, defines via `FInductive` the abstract syntax of a while-language with pure expressions and impure statements. The semantics is given by an interpreter defined as a CEK-style abstract machine [18] and parameterized by a fuel value. Family `Imp` defines the interpreter via `FRecursion`. It contains ~200 LOC.

A second family, `ImpGAI`, extends `Imp` and uses ~550 LOC. 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 abstraction 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 abstraction 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 other two families then extend `ImpGAI`. Family `ImpTI` (~200 LOC) is an abstract interpreter that does type inference [11]. 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 retains the computation ability of the host proof assistant; the two abstract interpreters in `ImpTI` and `ImpCP` are ready for program extraction to OCaml. We put them to test over queries.

We note that our plugin cannot yet allow the expression of two derived abstract interpreters nested within the same family. Work on nested inheritance [34, 45] points to a direction to further increase the expressive power of our language design.

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 them.

## 7 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. [13] engineer product lines of theorems and proofs built from feature modules. Feature composition is manual, which seems to have motivated later approaches based on DTC.

The design pattern *data types à la carte* (DTC) introduced for Haskell by Swierstra [42] is a source of inspiration for many approaches [14, 15, 39, 26, 20]. 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. [14] introduce *meta-theory 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 [39] adapt DTC to Agda by considering a restricted class of functors that admit least fixed points. Keuchel and Schrijvers [26] 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 [14] has been applied to implementing composable program adverbs [27] and been adapted to support unanticipated feature extensions that require type changes [15].

On the other hand, the extralinguistic nature of the approaches tend 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 is less readable and the resulting 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 [20] 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 [6] 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 family polymorphism in our approach allows names to be late bound.

Mulhern [32] 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 [24] 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 [38] is a Coq plug-in 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.

**Solutions to the expression problem** abound. Almost all involve some form of either explicit or implicit parameterization as extensibility hooks. Our approach is the first that applies family polymorphism [16], an idea from the context of OO languages, to the context of mechanized proofs.

Blume et al. [5] address the expression problem for a core subset of Standard ML by combining explicitly coded open recursion with a mechanism 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.

## 8 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. Generalizing the mechanism to support nested family polymorphism will make extensibility scale to sizable proof engineering efforts and thus make exciting directions for future work.

## REFERENCES


[1] 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

[2] 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

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

[4] 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

[5] 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

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

[7] 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

[8] 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)*.

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

[10] 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

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

[12] 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

[13] 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

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

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

[16] Erik Ernst. 2001. Family polymorphism. In *European Conf. on Object-Oriented Programming (ECOOP)*.

[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

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

[19] 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

[20] 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

[21] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. *Proc. of the ACM on Programming Languages (PACMPL)* 3, ICFP (2019).

[22] Martin Hofmann. 1997. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory.*

[23] 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

[24] 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

[25] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for type theory. In *Int'l Conf. on Formal Structures for Computation and Deduction (FSCD)*.

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

[27] 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

[28] 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

[29] Per Martin-Löf. 1982. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*. Vol. 104.

[30] 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)*.

[31] Per Martin-Löf. 1992. Substitution calculus. Notes from a lecture given in Göteborg.

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

[33] Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-inductive definitions. In *Int'l Workshop on Computer Science Logic*.

[34] 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

[35] 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

[36] 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.

[37] John C. Reynolds. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages*.

[38] 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

[39] 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

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

[41] Christopher Allan Stone. 2000. *Singleton kinds and singleton types*. Ph.D. Dissertation. Carnegie Mellon University.

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

[43] Kresten Krab Thorup. 1997. Genericity in Java with virtual types. In *European Conf. on Object-Oriented Programming (ECOOP)*.

[44] 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.

[45] 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