Technical Report CS-2025-02, School of Computer Science, University of Waterloo

Certified Compilers à la Carte (Extended Version)

OGHENEVWOGAGA EBRESAFE, University of Waterloo, Canada
IAN ZHAO, University of Waterloo, Canada
ENDE JIN, University of Waterloo, Canada
ARTHUR BRIGHT, University of Waterloo, Canada
CHARLES JIAN, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

Certified compilers are complex software systems. Like other large systems, they demand modular, extensible designs. While there has been progress in extensible metatheory mechanization, scaling extensibility and reuse to meet the demands of full compiler verification remains a major challenge.

We respond to this challenge by introducing novel expressive power to a proof language. Our language design equips the Rocq prover with an extensibility mechanism inspired by the object-oriented ideas of late binding, mixin composition, and family polymorphism. We implement our design as a plugin for Rocq, called Rocqet. We identify strategies for using Rocqet's new expressive power to modularize the monolithic design of large certified developments as complex as the CompCert compiler. The payoff is a high degree of modularity and reuse in the formalization of intermediate languages, ISAs, compiler transformations, and compiler extensions, with the ability to compose these reusable components—certified compilers *à la carte*. We report significantly improved proof-compilation performance compared to earlier work on extensible metatheory mechanization. We also report good performance of the extracted compiler.

Note. This technical report is an extended version of the following paper [16]:

> Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. Certified Compilers à la Carte. *Proc. ACM Program. Lang.* 9, PLDI, Article 162 (June 2025). https://doi.org/101145/3729261

1 Introduction

Compiler bugs threaten software security and reliability; they invalidate application-level guarantees and complicate the diagnosis of application bugs. Certified compilers, like CompCert [36] and CakeML [58], eliminate entire classes of compiler vulnerabilities, securing a critical part of the software stack with mathematically rigorous proofs.

Compilers are complex systems, however. Their construction calls for modular, extensible approaches that promote code reuse and compositional rather than monolithic designs [47, 44, 53, 30, 9, 48, 34]. The need for extensibility and composability is even more pressing in the case of certified compilers, where the cost of developing, maintaining, and extending mechanized proofs can be prohibitively high. Two open challenges have been identified.

Modularization of compiler extensions. This challenge echoes the expression problem [61], but in the novel setting of certified programming using proof assistants. The goal is ambitious: to support new language features by modularly extending verified compilers, to easily assemble new verified compilers by mixing and matching these verified extensions, and to do all this without rechecking the proofs of already verified components. The importance of this challenge is well recognized; for example, it was highlighted in a keynote at last year's PLDI [41].

Modularization of code representations and compiler transformations. This challenge concerns the intermediate representations (IRs) within a certified compiler. Both CompCert and CakeML are multi-pass compilers involving a series of IRs. Some IRs and passes are slight variations of one another, yet they are defined and verified with nearly identical, often copy-pasted text. This lack of modularity has been noted [12, 33]. It creates tedium, obstructs changes, blurs the distinct purposes of individual IRs and passes—yet it remains largely unresolved.

These challenges persist, largely because the compiler-and-proof engineer lacks effective means to organize programs in an *extensible*, *composable* way that *scales* to large certified components.

- Extensibility refers to minimizing hard links and maximizing extensibility hooks, so that certified components can be used in new contexts other than the original one in which they are defined.
- Composability is the ability to reuse certified components, in a mix-and-match style, without having to modify the components or recheck their proofs.
- Scalability means that extensibility hooks can describe both small components—like individual inductive types and induction proofs—and large components—like an entire family of related components—so that extension and composition are possible at both small and large scales, without causing a clutter of explicit parameters.

Prior efforts have been made [11, 12, 54, 31, 20, 27] to provide design patterns or language-level support for modularizing the development of mechanized proofs. Notably, our recent work [27] introduces an extensibility mechanism to the Rocq prover [51] (then called Coq), inspired by the object-oriented ideas of *late binding* and *family polymorphism* [18]. The resulting language design and implementation, called FPOP [27], comes close to meeting the needs of modular mechanized metatheories.

Despite all this exciting progress, the prior work is only intended to address mechanized proofs at a relatively small scale: metatheories of simply typed lambda calculi. We contend that it falls short of meeting the full demands of modularizing realistic certified optimizing compilers at the scale of CompCert.

Contributions. We identify a key generalization of FPOP that enables it to scale to large, complex compiler projects. FPOP allows the late binding of individual inductive types, as well as individual recursive definitions over inductive types, by making them polymorphic to the *family* they are nested within. Critically, FPOP does not support the late binding of larger components—in particular, families nested within a family are not polymorphic to their enclosing family and, hence, are not extensibility hooks in FPOP. This limitation constrains the scale at which verified compiler components can be defined and reused.

Hence, a first contribution of this paper is a **language design** generalizing FPOP to support *nested family polymorphism*. It allows families nested within a family to be used as extensibility hooks, which can be refined in accordance with other components of the enclosing family. Our language design also supports *traits*, which are mixins that introduce new functionalities or variations to a family. The *late binding of nested families and nested traits* is key to scaling extensibility and composability to mechanized proof developments at a much larger scale than previously attainable.

As a second contribution, we present Rocqet,[1] **an implementation of the language design** as a Rocq plugin. Like FPOP, Rocqet is implemented via a compilation to Rocq modules. Unlike FPOP, the compilation artifact of a nested family or a trait is not a "fixed point", which is closed to extension, but rather a functor parameterized by each of its enclosing family and thus can be reused in new contexts. In addition to supporting greater expressiveness than FPOP, Rocqet is also more efficient. For a major case study previously done with FPOP, Rocqet reduces proof-compilation time by a factor of more than 80.

A third contribution is an **extensible certified C compiler framework** constructed using Rocqet. Conceptually, this compiler uses the same IRs and passes as CompCert. But unlike CompCert, the compiler is structured by prioritizing extensibility and reuse.

- We verify a compiler for a base language, extend it with additional features of CompCert C, and compose these extensions to create custom compilers.

[1] The et in Rocqet is French for &, suggesting à la carte composition of families and traits—also a nod to the J& language [45].

- We use Rocqet to modularize the modeling of the RISC-V ISA, enabling different compiler extensions to customize the combination of RISC-V extensions they target.
- We use Rocqet to refactor CompCert, which currently consists of a few monolithic IRs and passes containing duplication, into finer-grained compiler transformations with sharing.

The payoff is cleaner, more modular mechanized proofs reused across IRs, passes, and compiler extensions, with good performance of the extracted compilers. Buckle up for a Rocqet ride!

2 A First Look at Rocqet in Action

This section previews how the language-design ideas in Rocqet come together to address programming challenges in the construction of extensible certified compilers. Throughout Sections 2–4, we use as a running example of a certified C compiler framework, zooming in on one compiler stage, front-end desugaring transformations on the input program, and one compiler extension, loops.[2]

The front end of the compiler performs a series of desugaring transformations to reduce the syntactic complexity of the input program—removing one-armed if statements, removing increment and decrement operators, etc. The target language of desugaring is called CoreC.

$$\text{C} \xrightarrow{\text{desugaring transformations}} \text{CoreC} \longrightarrow \text{Clight} \longrightarrow \cdots$$

Rather than a single monolithic pass for all desugaring, a more modular approach follows the *nanopass* principle [53, 30]: structuring them as a collection of many small passes, each performing a single task. These smaller passes are easier to read, define, and verify. However, this approach can lead to excessive boilerplate code repeated across nanopasses. In fact, this repetition already occurs in some passes in CompCert and CakeML, as previously noted [12, 33], even though those passes are not small enough to qualify as nanopasses.

Figures 1 and 2 show how this programming challenge can be addressed with Rocqet.

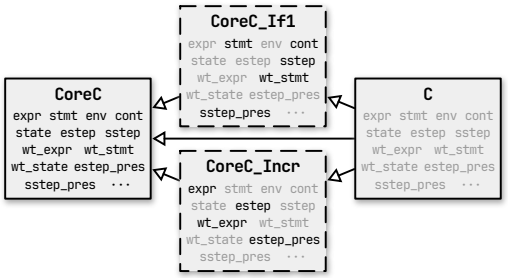
Reuse across code representations. As Figure 1 shows, The source language `C` is modeled as a `Family` composed of a shared component `CoreC` and several extensions `CoreC_If1`, `CoreC_Incr`, etc. `CoreC` is called the *base* family, `CoreC_If1` and `CoreC_Incr` are *mixins*, and `C` is the *derived* family.

The definition of the syntax and semantics is modular: each extension, defined in its own `Trait`, adds new constructors to the inductive types (`FInductive`) to model the new syntax and semantics introduced by the extension. Accordingly, metatheories proven by induction over these inductive types (`FInduction`) are also extended with new cases in their proofs.

For example, `CoreC_If1` models one-armed if statements: it adds a constructor `Sif1` to the `stmt` inductive type, and it adds constructors to `cont`, `sstep`, and `wt_stmt` to model the dynamic and static semantics of one-armed if. Now, consider `sstep_pres` in the base family `CoreC`. Its `on` clause indicates that the proof is by induction over `sstep`. Its `motive` clause indicates that it proves type preservation for statement reduction: `∀ S t S', sstep S t S' → wt_state S → wt_state S'`. Therefore, in accordance with the extension of `sstep` by `CoreC_If1`, the proof of `sstep_pres` is extended in `CoreC_If1` to handle—and only handle—the new `sstep` cases introduced by `CoreC_If1`. Traits like `CoreC_If1` make targeted refinements to their base family; they need not repeat contents of the base family.

Traits can be composed to create families. While Figure 1 shows only two traits, other extensions of `CoreC` can be similarly defined. When all the traits `CoreC_If1`, `CoreC_Incr`, etc. are composed with `CoreC` to derive the family `C`, a complete mechanization of the source language is obtained. In particular, `C.sstep_pres` and `C.estep_pres` are automatically proven metatheoretical results. For instance, running the Rocq command `Print C.sstep_pres` will display the proof term and its

[2] This example is based on CompCert. Desugaring transformations are not part of CompCert, though. We use them here for their simplicity. The idea can be applied to compiler passes in CompCert as well.



Dashed boxes represent traits (i.e., mixins).
Extensibility hooks that need no refinement are grayed out.

```
Family CoreC.                                                    Comp/CoreC.v
(* syntax *)
FInductive expr : Type :=                                   (* expression *)
| Evar : id → expr
| ... (* other expr constructors *) ....
FInductive stmt : Type :=                                   (* statement *)
| Sskip : stmt
| Sseq : stmt → stmt → stmt
| Sdo : expr → stmt
| Sif : expr → stmt → stmt → stmt
| ... (* other stmt constructors *) ....
FDefinition env : Type :=                          (* variable environment *)
  PTree.t (block * type).
FInduction cont : Type := ....                     (* evaluation context *)
FInductive state : Type := ....                    (* state of execution *)
(* small-step semantics of expressions and statements *)
FInductive estep : state → trace → state → Prop := ....
FInductive sstep : state → trace → state → Prop := ....
(* well-typedness *)
FInductive wt_expr : tyenv → expr → Prop := ....
FInductive wt_stmt : tyenv → stmt → type → Prop := ....
FInductive wt_state : tyenv → state → Prop := ....
(* type-preservation results of the small-step semantics *)
FInduction estep_pres on estep motive
λ S t S', (_ : estep S t S'), wt_state S → wt_state S'.
  ... (* handle all estep cases *) ...
End estep_pres.
FInduction sstep_pres on sstep motive
λ S t S', (_ : sstep S t S'), wt_state S → wt_state S'.
  ... (* handle all sstep cases *) ...
End sstep_pres.
...
End CoreC.
```

```
Trait CoreC_If1 extends CoreC.                              Comp/CoreC_If1.v
FInductive stmt : Type +=
| Sif1 : expr → stmt → stmt.                          (* one-armed if *)
FInductive cont : Type += ....
FInductive sstep : state → trace → state → Prop += ....
FInductive wt_stmt : tyenv → stmt → type → Prop += ....
FInduction sstep_pres.
  ... (* handle only new sstep cases *) ...
End sstep_pres.
End CoreC_If1.
```

```
Trait CoreC_Incr extends CoreC.                             Comp/CoreC_Incr.v
FInductive expr : Type +=
| Eincr : incr_or_decr → id → expr.                 (* incr/decr ops *)
FInductive estep : state → trace → state → Prop += ....
FInductive wt_expr : tyenv → expr → Prop += ....
FInduction estep_pres.
  ... (* handle only new estep cases *) ...
End estep_pres.
End CoreC_Incr.
```

```
Family C extends CoreC                                                  Comp/C.v
  using CoreC_If1, CoreC_Incr, ....
End C.                                  (* mix all CoreC extensions into CoreC *)
```

Figure 1. The source language `C` and its metatheories are mechanized by composing a shared component `CoreC` with extensions of `CoreC` (such as `CoreC_If1` and `CoreC_Incr`).

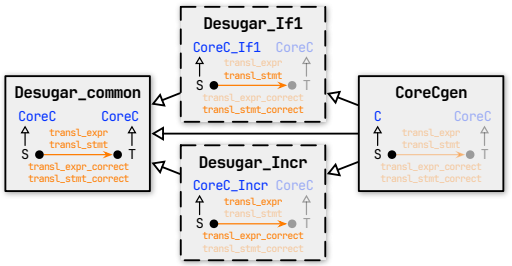
type: `∀ S t S', C.sstep S t S' → C.wt_state S → C.wt_state S'`. The proof term is synthesized by Rocqet from the proof cases in `CoreC` and those in the traits.

We will use `CoreC` as the target language for mechanizing the desugaring transformations and use the extensions as the source languages. The extensibility afforded by family polymorphism allows us to easily mechanize multiple distinct IRs that precisely capture the differences between the input and output of each desugaring transformation. The alternative would be either to use a single language as both the source and target or to duplicate the base-family logic in all extensions. The former would litter the code with extra proofs arguing that the output is indeed in the desugared form, while the latter would be brittle and hard to maintain. Neither approach is modular.

We note that our use of Rocqet up to this point does not exercise expressive power beyond FPOP [27]—we have only mechanized subject-reduction proofs thus far—although it is already putting pressure on the resources FPOP needs to compile the proofs.

Reuse across compiler transformations. As Figure 2 shows, the desugaring transformations and their correctness proofs are given by family `CoreCgen`. This family is obtained by composing a base family `Desugar_common` and several extensions `Desugar_If1`, `Desugar_Incr`, etc., each responsible for a nanopass. The shared `Desugar_common` is mostly mechanical, performing identity transformations and proving their correctness.

What is interesting here is that the source language of the translation is defined as a family `S` that is *nested* within `Desugar_common`. Unlike in FPOP, nested families and nested traits are extensibility



Dashed boxes represent traits (i.e., mixins). Extensibility hooks that need no refinement are grayed out.

```
Family Desugar_common.                                  Comp/Desugar_common.v
(* two nested families, serving as extensibility hooks *)
Family S extends CoreC. End S.          (* source language *)
Family T extends CoreC. End T.          (* target language *)
(* define the trivial translation from S to T *)
FRecursion transl_expr on S.expr motive λ_, res T.expr.
  Case S.Evar x := ret (T.Evar x).
  ... (* handle other expr cases *) ...
End transl_expr.
FRecursion transl_stmt on S.stmt motive λ_, res T.stmt.
  Case S.Sskip := ret T.Sskip.
  Case S.Sseq s1 s2 := do s1' ← transl_stmt s1;
    do s2' ← transl_stmt s2; ret (T.Sseq s1' s2').
  Case S.Sdo e s := ….  Case S.Sif e s1 s2 := ….
  ... (* handle other stmt cases *) ...
End transl_stmt.
(* prove the translation from S to T correct *)
FInduction transl_expr_correct on CoreC.estep
  motive λ S1 t S2 (_ : CoreC.estep S1 t S2),
    transl_expr_meets_spec S1 t S2.
  ... (* handle all estep cases *) ...
End transl_expr_correct.
```

```
FInduction transl_stmt_correct on CoreC.sstep
  motive λ S1 t S2 (_ : CoreC.sstep S1 t S2),
    transl_stmt_meets_spec S1 t S2.
  ... (* handle all sstep cases *) ...
End transl_stmt_correct.
End Desugar_common.
```

```
Trait Desugar_If1                                       Comp/Desugar_If1.v
  extends Desugar_common.
Trait S extends CoreC using CoreC_If1. End S.
FRecursion transl_stmt on S.stmt motive λ_, res T.stmt.
  Case S.Sif1 e s :=                    (* desugar one-armed if *)
    do e' ← transl_expr e;
    do s' ← transl_stmt s;
    ret (T.Sif e' s' T.Sskip).
End transl_stmt.
FInduction transl_stmt_correct.
  ... (* handle only new sstep cases *) ...
End transl_stmt_correct.
End Desugar_If1.
```

```
Trait Desugar_Incr                                      Comp/Desugar_Incr.v
  extends Desugar_common.
Trait S extends CoreC using CoreC_Incr. End S.
FRecursion transl_expr on S.expr motive λ_, res T.expr.
  Case S.Eincr op x := ….                (* desugar incr/decr *)
End transl_expr.
FInduction transl_expr_correct.
  ... (* handle only new estep cases *) ...
End transl_expr_correct.
End Desugar_Incr.
```

```
Family CoreCgen                                         Comp/CoreCgen.v
  extends Desugar_common
  using Desugar_If1, Desugar_Incr, ….
Family S := C.
Family T := CoreC.
End CoreCgen.
```

Figure 2. The translation from `C` to `CoreC` and its correctness proofs are mechanized by composing a shared component `Desugar_common` with extensions such as `Desugar_If1` and `Desugar_Incr`.

hooks in Rocqet. So an extension of `Desugar_common` can refine the nested family `S` to perform specialized transformations.

For example, the trait `Desugar_If1` enriches the behavior of `Desugar_common`. It refines `S` into a variant of `CoreC` that includes one-armed if, by mixing the trait `CoreC_If1` into `S`. It then defines the nanopass that compiles away one-armed if in the source language `S` (`transl_stmt`) and proves the nanopass correct (`transl_stmt_correct`). The definition of this nanopass is indeed microscopic, as it only needs to handle the new `Sif1` constructor introduced by `CoreC_If1`. The correctness proof of this nanopass is microscopic too, needing to prove only the new cases introduced to `sstep` by `CoreC_If1`. The proving can be done using tactics in much the same way as how the new cases would be proven in a complete induction over all of `sstep`'s constructors.

Importantly, there is no duplicated logic among the `Desugar_common` family and its extensions like the `Desugar_If1` and `Desugar_Incr` traits. Each trait makes targeted, coordinated refinements to the base family. Those extensibility hooks that need no refinement are automatically inherited and reused. Were it not for the ability in Rocqet to refine the nested family `S`, we would have to either resort to a single monolithic pass or duplicate the logic of the base family in each nanopass—neither is modular.

This example shows that *nested family polymorphism* allows an entire family (e.g., `S`) to be abstracted over its enclosing family, making it reusable in new contexts. Later in this section, we will see that this expressive power scales to even larger components—for example, it allows an entire compiler pass (e.g., `CoreCgen`, which itself contains nested families) to be abstracted over the compiler family it is nested within.

Fusing compiler transformations. In Figure 2, once all extensions to `Desugar_common` are composed into `CoreCgen`, the resulting pass `CoreCgen.transl_stmt`, as well as its correctness proof `CoreCgen.transl_stmt_correct`, are automatically synthesized by Rocqet.

Importantly, this generated `transl_stmt` does not apply the nanopasses in sequence—but instead *fuses* them into a single pass, which will be efficient when extracted and run! This fusion is possible because of late binding. In each nanopass trait, references to `transl_stmt` and `transl_expr` are late bound; they are polymorphic to the family they are nested within. So when the individual cases of the nanopass definition (e.g., the `Case`s of the `FRecursion` `transl_stmt` in `Desugar_common` and `Desugar_If1`) are inherited into `CoreCgen`, these references are resolved to the `transl_stmt` and `transl_expr` of the `CoreCgen` family. As a result, when `CoreCgen.transl_stmt` is applied to a statement, it performs all the desugaring transformations in a single tree traversal. The recursive functions responsible for this tree traversal are synthesized by Rocqet from all the inherited `Case`s.

This ability to fuse nanopasses seems unique to Rocqet. The nanopass framework [53, 30] implemented in Scheme and Racket does not support this kind of fusion. It requires repeated tree traversals, which leads to longer compilation times—a principal concern previously cited [30, 33].

Reuse across compiler extensions. A common recipe for an extensible compiler framework is to exercise foresight in designing a compiler for a base language and to provide extensibility hooks for anticipated feature extensions [44, 34].[3] As an exercise, suppose that loops are not part of CoreC. We want to structure loops as an extension to the base compiler `Comp` defined in Figures 1 and 2. This scenario is not unrealistic: modern domain-specific compilers, such as those based on MLIR [34], often have a minimal nucleus and only acquire features such as structured control flow when needed. In our exercise, let us extend the base compiler with three high-level loop constructs (`while`, `do-while`, and `for`, as in CompCert) and translate them to a lower-level construct `loop` s_1 s_2 in Clight.

We can define this extension as a new trait: `Trait Comp_Loops extends Comp`. The content of this trait is shown in the right column of Figure 3. It extends the base compiler `Comp` shown in the left column (part of the definition of `Comp` has been shown in Figures 1 and 2).[4] Nested family polymorphism allows all the nested components of `Comp`, including the IRs and compiler transformations, to be automatically inherited into and immediately reused by `Comp_Loops`. The trait `Comp_Loops` only needs to refine the nested families `CoreC`, `Desugar_common`, `Clight`, and `Clightgen`, utilizing the new expressive power in Rocqet. It makes coordinated changes to these extensibility hooks to accommodate the loop constructs: (1) it extends the inductive types in `CoreC` to model the three high-level loop constructs, (2) it extends the inductively defined `transl_stmt` and `transl_stmt_correct` in `Desugar_common` with new cases concerning loops, (3) it extends the inductive types in `Clight` to model the lower-level loop construct, and (4) it extends the inductively defined `lower_stmt` and `lower_stmt_correct` in `Clightgen` with new cases concerning the lowering of loops. Importantly, there is no need to repeat, change, or recheck the contents of the base `Comp` family.

`Comp_Loops` can be composed with other compiler extensions, in a mix-and-match style, to create new verified compilers, like `CompX`:

```
Family CompX extends Comp using Comp_Loops, ... (* other extensions *) ... . End CompX.
```

This composed compiler is not only modular but also efficient, because of fusion. The lowering transformation of `CompX` does not apply the lowering defined in `Comp.Clightgen` and that in

[3] Crafting a good base compiler does require foresight on possible extensions. Rocqet does not magically provide this foresight, but it will streamline the engineering of the extensible compiler framework, once a suitable design is identified.
[4] The definition of a family or trait can span multiple source files (e.g., `Comp` and `Comp_Loops`) or be contained in a single file (e.g., `CoreC`). It is also possible for a single file to contain multiple families.

```
Family Comp.
```

```
Family CoreC ...                                    Comp/CoreC.v
Trait CoreC_If1 ...                                 Comp/CoreC_If1.v
Trait CoreC_Incr ...                                Comp/CoreC_Incr.v
Family C ...                                        Comp/C.v
(* these nested families and traits have been shown in Figure 1 *)
```

```
Family Desugar_common ...                           Comp/Desugar_common.v
Trait Desugar_If1 ...                               Comp/Desugar_If1.v
Trait Desugar_Incr ...                              Comp/Desugar_Incr.v
Family CoreCgen ...                                 Comp/CoreCgen.v
(* these nested families and traits have been shown in Figure 2 *)
```

```
Family Clight.                                      Comp/Clight.v
... (* Clight is lower-level than CoreC: no side effects in expr. *)
End Clight.
```

```
Family Clightgen.                                   Comp/Clightgen.v
(* define the lowering transformation *)
FRecursion lower_expr on CoreC.expr motive
    λ(_ : CoreC.expr), res (Clight.stmt * Clight.expr).
  ... (* pull side effects out of expressions *) ...
End lower_expr.
FRecursion lower_stmt on CoreC.stmt motive
    λ(_ : CoreC.stmt), res Clight.stmt.
  ... (* map CoreC statements to Clight statements *) ...
End lower_stmt.
(* prove the lowering pass correct *)
FInduction lower_expr_correct on CoreC.estep
  motive λ S1 t S2 (_ : CoreC.estep S1 t S2),
    lower_expr_meets_spec S1 t S2.
  ... (* handle all estep cases *) ...
End lower_expr_correct.
FInduction lower_stmt_correct on CoreC.sstep
  motive λ S1 t S2 (_ : CoreC.sstep S1 t S2),
    lower_stmt_meets_spec S1 t S2.
  ... (* handle all sstep cases *) ...
End lower_stmt_correct.
End Clightgen.
(* compose two passes *)
FDefinition c_to_clight : C.stmt → Clight.stmt :=
  Clightgen.lower_stmt ∘ CoreCgen.transl_stmt.
```

```
Trait Comp_Loops extends Comp.
```

```
Trait CoreC.                                        Comp_Loops/CoreC.v
FInductive stmt : Type +=
| Swhile : expr → stmt → stmt
| Sfor : stmt → expr → stmt → stmt → stmt
| Sdowhile : stmt → expr → stmt.
FInductive cont : Type += ....
FInductive sstep : state → trace → state → Prop += ....
FInductive wt_stmt : tyenv → stmt → type → Prop += ....
FInduction sstep_pres on sstep motive ....
  ... (* handle only new sstep cases *) ...
End sstep_pres.
End CoreC.
```

```
Trait Desugar_common.                       Comp_Loops/Desugar_common.v
FRecursion transl_stmt on S.stmt motive λ_, res T.stmt.
  ... (* handle only new stmt cases *) ...
End transl_stmt.
FInduction transl_stmt_correct on S.sstep motive ....
  ... (* handle only new sstep cases *) ...
End transl_stmt_correct.
End Desugar_common.
```

```
Trait Clight.                                       Comp_Loops/Clight.v
FInductive stmt : Type += | Sloop : stmt → stmt → stmt.
...
End Clight.
```

```
Trait Clightgen.                                    Comp_Loops/Clightgen.v
FDefinition make_if (e : CoreC.expr)
  (s1 s2 : Clight.stmt) : Clight.stmt := ....
FRecursion lower_stmt on CoreC.stmt motive
    λ(_ : CoreC.stmt), res Clight.stmt.
  Case CoreC.Swhile e s := ....
  Case CoreC.Sfor s1 e s2 s3 := ....
  Case CoreC.Sdowhile s e :=
    do s' ← make_if e Clight.Sskip Clight.Sbreak;
    do ts1 ← lower_stmt s; ret (Clight.Sloop ts1 s')
End lower_stmt.
FInduction lower_stmt_correct on CoreC.sstep motive ....
  ... (* handle only new sstep cases *) ...
End lower_stmt_correct.
End Clightgen.
```

Figure 3. Extending a base compiler to support loops.

`Comp_Loops.Clightgen` in sequence. Instead, Rocqet synthesizes recursive functions that fuse the lowering of loops and other constructs into a single pass. For instance, upon the command `Extraction CompX.Clightgen.lower_stmt`, OCaml code is generated that uses a single tree traversal to pull side effects out of expressions *and* lower loops.

Reuse along multiple dimensions. As we have seen, Rocqet allows for the reuse of mechanized proofs along multiple dimensions: code representations, compiler transformations, and compiler extensions. These dimensions arise at different levels of nesting within a large-scale compiler project. Supporting reuse across *all* these dimensions arguably presents greater challenges than what the expression problem [61] captures: even reuse along a *single* dimension already requires reconciling two axes of extensibility—extending inductive types with new constructors, and extending inductively defined functions and proofs with new cases. FPOP is designed as a solution to the expression problem in the setting of certified programming. If we were to use it for modularizing a verified compiler, we would be forced to choose a single dimension of reuse and, therefore, unable to address the two challenges outlined in Section 1. By supporting *nested* family polymorphism, Rocqet addresses the needs of modularizing large-scale, complex certified developments.

3 Rocqet: Language Design

Having seen an example of Rocqet in action, we now examine the design of the language abstractions, which are responsible for Rocqet's greater expressive power compared to FPOP.

A key generalization that Rocqet makes over FPOP is that it supports *nested family polymorphism* [43, 45, 62, 32]. This new expressive power has two implications:

(a) Families and traits are extensibility hooks *themselves*.
(b) Software components are polymorphic to *every* family they are nested within.

For example, consider the recursive function `transl_stmt` (Figure 2), defined via the `FRecursion` command by induction. The `on` clause indicates that the recursive function is defined by induction over `S.stmt`, the `stmt` inductive type that `S` inherits from `CoreC`. It is first defined in `Desugar_common`. The context in which this `transl_stmt` is defined is unaware of extensions to `stmt` such as one-armed if statements, because `S` is simply defined to extend `CoreC`—nothing more. So `transl_stmt` is only required to handle cases such as `Sskip` and `Sseq` that are defined in `Comp.CoreC` (Figure 1).

As Rocqet allows nested families to be extensibility hooks themselves (a), references to the nested family `S` are late bound—that is, their meanings depend on the enclosing family. So when this `transl_stmt` is inherited into `Desugar_If1`, although it is still defined by induction on `S.stmt`, the context is now aware of the extension to `stmt` by `CoreC_If1`, because `S` now has a different, refined meaning—its behavior is refined by `CoreC_If1`. Hence, for exhaustivity of induction, Rocqet requires `transl_stmt` to be extended with a case that handles the `Sif1` constructor introduced by `CoreC_If1`.

Rocqet allows software components to be polymorphic to every enclosing family (b). So references to `S` is polymorphic, not only to the immediately enclosing family containing the desugaring transformations, but also to the outer family containing the entire compiler. Consider the `transl_stmt` in `Comp_Loops/Desugar_common.v`. Its `on` clause still refers to `S.stmt`, and `S` is still defined to extend `CoreC`. But since the outer family `Comp_Loops` refines `CoreC` by adding three new constructors to `stmt`, `transl_stmt` must be extended to handle these new cases.

Refinement. Nested software components are extensibility hooks that can be *refined* (aka *further-bound* [35]) when any of its enclosing families is extended.

- `FInductive` definitions can be refined by adding new constructors.

  Compared to FPOP, Rocqet supports mutual `FInductive` definitions and, by consequence, mutual `FRecursion` and mutual `FInduction` that operate by induction over mutually inductive types.

  In addition, Rocqet allows a singly inductive type to be refined into a mutually inductive type. This expressive power finds use in defining a compiler extension `Comp_Switch` supporting switch statements. For example, the `stmt` inductive type in `CoreC` is refined with a new constructor `Sswitch` and also with a new mutually inductive type `lbl_stmts` modeling the cases of a switch statement.

```
(* Refine the CoreC family in the Comp_Switch extension *)
FInductive stmt : Type +=      Comp_Switch/CoreC.v
| Sswitch : expr → lbl_stmts → stmt
with lbl_stmts : Type :=
| LSnil: lbl_stmts
| LScons: option Z → stmt → lbl_stmts → lbl_stmts.
```

```
FRecursion find_label on stmt motive
  λ (_ : stmt), label → cont → option (stmt * cont)
with find_label_ls on lbl_stmts motive
  λ (_ : lbl_stmts), label → cont → option (stmt * cont).
...
End find_label with find_label_ls.
```

  Accordingly, a `find_labels` function is made mutually recursive with another `find_label_ls` function, by mutual induction over `stmt` and `lbl_stmts`.

- `FRecursion` and `FInduction` definitions can be refined by adding new cases in accordance with the refinement of the inductive types they are defined over.
- `Family` and `Trait` definitions can be refined in three ways: by refining their existing components, by adding new components, or by refining the family or trait they extend.

To illustrate the last point, consider `Desugar_common` (Figure 2). Rocqet allows the identity of the family that `S` extends to be an extensibility hook, which `Desugar_If1` refines. For soundness, Rocqet requires this refinement to preserve that `S` still descends from `CoreC`.

To prevent circular reasoning, Rocqet requires that the relative order of nested components be preserved in derived families, as FPOP does.

- Refinement can also take the form of *overriding*. Opaque proofs can be freely overridden, as they possess no computational content. Components left undefined (think of them as abstract methods in OO languages) can also be overridden.

Traits and composition. Traits are reusable components that refine the behavior of families they are mixed into [5, 14, 19, 3, 24, 46]. For example, the command `Trait CoreC_If1 extends CoreC` begins the definition of a trait that can only be mixed into a family that is, or extends, `CoreC`. The command `Family C extends CoreC using CoreC_If1, CoreC_Incr` begins the definition of a family constructed via *mixin composition*: extending the behavior of `CoreC` with the refinements made by the two traits.

Rocqet allows nested families to be refined by traits. For instance, the trait `Comp_Loops` refines the nested family `CoreC`. Notice that when a trait refines nested families, Rocqet requires the families to be declared as traits in the enclosing trait. For instance, `Comp_Loops` declares `CoreC` as a trait (Figure 3), rather than as a family as `Comp` does. This requirement is because traits can only be *mixed into* families; they cannot be used as families. The `CoreC` in trait `Comp_Loops` is intended to be mixed into the `CoreC` in the family that `Comp_Loops` is mixed into.

Mixin composition may entail *nested composition*, if the family and traits being composed contain nested families or traits. Consider the family `CoreC`, the trait `CoreC_If1`, and the family `C` in `CompX`. All these components are implicit in `CompX`. The programmer writes no code for them; they are automatically synthesized by Rocqet via nested composition.

- `CompX.CoreC` is a family constructed by mixing `Comp_Loops.CoreC` into `Comp.CoreC`.
- `CompX.CoreC_If1` is a trait constructed by refining `Comp.CoreC_If1`. The refinement is about the families that `CoreC_If1` can be mixed into: it is required to be mixed into a family that is, or extends, the just constructed `CompX.CoreC`.
- `CompX.C` is a family constructed by composing the verbatim contents of `Comp.C` (which is empty), the just constructed `CompX.CoreC_Incr`, `CompX.CoreC_If1`, and `CompX.CoreC`.

Any ambiguity arising from mixin composition and nested composition is reported to the programmer as a conflict that must be resolved.

Mixin composition and nested composition may generate extra proof obligations. This is an instance of the known phenomenon of *feature interaction* [4]: features that work correctly in isolation may require coordination when composed. For example, consider two independent extensions, one adding a new constructor to an inherited `FInductive` type and another adding a new `FRecursion` function over that inductive type. Rocqet generates a proof obligation when the two extensions are composed, requiring that the extra recursive function handle the extra constructor.

Traits are not present in FPOP [27], but FPOP does support mixins by allowing the dual use of families as mixins. Muddling a conceptual distinction aside, this conflation of families and mixins has performance consequences. Our experience suggests that it would not scale well to large-scale developments in the presence of nested family polymorphism. The conflation would cause extensions like `Comp_Loops` and `Desugar_If1` to be compiled as complete families, which may require excessive memory usage, an issue not uncommon in large-scale verification efforts using the Rocq prover (e.g., [22]).

Equalities involving `FRecursion`. In Rocqet, nested components are polymorphic to every family they are nested within. This flexibility means that nested `FRecursion` definitions do not come with a fixed meaning; they can acquire new cases when inherited into a derived family. As a result, within the families that define or refine an `FRecursion` say, `transl_stmt` in `Comp.Desugar_common`, we cannot rely on the existence of a *definitional equality* between `transl_stmt` and a fixed point construction. That is, a reference to `transl_stmt` within `Comp.Desugar_common`—or even a reference to `Desugar_common.transl_stmt` within `Comp`—cannot be unfolded to a fixed-point definition exhaustively constructed from all the cases known in that context. Such a definitional equality would prevent `transl_stmt` from acquiring new cases in derived families of `Comp` or of `Desugar_common`.

Fundamentally, the definitional equality is too strong to hold because a *recursor* (aka *eliminator*) is not available. If `stmt` were an `Inductive` (non-extensible) instead of an `FInductive` (extensible), a recursor like `stmt_rect` would be available, where `P` is known as the *motive*:

```
stmt_rect : ∀ (P : stmt → Type), P Sskip → (∀ s1, P s1 → ∀ s2, P s2 → P (Sseq s1 s2)) →
            (∀ e, P (Sdo e)) → (∀ s1, P s1 → ∀ s2, P s2 → ∀ e, P (Sif e s1 s2)) → … → ∀ t, P t
```

The recursor is the induction principle used to define recursive functions or induction proofs over `stmt`. It requires that `stmt` be exhaustively generated by the constructors `Sskip`, `Sseq`, etc.

Although definitional equalities based on recursors are not possible, certain propositional equalities still hold. For example, like FPOP, Rocqet automatically generates equalities like

```
∀ s1 s2, transl_stmt (S.Sseq s1 s2) = do s1' ← transl_stmt s1; do s2' ← transl_stmt s2; ret (T.Sseq s1' s2')
```

upon `End transl_stmt` in `Comp.Desugar_common`. Such propositional equalities capture the *computational behavior* of `transl_stmt` on each constructor of `S.stmt` known in that context. An `fsimpl` tactic exists in Rocqet to facilitate rewriting along these equalities. It is useful in proofs such as `Comp.Desugar_common.transl_stmt_correct` to rewrite say, `transl_stmt (S.Sseq s1 s2)`. Compared to FPOP, Rocqet makes quality-of-life improvements to tactics. For example, `fsimpl` can be used in proof scripts to rewrite goals and hypotheses in a way that appears as if Rocq's `simpl` tactic were used to unfold or refold fixed-point definitions. `fconstructor` is a new tactic that can be used to automatically apply constructors of an `FInductive` in a way that appears as if Rocq's `constructor` tactic were used.

Equalities involving `Family` and `Trait`. In Rocqet, nested families and traits do not have fixed meanings within the context they are defined. In Figure 2, it is this flexibility that allows the nested family `S` to be refined in `Desugar_If1`, `Desugar_Incr`, and `CoreCgen`.

However, it is sometimes useful to have definitional equalities over nested families or traits. Consider the function `c_to_clight` shown in Figure 3. It chains two compiler passes, transforming a C statement to a Clight statement: `Clightgen.lower_stmt ∘ CoreCgen.transl_stmt`. For this function to be well-typed, the input and output types must match. The input type of `CoreCgen.transl_stmt` is `CoreCgen.S.stmt`, while the required input type of `c_to_clight` is `C.stmt`. Rocqet considers these two types interchangeable, because `CoreCgen` declares `S` via the command `Family S := C`. This command refines `S` to be definitionally equal to `C`, preventing `S` from being further refined by any family extending or refining `CoreCgen`. By contrast, if `CoreCgen` had declared `S` via the command `Family S extends C`, similar to how `Desugar_common` declares `S`, then `CoreCgen.S` and `C` would not be considered definitionally equal, and thus `c_to_clight` would be ill-typed.

Notice that despite the strong equality between `CoreCgen.S` and `C`, the nested family `C` may still be refined in a derived family of `Comp`. For example, statements in `CompX.C` include loops, yet `CompX.CoreCgen.S` is still definitionally equal to `CompX.C`. As a result, running the command `Check CompX.c_to_clight` will succeed and print the type `CompX.C.stmt → CompX.Clight.stmt`.

Figure 4. Translation of selected components from Figure 1.

Also notice that outside the family `Comp` or `CompX`, references to compiler components are no longer late bound but are resolved to fixed meanings. As a result, `CompX.CoreCgen.transl_stmt` is definitionally equal to a fixed-point function that exhaustively handles all constructors in `CompX.C.stmt`.

4 Compiling Rocqet to Rocq

We implement the language design described in Section 3 as a Rocq plugin that compiles high-level Rocqet features into Rocq. Despite Rocqet's greater expressive power compared to FPOP [27], our implementation is more efficient. Compilation is modular, as compilation artifacts—even for deeply nested components—can be shared without having to be rechecked. The compilation strategy also allows for seamless integration with interactive theorem proving, as components can be compiled step by step no matter how deeply nested they are.

Compiling nested components into multiply-parameterized modules. The essence of inheritance is implicit self-parameterization [7]. Nested family polymorphism extends this idea to all fields (i.e., nested components) within a family, including nested families, making each field polymorphic to its context. We take this principle quite literally to develop our compilation strategy. Compiling a field yields two artifacts:

- A `Module Type` representing the field's typing context within its immediately enclosing family.
- A `Module Type` (or `Module`, if the field is not an extensibility hook) parameterized by this context, representing the field itself.

Both artifacts may also be parameterized by additional context parameters that represent outer families enclosing the current family. For compiling the next field, its context artifact is constructed from the two compilation artifacts of the current field.

As an example, consider the field `Comp.CoreC.stmt` in Figure 1. It is compiled to two artifacts (Figure 4). First, the module type `Comp°CoreC°stmt°Ctx` (lines 5–9) represents the field's context; it contains the compilation artifacts of the fields preceding `stmt`. Second, the module type `Comp°CoreC°stmt` (lines 11–20) represents the field itself and has a parameter called `self[CoreC]` whose type is the context artifact. Both artifacts are parameterized by `self[Comp]` representing the outer family `Comp`. To reduce visual clutter, we use a lighter color for prefixes of mangled names; they are not the focus of the presentation.

Names bound in the typing context can be referenced via these self parameters. For example, in Figure 1, `stmt`'s constructor `Sdo` has type `expr → stmt`, which is compiled to `self[CoreC].expr → stmt` (line 18). Notice `self[CoreC]` has type `Comp°CoreC°stmt°Ctx self[Comp]` (lines 13–14), which depends on `self[Comp]`. So the type `self[CoreC].expr → stmt` is polymorphic to each of the enclosing families `CoreC` and `Comp`. Hence, the compilation artifact of `Sdo` can be shared with extensions of `CoreC` and extensions of `Comp`, even when they add new constructors to `expr`.

As another example, consider the field `CoreC` of `Comp`. Again, this field has two compilation artifacts: the context `Comp°CoreC°Ctx` (lines 2–3), and the field itself `Comp°CoreC` (lines 48–52), which is parameterized by `self[Comp] : Comp°CoreC°Ctx`. These two artifacts are included as the context information (lines 55–56) for compiling the field `CoreC_If1` occurring after `CoreC` in the `Comp` family, just as the two compilation artifacts of `stmt` (lines 5–9, 11–20) are included as the context information (lines 24–25) for compiling the field `env` occurring after `stmt` in the `Comp.CoreC` family.

Compiling `FInductive`. `FInductive`s are extensibility hooks, so they are compiled to module types. For example, we have seen that `stmt` in Figure 1 is translated to the module type `Comp°CoreC°stmt`. Importantly, the module type does not define a concrete `Inductive` type, which would be closed to extension. Instead, it only states the existence of `stmt` and the type of its constructors; it does not claim that `stmt` is exhaustively generated by these constructors. In particular, it does not provide a recursor like `stmt_rect`. The compilation artifact does provide a *partial recursor*, as in FPOP [27], that facilitates proving constructor disjointness and injectivity; we omit it here for brevity.

As we will see, this compilation artifact of `stmt` can be shared with extensions of `CoreC` and extensions of `Comp`, which may refine `stmt` by adding new constructors.

Compiling `FRecursion` and `FInduction`. These definitions are extensibility hooks too, so they are compiled to module types. For example, the field `Comp.Desugar_common.transl_stmt` (Figure 2) is translated to the module type `Comp°Desugar_common°transl_stmt` (Figure 5, lines 13–20), as well as a context artifact (omitted in Figure 5). The module type `Comp°Desugar_common°transl_stmt` only declares the type of the function but does not provide an implementation.

Though we cannot define `transl_stmt` as a fixed point by exhaustive induction at this point yet (to ensure extensibility), as discussed in Section 3, we do want the computational behaviors available as propositional equalities. To this end, a module type `Comp°Desugar_common°transl_stmt°CB` (lines 22–31) is emitted, which declares the computational behaviors of `transl_stmt` on constructors of `S.stmt`. For instance, the computational behavior of `transl_stmt` for `Sskip`, named `transl_stmt°Sskip°eq`, is axiomatized with the equality shown on lines 26–29. The right-hand side of the equality is a reference to the translation of the `Case Sskip` handler, which is defined in the module `Comp°Desugar_common°transl_stmt°Cases` (lines 2–11), which has been made available in the context. So Rocq can unfold the reference and simplify the axiomatized equality to

```
self[Desugar_common].transl_stmt self[Desugar_common].S.Sskip = ret self[Desugar_common].T.Sskip .
```

Rocqet's `fsimpl` tactic emulates Rocq's `simpl` tactic, by performing rewriting along such axiomatized equalities and also unfolding.

Figure 5. Translation of selected components from Figure 2.

Importantly, `FRecursion` need not be recompiled for families by which it is inherited or refined. These families can reuse (via Rocq's `Include` command) the already compiled case handlers and computational behaviors, without having them rechecked, thanks to the self-parameterization of the compilation artifacts. For example, the compilation of `CoreCgen` in `Comp` reuses the `transl_stmt` case handlers and computational behaviors compiled for `Desugar_common` and `Desugar_If1` (lines 71–84), by instantiating the self parameter of the outer family explicitly and the self parameter of the inner family implicitly (explained later).

`FInduction` definitions are compiled similarly, but since these proofs are considered opaque, no computational behavior needs to be declared.

Compiling nested `Family`. Nested families, if they are extensibility hooks, are compiled into module types. Consider `CoreC` in Figure 1. Upon `End CoreC`, the family is compiled into a module type `Comp°CoreC` (Figure 4, lines 48–52). This module type simply declares that a module named `CoreC` exists and has signature `Comp°CoreC°Sig`, without binding the name `CoreC` to any concrete implementation of the signature. The module type `Comp°CoreC` is then included as context information (e.g., lines 55–56) for compiling other fields in `Comp`; not revealing the concrete implementation of `CoreC` ensures that those other fields can reference `CoreC`'s contents only through late binding.

The signature `Comp°CoreC°Sig` (Figure 4, lines 37–46), parameterized by `self[Comp]` representing the enclosing family, is defined by including the compilation artifact of each field of `CoreC`. Recall that these artifacts have two parameters, `self[Comp]` and `self[CoreC]`. While the `self[Comp]` parameter can be easily instantiated by the `self[Comp]` in scope, the instantiation of `self[CoreC]` is left to Rocq: Rocq implicitly instantiates any missing module argument with the current interactive module environment, which is exactly what we want!

Now, consider the compilation of the family `C` nested in `Comp` (Figure 1). `C` extends `CoreC`, so how does its compilation reuse the compilation artifacts of say, `CoreC.stmt`? One idea would be to include

in `Comp°C°stmt` the compilation artifact of `CoreC.stmt` via `Include Comp°CoreC°stmt self[Comp]`. But in a derived family of `Comp`, like `CompX`, `CoreC.stmt` may have more constructors than `Comp°CoreC°stmt` declares, which would prevent `Comp°C°stmt` from being reused by `CompX°C°stmt`. The solution is to not hard-code the compilation artifact of `CoreC.stmt` to be included in `Comp°C°stmt`, but instead to make the inclusion truly polymorphic to the enclosing `Comp` family via a `self[Comp]` prefix (Figure 4, lines 89–91): `Include self[Comp].CoreC.stmt°Art self[Comp]`. This translation requires the signatures `Comp°CoreC°Sig` and `CompX°CoreC°Sig` to bind the name `stmt°Art` to `Comp°CoreC°stmt` and `CompX°CoreC°stmt`, respectively. The signatures act like "dispatch tables" that enable the late binding of compilation artifacts.

Compiling nested `Family` with definitional equality. As discussed in Section 3, nested families are not always extensibility hooks. Nested families, if they are not extensibility hooks, are compiled into concrete modules. For example, in Figure 2, `CoreCgen.S` is specified to be definitionally equal to `C`. This relationship is reflected in `Comp°CoreCgen°Sig` (Figure 5, lines 66–87). Rather than merely declaring the existence of `S`, it binds `S` via `Module S := self[Comp].C` (line 68). Notice the module `S` is definitionally equal to the self-prefixed reference `self[Comp].C`, so this definitional equality is preserved in all contexts where `Comp°CoreCgen°Sig` is included, even if `C` is refined.

Compiling `Trait`. Traits are compiled similarly to families, but with two key distinctions. First, unlike families, which are compiled by using `Include` to reuse the compilation artifacts of inherited fields, traits only compile their delta with respect to the family they will be mixed into. Second, when a top-level family is closed (e.g., `End Comp`), nested families must be compiled to concrete modules, whereas traits need not be. Hence, traits have leaner compilation artifacts, which offer them a performance advantage. Compared to FPOP, which conflates families and mixins and lacks support for traits, our implementation of Rocqet significantly reduces memory usage during compilation.

Compiling `FDefinition`. By default, an `FDefinition` cannot be overridden and is thus compiled to a concrete module. An example is `CoreC.env` in Figure 1. Its compilation artifact `Comp°CoreC°env` (Figure 4, lines 28–34) is a concrete module that exposes the implementation of `CoreC.env`, which prevents it from being refined anywhere `Comp°CoreC°env` is included. An `FDefinition` that is left undefined, by contrast, can be overridden and is thus compiled to a module type that merely specifies the type of the field without revealing the concrete implementation.

Compiling top-level `Family`. Compilation differs for nested families and top-level families. Closing a nested family generates a module type parameterized by its enclosing families. However, closing a top-level family should produce a concrete module without any self parameters. This concrete module is really the fixed point of the self-parameterized translation of the family's fields.

This fixed point is taken by a well-founded induction over the list of fields in the family: each field's compilation artifact is included in the same order as it appears in the Rocqet program, with the appropriate self parameters applied. As noted earlier, the self parameter representing the immediately enclosing family is left for Rocq to instantiate, while the remaining self parameters are explicitly instantiated. Below, we illustrate how different types of fields participate in the construction of the fixed points (Figure 6) compiled for the top-level families `Comp` and `CompX`.

- `FDefinition`. Take `CompX.C.env` as an example. Its concrete implementation is available in Figure 4, in the module `Comp°CoreC°env`. In Figure 6, this module is included via the command `Include Comp°CoreC°env C.Ctx` (line 51) without having to be rechecked.
- `FInductive`. Take `Comp.C.stmt` as an example. The inductive type and its constructors, which are only axiomatized in `Comp°C°stmt` in Figure 4, are *concretized* as a Rocq `Inductive` type in Figure 6 (line 9), which comes with recursors that will allow `FRecursion` and `FInduction` over `stmt` to also

```
1  Module Comp.
2   Module CoreC. ... End CoreC.                    (* concretize CoreC *)

4   Module C.                                       (* concretize C *)
5    Module Ctx.
6     Include CoreC.Ctx.  Module CoreC := Comp.CoreC.
7    End Ctx.
8    Inductive expr : Type := ....
9    Inductive stmt : Type := Sskip : ... | ... | Sif1 : ....
10   Include Comp°CoreC°env C.Ctx.
11   ...
12  End C.

14  Module Desugar_common. ...                      (* concretize Desugar_common *)
15  End Desugar_common.

17  Module CoreCgen.                                (* concretize CoreCgen *)
18   Module Ctx.
19    Include Desugar_common.Ctx.
20    Module Desugar_common := Comp.Desugar_common.
21   End Ctx.
22   Module S := C.  Module T := CoreC.
23   ...
24   (* reuse case handlers *)
25   Include Comp°Desugar_common°transl_stmt°Cases
26           CoreCgen.Ctx.
27   Include Comp°Desugar_If1°transl_stmt°Cases
28           CoreCgen.Ctx.
29   (* concretize transl_stmt using the recursor of stmt *)
30   Def transl_stmt: S.stmt → res T.stmt :=
31    stmt_rect transl_stmt°Sskip ... transl_stmt°Sif1.
32   (* concretize computational behaviors of transl_stmt *)
33   Fact transl_stmt_Sskip_eq:
34    transl_stmt S.Sskip = transl_stmt°Sskip.
35   Proof. reflexivity. Qed.
36   ...
37  End CoreCgen.

39  ...                                             (* concretize other fields of Comp *)
40 End Comp.
```

```
41 Module CompX.
42  Module CoreC. ... End CoreC.                    (* concretize CoreC *)

44  Module C.                                       (* concretize C *)
45   Module Ctx.
46    Include CoreC.Ctx.  Module CoreC := CompX.CoreC.
47   End Ctx.
48   Inductive expr : Type := ....
49   Inductive stmt : Type := Sskip : ... | ... | Sif1 : ...
50   | Swhile : ... | Sfor : ... | Sdowhile : ....
51   Include Comp°CoreC°env C.Ctx.
52   ...
53  End C.

55  Module Desugar_common. ...                      (* concretize Desugar_common *)
56  End Desugar_common.

58  Module CoreCgen.                                (* concretize CoreCgen *)
59   Module Ctx.
60    Include Desugar_common.Ctx.
61    Module Desugar_common := CompX.Desugar_common.
62   End Ctx.
63   Module S := C.  Module T := CoreC.
64   ...
65   (* reuse case handlers *)
66   Include Comp°Desugar_common°transl_stmt°Cases
67           CoreCgen.Ctx.
68   Include Comp°Desugar_If1°transl_stmt°Cases
69           CoreCgen.Ctx.
70   Include Comp_Loops°Desugar_common°transl_stmt°Cases
71           CoreCgen.Ctx.
72   (* concretize transl_stmt using the recursor of stmt *)
73   Def transl_stmt: S.stmt → res T.stmt := stmt_rect
74    transl_stmt°Sskip ... transl_stmt°Sif1
75    transl_stmt°Swhile transl_stmt°Sfor transl_stmt°Sdowhile.
76   ...
77  End CoreCgen.

79  ...                                             (* concretize other fields of CompX *)
80 End CompX.
```

Figure 6. Translation of top-level families `Comp` (Figure 3) and `CompX`. All self-parameterizations are eliminated.

be concretized. For mutual `FInductive` types, in addition to including the corresponding mutual `Inductive` types, Rocqet also generates mutual recursors on an as-needed basis.

- `FRecursion` and `FInduction`. Take `Comp.CoreCgen.transl_stmt` as an example. The recursive function and its computational behaviors, which are only axiomatized previously, are now concretized in Figure 6. First, the concrete definitions of all the case handlers are included, without being rechecked (lines 25–28). Then, the recursive function is defined by applying the recursor `rect_stmt` to the case handlers (lines 30–31). Finally, the computational behaviors can be proved trivially by `reflexivity`, as the two sides of the equality are now definitionally equal (lines 33–36). `FInduction` is concretized similarly, but without proving any computational behavior.
- Nested `Family`. Nested families are concretized into nested modules, so that it is possible to access nested components via, for example, `Extraction CompX.CoreCgen.transl_stmt`. The fields of a nested family are concretized with the same mechanism as described herein. Before the fields are concretized, a module `Ctx` is emitted that represents the enclosing family of the nested family. For example, in Figure 6, the `CoreCgen` family nested in `CompX` is concretized to a nested module with the same name (lines 58–77). Before concretizing the fields of `CoreCgen`, a module `Ctx` (lines 59–62) is emitted. This module `Ctx` can be used to instantiate the self parameter when reusing the compilation artifact of a nested component. For example, the case handler that has been compiled for `Comp_Loops.Desugar_common.transl_stmt` is reused via the command `Include Comp_Loops°Desugar_common°transl_stmt°Cases CoreCgen.Ctx` on lines 70–71.

Trusted base. The trusted base of any development using Rocqet consists of the Rocq prover and the Rocqet compilation process. Trust in the compilation process is to a large extent mitigated, because the compiled code is checked by Rocq. The only trust required in the compilation process is that it correctly generates the language definition and the final theorem statement; no trust is placed in any axiomatized definitions or facts. For executing the extracted code, trust is also required in Rocq's extraction machinery and the OCaml compiler.

Performance. As we show later, our implementation of Rocqet scales significantly better than FPOP [27] despite providing greater expressive power. The speedup is due to careful language-design and engineering decisions, such as the decoupling of the mixin functionality from families.

5 A Formal Model of Rocqet: A Core Dependent Type Theory

We provide a formal model of Rocqet as a core dependent type theory. It extends the Martin-Löf dependent type theory [37] and builds on the formal model of FPOP [27], incorporating facilities to encode nested family polymorphism. In particular, it encodes a family with a language construct that resembles a record (of nested components) but where each component is universally quantified over all the components in its context—including those in all of its enclosing families. The core type theory also has a construct for encoding traits as functors that can be composed. We prove the canonicity and consistency of this core type theory and provide example encodings. The main paper focuses on the design and implementation of Rocqet; we defer the development of the formal model to Section A and Section B.

6 Case Study: An Extensible Certified C Compiler Framework

We apply Rocqet and the reuse strategies outlined in Section 2 to construct an extensible certified compiler framework for C-like languages. While we build on CompCert's IRs, we make some key architectural innovations that prioritize modularity and promote reuse.

Base compiler. Our development begins with a minimal subset of CompCert C. This base language excludes structured loops, switch statements, function calls, heap memory access, and structs and unions. The resulting language resembles Imp [49]. A verified compiler for this base language is mechanized. The IRs, passes, and simulation proofs are reused by all extensions.

Reuse across similar IRs. CompCert's front end contains three closely related languages: Csharpminor, Cminor, and CminorSel (Figure 7). Among them, Csharpminor and Cminor have nearly identical *syntax*, differing mainly in their representation of switch statements; Cminor has a lower-level switch construct. Their key *semantic* distinction lies in stack-space modeling: Csharpminor uses a map data structure, while Cminor utilizes a native stack pointer. CminorSel differs from Cminor by introducing machine-dependent instructions but maintains similar syntax and semantics otherwise. We factored these common features into a nested base family `Cfam`. The derived families `Csharpminor`, `Cminor`, and `CminorSel` only need to model their distinctive features by instantiating or refining extensibility hooks. This design makes the distinctions between these IRs explicit.

In CompCert's back end, two IRs, Linear and Mach, share the same instruction structure but differ in the modeling of stack access in the semantics: Linear uses an abstract `slot`, while Mach employs pointer-offset addressing. In our framework, the IRs are specified as derived families (`Linear` and `Mach`) of a base family `Lfam`. Rocqet enables the sharing of both syntactic and semantic definitions across these two IRs.

Nanopasses. We decompose CompCert's program transformations into a nanopass style [52, 53, 30] for improved modularity. For example, the `SimplExpr` pass of CompCert removes side effects from C expressions, turning them into Clight, where only statements have side effects.

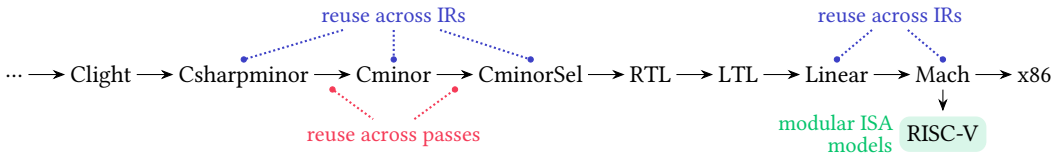


Figure 7. Rocqet promotes modular design and reuse of code representations and transformations.

We decompose this pass into nanopasses. Each nanopass focuses on only one type of effectful expressions (`Epostincr`, `Eassignop`, etc.). These nanopasses are mechanized as traits that extend a base translation between C and Clight. Each of them performs a surgical transformation and proves that transformation correct. The complete `SimplExpr` pass is then obtained by mixing these nanopasses into the base translation.

Reuse across compiler passes. The compiler passes between `Csharpminor`, `Cminor`, and `CminorSel` are known as `Cminorgen` and `Selection` in CompCert. As we have mechanized the IRs by making them share a common base, the compiler passes between them can also enjoy reuse. We define a base family `CfamTransl` that mechanizes the common, uninteresting transformations that have to be performed by all passes between two IRs derived from `Cfam`. The passes `Cminorgen` and `Selection` then reuse the transformations and proofs from `CfamTransl`, only making targeted refinements that are concerned with the compilation of the distinctive features of the IRs. In addition to the sharing between the two passes, each of `Cminorgen` and `Selection` is developed in a nanopass style using traits.

Modular RISC-V modeling. The RISC-V ISA is designed to be modular [28]. We capitalize on this modularity to mechanize the RISC-V syntax and semantics in a modular fashion. RISC-V offers a base instruction set `RV32I`, and another `RV64I` that builds on `RV32I`. These instruction sets have many extensions such as multiplication (`M`), floating-point operations (`F`, `D`), and vector processing (`V`). Rocqet allows this modularity in the RISC-V design to be faithfully modeled in a proof assistant. In the back end of our compiler framework, the `RV32I` and `RV64I` ISA base is modeled as concrete families, with `RV64I` extending `RV32I`, while extensions such as `D` and `M` are realized as traits that extend the base families. Thus, these extensions can be freely composed in a mix-and-match style. Importantly, the composability allows a compiler extension to customize the combination of RISC-V extensions it requires—a key motivation for the RISC-V initiative.

Extensions to the base compiler. The organization of the base compiler follows the reuse strategies described thus far. Additional C features are mechanized as traits that extend the base compiler family and, therefore, embody the same reuse principles.

(1) Structured loops are supported in a trait called `Comp_Loops` (Figure 3). This extension involves extending the IRs in the front end of the compiler with high-level or mid-level loop constructs.

(2) Switch statements are supported by a trait that extends the compiler front end with switch constructs and the back end with jump tables.

(3) Heap access is supported by a trait that extends the base compiler with heap operations. Heap operations manifest as references and pointers in the compiler front end and are made explicit via loads and stores in the back end.

(4) Structs and unions are supported by a trait that extends the base compiler with field access. Fields can be from structures or unions but are represented by the same expression form. This extension can either build on the heap extension or be independently developed. If developed

```
Family Comp. … End Comp.

Trait Comp_Loops extends Comp. … End Comp_Loops.
Trait Comp_Switch extends Comp. … End Comp_Switch.
Trait Comp_Heap extends Comp. … End Comp_Heap.
Trait Comp_Field extends Comp. … End Comp_Field.
Trait Comp_External extends Comp. … End Comp_External.
Trait Comp_Builtin extends Comp. … End Comp_Builtin.
Trait Comp_Call extends Comp. … End Comp_Call.

Family CompCert extends Comp
 using Comp_Loops, Comp_Switch, Comp_Builtin,
       Comp_Heap, Comp_External, Comp_Field, Comp_Call.
End CompCert.
```

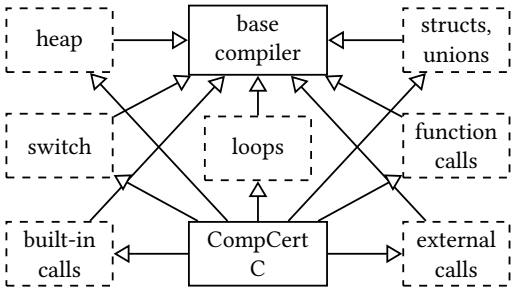


Figure 8. Rocqet enables à la carte composition of compiler extensions.

independently, it leads to a feature interaction when composed with the heap extension, as fields can be lvalues, which require heap access.

(5) Function calls are supported in three extensions that extend the base compiler with calls to built-in functions (e.g., `__builtin_sel`), external functions (e.g., `malloc`), and user-defined functions.

These extensions can be composed with the base compiler to yield custom compilers that support different combinations of C features. The ability to select a subset of these extensions is useful, for example, in developing safety-critical applications for embedded systems, where it is not uncommon that features like dynamic memory allocation and external function calls may be prohibited due to resource constraints.

When all the extensions are combined, they recover a compiler for CompCert C (Figure 8). It is possible to further extend this compiler with additional features beyond those in CompCert C; we leave this as future work.

Experience report. Most of the effort in developing this extensible compiler framework was spent on designing and engineering the base compiler. Once the base compiler was established, building extensions became straightforward. The fact that each extension is a separate trait focusing on a single feature eased development and coordination, allowing us to develop and verify each extension independently.

The sharing among IRs and passes reduced code and proof complexity. For instance, in the `Comp_Loops` extension, the IR modules `Cminor`, `CminorSel`, and `Csharpminor` share the same loop construct, which means that the syntax and semantics of loops need to be mechanized only once and, thus, that the `Cminorgen` and `Selection` passes only need to be refined with a single identity transformation in this compiler extension. This simplification reduced engineering effort.

For porting many of the CompCert's program transformations to the Rocqet syntax, an AI-powered coding assistant like GitHub Copilot proved effective at handling mechanical conversion tasks. The high success rate of this automated conversion is a potential indicator that the language design of Rocqet allows a programming style familiar and intuitive to a working Rocq programmer.

7 Evaluation

Performance of proof compilation compared to FPOP. We evaluate the speedup in proof-compilation time that Rocqet offers over FPOP. Our benchmarks are from the prime case study that Jin et al. [27] conducted with their FPOP implementation. This case study concerns the type-safety proofs of five extensions to the simply typed lambda calculus (λ): Booleans ($\mathbb{B}$), products ($\times$), sums ($+$), term-level fixed points (Y), and iso-recursive types (μ). Both FPOP and Rocqet can express these extensions as mixins (families in FPOP and traits in Rocqet) to a base STLC mechanization. These

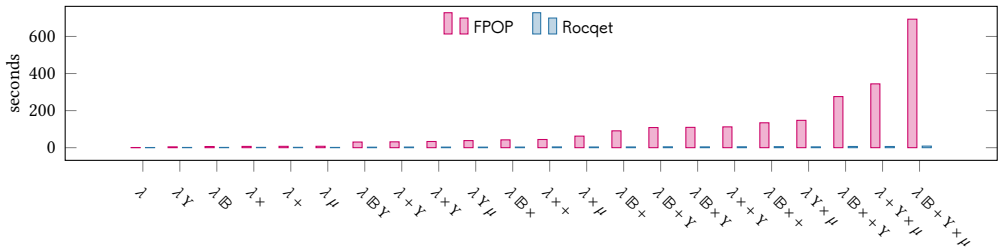


Figure 9. Proof-compilation times for different combinations of STLC extensions: FPOP vs. Rocqet.

Table 1. Proof-compilation times (in seconds) of CompCert modules.

	SimplExpr	Cshmgen	Cminorgen	Selection	RTLgen	Linearize	Stacking	Asmgen
Rocq	28.3	32.1	6.7	46.1	22.5	23.4	23.6	66.6
Rocqet	1680.0	267.9	209.3	128.4	512.4	97.1	242.0	159.8

extensions are picked and composed to yield a host of STLC variants and their type-safety proofs. Proofs can be written in largely the same style as those in Software Foundations [49].

We measure the time taken for FPOP and Rocqet to compile different combinations of STLC extensions (including the time taken to compile the base STLC mechanization). Figure 9 presents the results.[5] The x-axis shows the combinations we benchmarked, ordered by increasing number of extensions. As the complexity of the benchmark increases, the time taken by FPOP rises substantially. In contrast, Rocqet scales well: compilation time increases in proportion to the number of extensions composed. When all five extensions coexist, Rocqet achieves a speedup of 81× over FPOP.

A key reason for this speedup is that Rocqet avoids the iterated self-parameter instantiation (as in Figure 6) for each mixin, only doing it once for the entire composition. Proof-compilation performance also benefits from Rocqet optimizations that reduce the number and size of generated contexts, which are Rocq `Module Type`s. These optimizations include reusing previously generated contexts when possible and generating smaller contexts that contain only the necessary components serving as extensibility hooks.

Performance of proof compilation compared to Rocq. While Rocqet significantly outperforms FPOP in compilation times, its powerful extensibility mechanism inevitably introduces an overhead. We evaluate the slowdown that Rocqet introduces over Rocq for compiling the CompCert modules. Table 1 shows the results. For each module, the reported Rocqet time includes that for the base compiler and all the extensions. An average slowdown of 10× (geometric mean) is observed. The SimplExpr pass is a notable outlier, taking considerably more time. It imports a complex C semantics, stressing memory and computation due to compilation into a large number of Rocq `Module`s—likely in ways not anticipated by Rocq's implementation for typical usage patterns. Despite the current cost on proof compilation, we view this as an acceptable trade-off for gaining extensibility and reuse at a very large scale. Improving proof-compilation performance remains future work.

Performance of extracted code. We extract an executable compiler from our port of CompCert in Rocqet. We evaluate the performance of this compiler against the original CompCert, on CompCert's default test suite. This suite encompasses a range of C programs, from basic algorithms to complex applications. Table 2 shows the results. Each cell shows the total compilation time on all test cases under a directory in the test suite.

[5] All measurements in Section 7 were taken on a machine with an AMD Ryzen 7 7700X (4.5 GHz) and 64 GB RAM.

Table 2. Benchmark compilation times (in seconds) with the extracted compilers, on CompCert's test suite.

	raytracer	regression	compression	c	spass	abi
Rocq	0.725	5.582	0.445	1.654	17.236	19.928
Rocqet	0.725	5.601	0.444	1.649	17.259	19.924

Table 3. Lines of code needed for building compiler extensions without and with Rocqet.

	base	+loops	+switch	+calls	+builtin	+external	+heap	+fields
Rocq	16 739	18 149	17 762	18 338	18 162	17 141	19 407	17 100
Rocqet	17 021	1 410	1 023	1 599	1 423	402	2 668	361

The measurements show that the extracted compiler has comparable performance to the original CompCert. This finding is interesting, as it confirms the fusion of nanopasses in our extracted compiler. The prior work on the nanopass framework does not support fusion, and it reports ~1.7× longer compilation times compared to a non-nanopass compiler [30]. Our results suggest that the extensibility and modularity afforded by Rocqet come at minimal performance cost.

Reuse. We evaluate the degree of reuse that Rocqet enables for developing compiler extensions. Our primary metric is lines of code (LOC). Table 3 reports the results.

The "base" column compares the base compiler written in Rocqet to a stripped-down version of CompCert in Rocq that includes only equivalent features. The remaining columns report the code size for each compiler extension. Rocqet numbers are counted directly. Rocq numbers are estimates—we did not implement in Rocq these stripped-down versions of CompCert, as doing so would require substantial engineering effort without yielding new insights. We obtain the LOC for the base compiler in Rocq by removing lemmas, induction cases, and other definitions not present in the Rocqet version of the base compiler. To estimate the extra LOC needed for a compiler extension, we use the LOC from Rocqet as a proxy. This proxy is a reasonable estimate, as the Rocqet proofs in the compiler extensions are written following a similar style to CompCert.

The results confirm that Rocqet enables significant reuse. Once the base compiler is implemented, writing a new compiler extension in Rocqet requires only the code and proofs specific to the new feature; all other components are inherited from the base compiler and reused. In contrast, extending the base compiler in Rocq requires duplicating code from the base compiler, which leads to a substantial increase in code size and makes the codebase harder to maintain.

8 Related Work

Extensibility and compiler verification. There are two main approaches to compiler verification.

One approach aims at compilers that, in addition to compiled code, also produce a correctness proof of the compiled code. Such compilers are known as *certifying compilers* or *proof-producing compilers*. Rupicola [50] is a certifying compiler that casts verified compilation of Gallina programs to C code as proof search. A design goal of Rupicola is extensibility. Rupicola is extensible in a different sense than what we have discussed so far: in Rupicola, new translations can be added in the form of lemmas, which are then used by the code-generating proof search. Relying on proof search potentially limits scalability; it is reported that Rupicola compiles 2–15 statements per second.

Another approach to compiler verification aims at compilers that are proven correct. Such compilers are known as *certified compilers*—two prime examples are CompCert [36] and CakeML [58]. Tatlock and Lerner [59] and Gross et al. [23] introduce frameworks that enable modularly extending

a certified compiler with new optimizations in the form of rewrite rules. These frameworks are orthogonal and complementary to our work.

Extensibility and metatheory mechanization. Our work is more closely related to prior work on extensible metatheory mechanization [11–13, 54, 31, 20, 27, 40], which is driven by a fundamental tension between adding new constructors to an inductive type and adding new functions or theorems that operate by induction over that type—an instance of the expression problem [61] in the context of proof languages.

Existing solutions to the expression problem in this space are largely extralinguistic, in the sense that they are code-organizing techniques. Much prior work [12, 13, 54, 31, 20] is inspired by the *data types à la carte* (DTC) encoding technique [57]. Notably, the *metatheory à la carte* work [12] uses Church encodings for data types, Mendler-style folds for evaluation, and type classes for feature composition. Techniques based on encodings may lead to convoluted code, which makes the programming style less accessible and idiomatic. Likely for this reason, their use has been limited to smaller-scale developments than certified compilers.

In contrast, family polymorphism in FPOP and Rocqet is a language-design solution. Thus, it is to a large degree not confined by the host proof language. By introducing new language features, Rocqet allows a more idiomatic style of programming and proving—languages and their metatheories can be mechanized in a way that aligns closely with textbook presentations. To our knowledge, our work is the first to address the expression problem in the context of compiler verification, which involves a broader set of challenges than metatheory mechanization.

Nested family polymorphism. A key reason for the scalability of our approach is that the name of every nested component—ranging from small elements like inductive types and induction proofs to larger structures like entire families and traits—is a potential hook for extending behavior.

This design draws inspiration from prior languages supporting nested family polymorphism [43, 45, 62, 32] in standard OO or functional contexts. Rocqet is the first to support it in an interactive theorem prover. Previous work on a Java-like language [42] reports performance overhead introduced by a compilation scheme that uses wrapper objects to support nested family polymorphism. The indirection introduced in Rocqet's compilation scheme is mostly concerned with creating fine-grained functions (e.g., cases of `FRecursion`), which does not seem to slow down extracted code as our experiments suggest.

9 Conclusion

We have presented the design and implementation of Rocqet. It extends the Rocq prover with novel, powerful language abstractions, offering a high degree of extensibility that may seem unusual even for a standard object-oriented or functional language. We were guided in the design of Rocqet by a real, important use case: the construction of verified compilers. The new expressive power afforded by Rocqet allows the monolithic design of a verified compiler to be modularized into reusable components, supporting flexible extension and à la carte composition of IRs, compiler transformations, and entire compilers. Our experience suggests that the rich abstraction mechanisms in Rocqet are practical and effective for structuring machine-checked proofs of large, complex certified systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. Any opinions and findings are those of the authors and do not necessarily reflect the position of any funders.

Data-Availability Statement

The artifact accompanying this paper is archived on Zenodo [17]. The latest versions of Rocqet and the Rocqet port of CompCert can be found at the following links:

https://github.com/rocqetry/rocqet
https://github.com/rocqetry/CompCert

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] Davide Ancona and Elena Zucca. 2002. A calculus of module systems. *Journal of Functional Programming (JFP)* 12, 2 (2002). https://doi.org/10.1017/S0956796801004257

[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] 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

[6] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32 (1986).

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

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

[9] Laurence E. Day and Graham Hutton. 2013. Compilation à la carte. In *Symp. on Implementation and Application of Functional Languages (IFL)*. https://doi.org/10.1145/2620678.2620680

[10] 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

[11] 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

[12] 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

[13] 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

[14] Dominic Duggan and Constantinos Sourelis. 1996. Mixin modules. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/232627.232654

[15] Peter Dybjer. 1995. Internal type theory. In *Int'l Workshop on Types for Proofs and Programs*.

[16] Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. Certified compilers à la carte. *Proc. of the ACM on Programming Languages (PACMPL)* 9, ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI) (2025). https://doi.org/10.1145/3729261

[17] Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. *Certified Compilers à la Carte (Artifact)*. https://doi.org/10.5281/zenodo.15052648

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

[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). https://doi.org/10.1145/3341711

[22] Jason Gross, Andres Erbsen, Jade Philipoom, Rajashree Agrawal, and Adam Chlipala. 2024. Towards a scalable proof engine: A performant prototype rewriting primitive for Coq. *Journal of Automated Reasoning (JAR)* 68, 3 (Aug. 2024). https://doi.org/10.1007/s10817-024-09705-6

[23] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Accelerating verified-compiler development with a verified rewriting engine. In *Int'l Conf. on Interactive Theorem Proving (ITP)*. https://doi.org/10.4230/LIPIcs.ITP.2022.17

[24] Tom Hirschowitz and Xavier Leroy. 2005. Mixin modules in a call-by-value setting. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 27, 5 (Sept. 2005). https://doi.org/10.1145/1086642.1086644

[25] 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

[26] Antonius J. C. Hurkens. 1995. A simplification of Girard's paradox. In *Int'l Conf. on Typed Lambda Calculi and Applications*.

[27] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible metatheory mechanization via family polymorphism. *Proc. of the ACM on Programming Languages (PACMPL)* 7, PLDI (2023). https://doi.org/10.1145/3591286

[28] David Kanter. 2016. RISC-V offers simple, modular ISA: New CPU instruction set is open and extensible. *Microprocessor Report* (March 2016). https://riscv.org/wp-content/uploads/2016/04/RISC-V-Offers-Simple-Modular-ISA.pdf

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

[30] Andrew W. Keep and R. Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/2500365.2500618

[31] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *9th ACM SIGPLAN Workshop on Generic Programming*. https://doi.org/10.1145/2502488.2502491

[32] Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested family polymorphism with extensible variant types. *Proc. of the ACM on Programming Languages (PACMPL)* 8, OOPSLA1 (April 2024). https://doi.org/10.1145/3649836

[33] Lindsey Kuper. 2019. My first fifteen compilers. https://blog.sigplan.org/2019/07/09/my-first-fifteen-compilers

[34] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Int'l Symp. on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO51591.2021.9370308

[35] Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. 1993. *Object-oriented programming in the BETA programming language*. Addison-Wesley.

[36] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning (JAR)* 43, 4 (Dec. 2009). https://doi.org/10.1007/s10817-009-9155-4

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

[38] 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)*.

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

[40] Dawn Michaelson, Gopalan Nadathur, and Eric Van Wyk. 2023. A modular approach to metatheoretic reasoning for extensible languages. arXiv:2312.14374 [cs.PL]

[41] Magnus O. Myreen. 2024. Much still to do in compiler verification (a perspective from the CakeML project). Keynote at the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2024). Recording: https://www.youtube.com/watch?v=eLFoHQgS6dA.

[42] Nathaniel Nystrom. 2006. *Programming languages for scalable software extension and composition*. Ph.D. Dissertation. Cornell University. https://hdl.handle.net/1813/3726

[43] 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

[44] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: an extensible compiler framework for Java. In *Int'l. Conf. on Compiler Construction (CC)*. https://doi.org/10.1007/3-540-36579-6_11

[45] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: nested intersection for scalable software composition. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1167473.1167476

[46] 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

[47] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972). https://doi.org/10.1145/361598.361623

[48] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. 2017. Miniphases: compilation using modular and efficient tree transformations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062346

[49] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2025. Software foundations: Volume 2 (programming language foundations). (2025). Version 6.8.

[50] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523706

[51] Rocq [n.d.]. The Rocq prover. https://rocq-prover.org.

[52] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A nanopass infrastructure for compiler education. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/1016850.1016878

[53] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2005. Educational pearl: A nanopass framework for compiler education. *Journal of Functional Programming (JFP)* 15, 5 (2005). https://doi.org/10.1017/S0956796805005605

[54] 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

[55] Lau Skorstengaard. 2019. An introduction to logical relations. (2019).

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

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

[58] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming (JFP)* 29 (2019). https://doi.org/10.1017/S0956796818000229

[59] Zachary Tatlock and Sorin Lerner. 2010. Bringing extensibility to verified compilers. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1806596.1806611

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

[61] 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.

[62] 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

A Syntax and Semantic Models of FMLTT$^{2.0}$

This appendix provides the metatheoretical details of FMLTT$^{2.0}$, a dependent type core calculus based on FMLTT [27]. Compared to FMLTT, FMLTT$^{2.0}$ simplifies some typing rules and equips the ability to encode nested inheritance.

In this appendix, we will use de Bruijn indices and *explicit substitutions* [1, 39]: substitutions γ and their applications (e.g., $T[\gamma]$, which applies γ to the type T) are part of the syntax rather than meta-operations. However, we may use named binders for better readability. We work in an intrinsically typed setting: terms are well typed by construction. Consequently, we omit without ambiguity some obvious premises needed for well-formedness.

Our style of syntax formulation follows a recent trend [2, 8, 21, 56] known as the "algebraic presentation" of MLTT. These works formulate the syntax using different tools or frameworks. Altenkirch and Kaposi [2] use *Quotient Inductive Inductive Type* (QIIT), Coquand [8] uses *category with family* (CwF), and Sterling [56] uses *Generalized Algebraic Theory* (GAT). Our formulation can be considered to be based on QIIT. However, we will manually quotient upon the syntax along *judgmental equalities* for the reader. The semantic model we develop in this appendix respects these judgmental equalities by construction.

A.1 Review MLTT with Explicit Substitutions and Universe Levels

We review the base MLTT fragment of FMLTT$^{2.0}$ first. Our formulation is close to that of Sterling [56], except for that we are using a more conventional notation and that we use manual quotient instead of GAT.

$$
\begin{array}{rcl}
\text{Contexts} & \Gamma, \Delta, \Theta & \cdot \mid \Gamma, A \\
\text{Substitutions} & \gamma & \mathtt{p}^n \mid \gamma, t \mid \gamma_1 \circ \gamma_2 \mid \pi_1 \gamma \mid \mathtt{id} \\
\text{Types} & A, B, T & T[\gamma] \mid \mathbb{U} \mid \mathbb{B} \mid \bot \mid \top \mid \Pi(A, B) \mid \Sigma(A, B) \mid \mathtt{Eq}(t_1, t_2) \mid \mathbb{S}(t) \mid \mathtt{El}(t) \\
\text{Terms} & t, s & t[\gamma] \mid \mathtt{var}_n \mid \pi_2 \gamma \mid \mathtt{c}(T) \mid () \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(t_1, t_2, t_3) \mid \lambda(t) \mid \mathtt{app}(t) \mid \\
 & & \langle t_1, t_2 \rangle \mid \mathtt{fst}\ t \mid \mathtt{snd}\ t \mid \mathtt{refl}(t) \mid \mathtt{J}(t_1, t_2)
\end{array}
$$

$$
\boxed{\Gamma \vdash} \qquad \boxed{\Gamma \vdash \gamma : \Delta} \qquad \boxed{\Gamma \vdash_j T} \qquad \boxed{\Gamma \vdash t : T}
$$

$$
\frac{}{\cdot \vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash_j A}{\Gamma, A \vdash}
$$

$$
\frac{j < k}{\Gamma \vdash_k \mathbb{U}_j} \qquad \frac{}{\Gamma \vdash_0 \mathbb{B}} \qquad \frac{}{\Gamma \vdash_0 \bot} \qquad \frac{}{\Gamma \vdash_0 \top} \qquad \frac{\Gamma \vdash_j A \quad \Gamma, A \vdash_j B}{\Gamma \vdash_j \Pi(A, B)} \qquad \frac{\Gamma \vdash_j A \quad \Gamma, A \vdash_j B}{\Gamma \vdash_j \Sigma(A, B)}
$$

$$
\frac{\Gamma \vdash_j A \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash_j \mathtt{Eq}(x, y)} \qquad \frac{\Gamma \vdash_j A \quad \Gamma \vdash a : A}{\Gamma \vdash_j \mathbb{S}(a)} \qquad \frac{\Delta \vdash_j T \quad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash_j T[\gamma]}\ [\text{TY/SUB}]
$$

$$
\frac{}{\Gamma \vdash A\left[\mathtt{p}^0\right] \equiv A} \qquad \frac{}{\Gamma \vdash A[\gamma_1 \circ \gamma_2] \equiv A[\gamma_1][\gamma_2]}
$$

$$
\frac{\Gamma \vdash \gamma : \Delta}{\begin{array}{c}
\Gamma \vdash \mathbb{U}[\gamma] \equiv \mathbb{U} \quad \Gamma \vdash \mathbb{B}[\gamma] \equiv \mathbb{B} \quad \Gamma \vdash \bot[\gamma] \equiv \bot \quad \Gamma \vdash (\Pi(A, B))[\gamma] \equiv \Pi(A[\gamma], B\left[\gamma^\uparrow\right]) \\
\Gamma \vdash (\Sigma(A, B))[\gamma] \equiv \Sigma(A[\gamma], B\left[\gamma^\uparrow\right]) \quad \Gamma \vdash (\mathtt{Eq}(a, b))[\gamma] \equiv \mathtt{Eq}(a[\gamma], b[\gamma]) \quad \Gamma \vdash \mathbb{S}(a)[\gamma] \equiv \mathbb{S}(a[\gamma])
\end{array}}
$$

$$\frac{a < b : \mathbb{N} \quad \Gamma \vdash_a A}{\Gamma \vdash_b \Uparrow_a^b A} \qquad \text{[TM/LIFTEQ]}\ \Gamma \vdash t : A \Leftrightarrow \Gamma \vdash t : \Uparrow_a^b A \qquad \text{[CON/TY/LIFTEQ]}\ \Gamma, \Uparrow_a^b A \vdash \Leftrightarrow\ \Gamma, A \vdash$$

$$\Gamma \vdash \Uparrow_a^b A[\gamma] \equiv \Uparrow_a^b A[\gamma] \quad \Gamma \vdash \Uparrow_b^c \Uparrow_a^b A \equiv \Uparrow_a^c A \quad (\Gamma, \Uparrow_a^b A) \equiv (\Gamma, A) \vdash$$

$$\frac{m < a}{\Gamma \vdash_b \Uparrow_a^b \mathbb{U}_m \equiv \mathbb{U}_m \quad \Gamma \vdash \Uparrow_a^b \Pi(A,B) \equiv \Pi(\Uparrow_a^b A, \Uparrow_a^b B) \quad \Gamma \vdash \Uparrow_a^b \Sigma(A,B) \equiv \Sigma(\Uparrow_a^b A, \Uparrow_a^b B)}$$
$$\Gamma \vdash \Uparrow_a^b \mathtt{Eq}(x,y) \equiv \mathtt{Eq}(x,y) \quad \Gamma \vdash \Uparrow_a^b \mathbb{S}(x) \equiv \mathbb{S}(x)$$

[TM/CODE]
$$\frac{\Gamma \vdash_j T}{\Gamma \vdash \mathtt{c}(T) : \mathbb{U}_j} \qquad \frac{\Gamma \vdash T : \mathbb{U}_j}{\Gamma \vdash_j \mathtt{El}(T)} \qquad \frac{}{\Gamma \vdash \mathtt{El}(\mathtt{c}(T)) \equiv T} \qquad \frac{}{\Gamma \vdash \mathtt{c}(\mathtt{El}(T)) \equiv T : \mathbb{U}} \qquad \frac{}{\Gamma \vdash () : \top}$$

$$\frac{\Gamma \vdash t : \top}{\Gamma \vdash t \equiv () : \top} \qquad \text{[TM/SUB]}\ \frac{\Delta \vdash t : T \quad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash t[\gamma] : T[\gamma]} \qquad \frac{}{\Gamma \vdash t\left[\mathtt{p}^0\right] \equiv t : T} \qquad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda(t) : \Pi(A,B)}$$

$$\frac{\Gamma \vdash t : \Pi(A,B)}{\Gamma, A \vdash \mathtt{app}(t) : B} \qquad \frac{}{\Gamma, A \vdash \mathtt{app}(\lambda(t)) \equiv t : B} \qquad \frac{}{\Gamma \vdash \lambda(\mathtt{app}(t)) \equiv t : \Pi(A,B)}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[(\mathtt{p}^0, u)]}{\Gamma \vdash (u,v) : \Sigma(A,B)} \qquad \text{[TM/PAIR/PROJ]}\ \frac{\Gamma \vdash t : \Sigma(A,B)}{\Gamma \vdash \mathtt{fst}\, t : A \quad \Gamma \vdash \mathtt{snd}\, t : B[(\mathtt{p}^0, \mathtt{fst}\, t)]}$$

$$\Gamma \vdash \mathtt{fst}\, \langle u, v \rangle \equiv u : A \quad \Gamma \vdash \mathtt{snd}\, \langle u, v \rangle \equiv v : B[(\mathtt{p}^0, u)] \qquad \Gamma \vdash \langle \mathtt{fst}\, t, \mathtt{snd}\, t \rangle \equiv t : \Sigma(A,B)$$

$$\frac{}{\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathbb{B}} \qquad \frac{\Gamma \vdash c : \mathbb{B} \quad \Gamma \vdash a : T \quad \Gamma \vdash b : T}{\Gamma \vdash \mathtt{if}(c,a,b) : T} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathtt{refl}(a) : \mathtt{Eq}(a,a)}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma, A, \mathtt{Eq}(u[\pi_1], \pi_2) \vdash C \quad \Gamma \vdash w : C\left[\mathtt{p}^0, u, \mathtt{refl}(u)\right] \quad \Gamma \vdash v : A \quad \Gamma \vdash t : \mathtt{Eq}(u,v)}{\Gamma \vdash \mathtt{J}(w,t) : C\left[\mathtt{p}^0, v, t\right]}$$

$$\Gamma \vdash \mathtt{if}(\mathtt{tt}, a, b) \equiv a : T \quad \Gamma \vdash \mathtt{if}(\mathtt{ff}, a, b) \equiv b : T \qquad \Gamma \vdash \mathtt{J}(w, \mathtt{refl}(u)) \equiv w : C\left[\mathtt{p}^0, u, \mathtt{refl}(u)\right]$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \star : \mathbb{S}(a)} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash x : \mathbb{S}(a)}{\Gamma \vdash x \equiv a : A}$$

$$\Gamma \vdash (\lambda(t))[\gamma] \equiv \lambda(t\left[\gamma^{\uparrow}\right]) : \Pi(A,B) \quad \Gamma \vdash (u,v)[\gamma] \equiv (u[\gamma], v[\gamma]) : \Sigma(A,B)$$
$$\Gamma \vdash \mathtt{El}(T[\gamma]) \equiv (\mathtt{El}(T))[\gamma] \quad \Gamma \vdash \mathtt{tt}[\gamma] \equiv \mathtt{tt} : \mathbb{B} \quad \Gamma \vdash \mathtt{ff}[\gamma] \equiv \mathtt{ff} : \mathbb{B}$$
$$\Gamma \vdash (\mathtt{if}(c,a,b))[\gamma] \equiv \mathtt{if}(c[\gamma], a[\gamma], b[\gamma]) : T \quad \Gamma \vdash (\mathtt{J}(w,t))[\gamma] \equiv \mathtt{J}(w[\gamma], t[\gamma]) :$$

$$\frac{}{\Gamma \vdash \epsilon : \cdot} \qquad \frac{\Gamma \vdash \gamma : \cdot}{\Gamma \vdash \gamma \equiv \epsilon : \cdot} \qquad \frac{\Delta \vdash \delta : \Theta \quad \Gamma \vdash \gamma : \Delta}{\Gamma \vdash \delta \circ \gamma : \Theta} \qquad \frac{}{\Gamma \vdash \mathtt{id} \equiv \mathtt{p}^0 : \Gamma}$$

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

$$\frac{\Gamma, A_n, ..., A_1, A_0 \vdash}{\Gamma, A_n, ..., A_1, A_0 \vdash \mathtt{var}_n \ : \ A_n\left[\mathtt{p}^{n+1}\right]}\ [\textsc{tm/var}] \qquad \frac{\Gamma \vdash \gamma \ : \ \Delta \quad \Delta \vdash A}{\Gamma, A[\gamma] \vdash \gamma^{\uparrow A} \equiv (\gamma \circ \mathtt{p}^1, \mathtt{var}_0) \ : \ \Delta, A}\ [\textsc{sub/dbj/shift}]$$

$$\frac{\Gamma \vdash \gamma \ : \ (\Delta, A)}{\Gamma \vdash \pi_1\gamma \ : \ \Delta} \qquad \frac{\Gamma \vdash \gamma \ : \ (\Delta, A)}{\Gamma \vdash \pi_2\gamma \ : \ A[\pi_1\gamma]} \qquad \frac{}{\Gamma \vdash (\pi_1\gamma, \pi_2\gamma) \equiv \gamma \ : \ \Delta}$$

$$\Gamma \vdash \gamma_1 \circ (\gamma_2 \circ \gamma_3) \equiv (\gamma_1 \circ \gamma_2) \circ \gamma_3 \ : \ \Theta \qquad \Gamma \vdash \mathtt{p}^0 \circ \gamma \equiv \gamma \circ \mathtt{p}^0 \equiv \gamma \ : \ \Theta$$

Variable using De Bruijn Indices and Explicit Substitution. de Bruijn indices and explicit substitutions make details about binders and substitutions clear. Using explicit substitutions obviates the need for special treatment of substitutions in the meta-theoretical proofs, as substitutions are part of the syntax. The form $\mathtt{var}_n$ represents a 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 γ to terms valid in the context Δ yields terms valid in Γ ([TM/SUB] and [TY/SUB]). The two main forms of substitutions are weakening ([SUB/WK]) and extension ([SUB/EXT]): $t[\mathtt{p}^n]$ introduces n free variables into the context of t, and $t[\gamma, t']$ substitutes t' for $\mathtt{var}_0$ in t and then applies γ. For example, rule [TM/PAIR/PROJ] states that if t is a dependent pair that has type $\Sigma(A, B)$, then $\mathtt{snd}\ t$ has type $B\left[\mathtt{p}^0, \mathtt{fst}\ t\right]$, where $\mathtt{p}^0$ is the identity substitution ([SUB/ID]). We occasionally use the notation $\mathtt{id}$ for $\mathtt{p}^0$.

To simplify, $\mathtt{p}^n$ is a short hand for $\pi_1^n\ \mathtt{id}$ and $\mathtt{var}_n$ is a short hand for $\pi_2\ \pi_1^n$. Thus during meta-theoretic reasoning, we will only deal with π_1 and π_2.

Consequently, function application changes to an equivalent formulation, and becomes a "direct inverse" of typing rule for function abstraction. For example, the named notation $\mathtt{app}(f, t)$ can be equivalently represented by $\mathtt{app}(f)\left[\mathtt{p}^0, t\right]$.

Finally, we have [SUB/DBJ/SHIFT] defined using [SUB/WK] and [TM/VAR]. This rule applying substitution γ to the earlier portion of the context. We usually omit A in the $\gamma^{\uparrow A}$ because it can be inferred from the context.

Judgmental Equality instead of Operational Semantics. Conventional PL formulation usually relies on *operational semantics* to specify the execution of programs. However, in this MLTT formulation [8, 56, 2], we use *judgmental equalities* between two derivations to represent such execution.

The reason for such alternation is the existence of dependent type — especially when we want to consider type $M(1 + 0) \equiv M(1)$ since $1 + 0 \equiv 1$ in arithmetic. If we use operational semantic, then $1 + 0$ and 1 are considered different syntax objects and we need extra handling/rules to *convert* between terms of (different) types $M(1 + 0)$ and $M(1)$. The easiest solution is to introduce *judgmental equality* and thus *quotiented syntax*. And thus $1 + 0$ and 1 are consider the "same" syntax (under the quotient along judgmental equalities) so we have $M(1 + 0) \equiv M(0)$ also the same syntax object under the quotient along judgmental equalities.

Using judgmental equalities to express computational information does raise some concerns (†):

(1) Shifting from operational semantics urges us to find an alternative notion of *type safety*. For example, type safety claims closed boolean terms reduce to values (e.g. $\mathtt{tt}$ and $\mathtt{ff}$) upon termination, proving that we have a **complete** set of reduction rules.

Comparably, what notion do we need to prove to show that we have **enough** judgmental equalities?

(2) Equality is a bidirectional notion, so the execution model is unclear.

We will resolve both concerns after constructing the canonicity model.

Type Connectives. The fundamental two connectives in dependent type theory are dependent functions $\Pi(\cdot,\cdot)$ and dependent pairs $\Sigma(\cdot,\cdot)$. We also have identity types $\mathtt{Eq}(x,y)$ to express, at type level, that term x, y are equal. The elimination principle is the standard J-rule $\mathtt{J}(\cdot,\cdot)$ [60, §1.12]. Moreover, we have singleton type $\mathbb{S}(a)$ with a unique inhabitant, and a boolean type $\mathbb{B}$ with no eliminator.

Universe levels. A (code of a) type has type $\mathbb{U}$. For example, $\Gamma \vdash \mathtt{c}(\top) : \mathbb{U}$, which makes $\mathbb{U}$ another type.

However, it is unsound to say $\Gamma \vdash \mathtt{c}(\mathbb{U}) : \mathbb{U}$, and it is also generally unsound to say that we have a set of all sets (or a universe of all types) [26].

But it is sound to say, we have a large type $\mathbb{U}_0$ classifying all the *small* types, and $\mathbb{U}_1$ classifying all the large types (e.g. $\mathbb{U}_0$ and $\Pi(\mathbb{U}_0, \mathbb{U}_0)$), and etc, to consititue an *infinite hierachy* of universes $\mathbb{U}_0 : \mathbb{U}_1 : \mathbb{U}_2 ....$

This index is *universe level*, which is a natural number used to address the forementioned *size issue*. The level of a universe specifies "how large" that universe is. It is used in both type judgment $\Gamma \vdash_i T$ and universe index $\mathbb{U}_i$. The judgment form $\Gamma \vdash_i T$ indicates that (the code of) the type T inhabits universe $\mathbb{U}_i$. (i.e. [TM/CODE]).

All the type connectives (e.g. $\Pi(,)$ and $\Sigma(,)$) will work on types of the same level. Following Sterling [56], we also introduce explicit universe lifting about the type $\Uparrow_a^b A$ to connect types of different level, which makes the *infinite hierachy* also *cumulative* (i.e. $\vdash T : \mathbb{U}_i$ implies $\vdash \Uparrow_i^{i+1} T : \mathbb{U}_{i+1}$, and they have same term as inhabitants [TM/LIFTEQ]).

Equalities on judgments. With universe lifting, we have one special equality about judgments, adopted from Sterling [56, §4.1]. It is saying that the terms of lifted type is the same as the unlifted (i.e. [TM/LIFTEQ]). Compared to other rule, this rules is special as it is an *equality between two judgments*, while rest of the equalities are all about equalities between derivations.

This rule can be easily formulated in GAT as sort equalties [56, §4.1]. Here we show how to formulate it in QIIT by slightly altering the Agda formulation from Altenkirch and Kaposi [2].

```
data Sort : Set where
data | _ |  : Sort → Set where
  Con' : Sort
  Ty'  : | Con' | → Nat → Sort -- Nat is universe level
  Tms' : | Con' | → | Con' | → Sort
  Tm'  : (Γ : | Con' |) → | Ty' Γ i | → Sort
  -- introduce the lifting and the tm/LiftEq
  ⇑    : (i : Nat) → (j : Nat) →  | Ty' Γ i | → | Ty' Γ j |
  ⇑tmLiftEq : Tm' Γ (⇑ i j T) ≡ Tm' Γ T
  -- an equality between the term of type "Sort"

  -- From here, we re-introduce Con, Ty, Tm and Tms
  Con  : Set
  Con  = | Con' |
  Ty   : Con → Nat → Set
  Ty Γ j  = | Ty' Γ j |
```

```
Tms  : Con → Con → Set
Tms Γ Δ = | Tms' Γ Δ |
Tm   : (Γ : Con) → Ty Γ j → Set
     ≡ (Γ : | Con' |) → | Ty' Γ j | → Set
Tm Γ T  = | Tm' Γ T |
-- Con, Ty, Tm and Tms has exactly the same signature as in
--      (Altenkirch and Kaposi, 2016)
-- so the rest of the rules in (Altenkirch and Kaposi, 2016) can
--      be put here to recover the syntax
```

This encoding works by making the judgment into another derivation on judgment `Sort`. Once they are derivations, we can pose equalties on them. After encoding this synatx rule in QIIT, we are confident about the validity of our syntax rule and we can use the elimination principle of QIIT [2] to reason about our manually-quotiented syntax.

A.2 FMLTT$^{2.0}$ based on MLTT

FMLTT$^{2.0}$ extends the MLTT in Section A.1 with linkage signatures, linkages, linkage transformers, W-type signatures and W-types.

$$
\begin{array}{rcl}
\text{Types} & A,B,T & \ldots \mid \mathtt{w}\pi_1^i(\tau) \mid \mathtt{w}\pi_2^i(\tau) \mid \mathbb{L}(\sigma) \mid \nu\pi_2(\sigma) \mid \mathtt{CaseTy}(A,B,T) \mid \mathbb{L}^+(\sigma) \\
\text{Terms} & t,s,\ell & \ldots \mid \mathtt{W}(\tau) \mid \mathtt{Wsup}_i(\tau,t_1,t_2) \mid \mu^\bullet \mid \mu^+(\ell,t) \mid \mathtt{inh}(h,\ell) \mid \\
 & & \mathtt{Wrec}(\tau,\ell,t) \mid \mu\pi_1(\ell) \mid \mu\pi_2(\ell) \mid \mathtt{R}\pi^i(\ell) \mid \mathtt{iL}^+(n,\sigma,h,\ell) \mid \mathtt{eL}^+(\ell) \\
\text{Contextual Nat} & n & 0 \mid n+1 \mid \ell.\mathtt{size} \\
\text{Linkage signatures} & \sigma & \nu^\bullet \mid \nu^+(\sigma;\ A \vdash T) \mid \nu\pi_1(\sigma) \mid \mathtt{RecSig}(\tau,T) \mid \sigma[\gamma] \mid \ell.\mathtt{sig} \\
\text{Linkage transformers} & h & {}^I\mathtt{Id} \mid {}^I\mathtt{Ext}(h,t) \mid {}^I\mathtt{Ov}^1(h,t) \mid {}^I\mathtt{Inherit}(h) \mid \\
 & & {}^I\mathtt{Nest}(h,h',p) \mid h[\gamma] \mid \ell.\mathtt{inh} \mid h \circ h' \\
\text{Mixin} & \mathtt{h} & h \oplus\!\!\!\!\ominus\ h' \\
\text{W-type signatures} & \tau & \mathtt{w}^\bullet \mid \mathtt{w}^+(\tau,A,B) \mid \tau[\gamma] \mid \mathtt{w}^-(\tau)
\end{array}
$$

$$
\boxed{\Gamma \vdash n\ \mathtt{Nat}} \qquad \boxed{\Gamma \vdash_l \sigma\ \mathtt{LSig}^n} \qquad \boxed{\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2} \qquad \boxed{\mathtt{ProjWk}\ A_2\ A_1 \restriction} \qquad \boxed{\Gamma \vdash_m \tau\ \mathtt{WSig}^N}
$$

$$
\boxed{\mathtt{h} \in \left\{(\sigma_4,h_3,h_4) \;\middle|\; \Gamma \vdash \sigma_4\ \mathtt{LSig}, \Gamma \vdash h_3 : \sigma_2 \twoheadrightarrow \sigma_4, \Gamma \vdash h_4 : \sigma_3 \twoheadrightarrow \sigma_4\right\}}
$$

[WSIG/EMPTY]

$$
\frac{}{\Gamma \vdash_m \mathtt{w}^\bullet\ \mathtt{WSig}^0}
$$

$$
\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^0}{\Gamma \vdash_m \tau \equiv \mathtt{w}^\bullet\ \mathtt{WSig}^0}
$$

[WSIG/ADD]

$$
\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^N \qquad \Gamma \vdash_m A \qquad \Gamma, A \vdash_m B}{\Gamma \vdash_m \mathtt{w}^+(\tau,A,B)\ \mathtt{WSig}^{N+1}}
$$

$$
\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^N \qquad J < N}{\Gamma \vdash_m \mathtt{w}\pi_1^J(\tau) \qquad \Gamma, \mathtt{w}\pi_1^J(\tau) \vdash_m \mathtt{w}\pi_2^J(\tau)}
$$

$$
\frac{\Gamma \vdash \gamma : \Theta}{\begin{array}{c}\Gamma \vdash \mathtt{w}\pi_1^{J+1}(\mathtt{w}^+(\tau,A,B)) \equiv \mathtt{w}\pi_1^J(\tau) \qquad \Gamma, \mathtt{w}\pi_1^{J+1}(\mathtt{w}^+(\tau,A,B)) \vdash \mathtt{w}\pi_2^{J+1}(\mathtt{w}^+(\tau,A,B)) \equiv \mathtt{w}\pi_2^J(\tau) \\ \Gamma \vdash \mathtt{w}\pi_1^0(\mathtt{w}^+(\tau,A,B)) \equiv A \qquad \Gamma, \mathtt{w}\pi_1^0(\mathtt{w}^+(\tau,A,B)) \vdash \mathtt{w}\pi_2^J(\mathtt{w}^+(\tau,A,B)) \equiv B\end{array}}
$$

$$
\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^{N+1}}{\Gamma \vdash_m \mathtt{w}^-(\tau)\ \mathtt{WSig}^N}
$$

$$\frac{\Gamma \vdash \gamma \,:\, \Theta}{\Gamma \vdash \mathtt{w}^{\bullet}[\gamma] \equiv \mathtt{w}^{\bullet}\ \mathtt{WSig}^{0} \quad \Gamma \vdash \mathtt{w}^{+}(\tau, A, B)[\gamma] \equiv \mathtt{w}^{+}(\tau[\gamma], A[\gamma], B\big[\gamma^{\uparrow}\big])\ \mathtt{WSig}^{N+1}}$$
$$\Gamma \vdash \mathtt{w}\pi_1^J(\tau)[\gamma] \equiv \mathtt{w}\pi_1^J(\tau[\gamma]) \quad \Gamma, \mathtt{w}\pi_1^J(\tau[\gamma]) \vdash_i \mathtt{w}\pi_2^J(\tau)\big[\gamma^{\uparrow}\big] \equiv \mathtt{w}\pi_2^J(\tau[\gamma])$$

$$\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^N}{\Gamma \vdash \mathtt{W}(\tau) \,:\, \mathbb{U}_{m+1}} \qquad \frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^N \quad \Gamma \vdash t_1 \,:\, \mathtt{w}\pi_1^I(\tau) \quad \Gamma, \mathtt{w}\pi_2^I(\tau)\big[\mathtt{p}^0, t_1\big] \vdash t_2 \,:\, \mathtt{El}(\mathtt{W}(\tau))}{\Gamma \vdash \mathtt{Wsup}_I(\tau, t_1, t_2) \,:\, \mathtt{El}(\mathtt{W}(\tau))}$$

$$\frac{\Gamma \vdash \gamma \,:\, \Theta}{\Gamma \vdash (\mathtt{Wsup}(T, a, b))[\gamma] \equiv \mathtt{Wsup}(T[\gamma], a[\gamma], b\big[\gamma^{\uparrow}\big]) \,:\, \mathtt{El}((\mathtt{W}(\tau[\gamma])))}$$

$$\frac{\Gamma \vdash_m \tau\ \mathtt{WSig}^N \quad \Gamma \vdash_m A \quad \Gamma, A \vdash_m B}{\Gamma \vdash \mathtt{w}^{-}(\mathtt{w}^{+}(\tau, A, B)) \equiv \tau\ \mathtt{WSig}^N} \qquad \frac{\Gamma \vdash \gamma \,:\, \Theta}{\Gamma \vdash \mathtt{w}^{-}(\tau)[\gamma] \equiv \mathtt{w}^{-}(\tau[\gamma])\ \mathtt{WSig}^N}$$

$$\frac{\Gamma \vdash_l w\ \mathtt{WSig}^N \quad l < l'}{\Gamma \vdash_{l'} \Uparrow_l^{l'} w\ \mathtt{WSig}^N}$$

$$\Gamma \vdash \Uparrow_l^{l'} \mathtt{w}^{+}(\tau, A, B) \equiv \mathtt{w}^{+}(\Uparrow_l^{l'} \tau, \Uparrow_l^{l'} A, \Uparrow_l^{l'} B)\ \mathtt{WSig}^N \quad \Gamma \vdash \mathtt{w}^{-}(\Uparrow_l^{l'} \tau) \equiv \Uparrow_l^{l'} \mathtt{w}^{-}(\tau)\ \mathtt{WSig}^N$$
$$\Gamma \vdash_m \mathtt{w}\pi_1^J(\Uparrow_l^{l'} \tau) \equiv \Uparrow_l^{l'} \mathtt{w}\pi_1^J(\tau) \quad \Gamma, \mathtt{w}\pi_1^J(\Uparrow_l^{l'} \tau) \vdash_m \mathtt{w}\pi_2^J(\Uparrow_l^{l'} \tau) \equiv \Uparrow_l^{l'} \mathtt{w}\pi_2^J(\tau)$$

$$\Gamma \vdash 0\ \mathtt{Nat} \qquad \frac{\Gamma \vdash n\ \mathtt{Nat}}{\Gamma \vdash n + 1\ \mathtt{Nat}}$$

$$\frac{\Delta \vdash n\ \mathtt{Nat} \quad \Gamma \vdash \gamma \,:\, \Delta}{\Gamma \vdash n[\gamma]\ \mathtt{Nat} \quad \Gamma \vdash 0[\gamma] \equiv 0\ \mathtt{Nat} \quad \Gamma \vdash n + 1[\gamma] \equiv n[\gamma] + 1\ \mathtt{Nat}} \qquad \frac{\Gamma \vdash_l \sigma\ \mathtt{LSig}^n}{\Gamma \vdash_l \mathbb{L}(\sigma)}$$

$$\frac{\Delta \vdash_l \sigma\ \mathtt{LSig}^n \quad \Gamma \vdash \gamma \,:\, \Delta}{\Gamma \vdash_l \sigma[\gamma]\ \mathtt{LSig}^n \quad \Gamma \vdash \mathbb{L}(\sigma[\gamma]) \equiv (\mathbb{L}(\sigma))[\gamma]}$$

$$\frac{}{\Gamma \vdash_l \nu^{\bullet}\ \mathtt{LSig}^0} \qquad \frac{\Gamma \vdash_l \sigma\ \mathtt{LSig}^0}{\Gamma \vdash_l \sigma \equiv \nu^{\bullet}\ \mathtt{LSig}^0} \qquad \frac{\Gamma \vdash_l \sigma\ \mathtt{LSig}^n \quad \Gamma, A \vdash_l T}{\Gamma \vdash_l \nu^{+}(\sigma;\ A \vdash T)\ \mathtt{LSig}^{n+1}}\ [\textsc{lsig/add}]$$

$$\frac{\Theta \vdash_l \sigma\ \mathtt{LSig}^{n+1} \quad \Gamma \vdash \gamma \,:\, \Theta}{\Gamma \vdash_l \nu\pi_1(\sigma)\ \mathtt{LSig}^n \quad \Gamma \vdash_l \nu\pi_1'(\sigma) \quad \Gamma, \nu\pi_1'(\sigma) \vdash_l \nu\pi_2(\sigma)}\ [\textsc{lsig/proj}]$$
$$\Gamma \vdash \nu^{+}(\nu\pi_1(\sigma);\ \nu\pi_1'(\sigma) \vdash \nu\pi_2(\sigma)) \equiv \sigma\ \mathtt{LSig}^{n+1}$$

$$\frac{\Gamma \vdash \sigma\ \mathtt{LSig}^n \quad \Gamma \vdash A \quad \Gamma, A \vdash_i T}{\Gamma \vdash \nu\pi_1(\nu^{+}(\sigma;\ A \vdash T)) \equiv \sigma\ \mathtt{LSig}^n \quad \Gamma \vdash \nu\pi_1'(\nu^{+}(\sigma;\ A \vdash T)) \equiv A}$$
$$\Gamma, A \vdash \nu\pi_2(\nu^{+}(\sigma;\ A \vdash T)) \equiv T$$

$$\frac{\Gamma \vdash \gamma \ \mathtt{LSig}^{\Theta}}{\begin{array}{c}\Gamma \vdash \nu^{\bullet}[\gamma] \equiv \nu^{\bullet} \ \mathtt{LSig}^{n} \quad \Gamma \vdash (\nu^{+}(\sigma;\ A \vdash T))[\gamma] \equiv \nu^{+}(\sigma[\gamma];\ A[\gamma] \vdash T\big[\gamma^{\uparrow}\big]) \ \mathtt{LSig}^{n+1} \\ \Gamma \vdash (\nu\pi_1(\sigma))[\gamma] \equiv \nu\pi_1(\sigma[\gamma]) \ \mathtt{LSig}^{n} \quad \Gamma \vdash (\nu\pi_1'(\sigma))[\gamma] \equiv \nu\pi_1'(\sigma[\gamma]) \\ \Gamma, \nu\pi_1'(\sigma[\gamma]) \vdash (\nu\pi_2(\sigma))\big[\gamma^{\uparrow}\big] \equiv \nu\pi_2(\sigma[\gamma])\end{array}}$$

$$\frac{\Gamma \vdash_{l} \sigma \ \mathtt{LSig}^{N} \quad l < l'}{\Gamma \vdash_{l'} \Uparrow_{l}^{l'} \sigma \ \mathtt{LSig}^{n}}$$

$$\begin{array}{c}\Gamma \vdash \Uparrow_{l}^{l'} \mathbb{L}(\sigma) \equiv \mathbb{L}(\Uparrow_{l}^{l'} \sigma) \quad \Gamma \vdash \Uparrow_{l+1}^{l'+1} \mathbb{L}^{+}(\sigma) \equiv \mathbb{L}^{+}(\Uparrow_{l}^{l'} \sigma) \quad \Gamma \vdash_{l'} \Uparrow_{l}^{l'} \nu^{\bullet} \equiv \nu^{\bullet} \ \mathtt{LSig}^{0} \\ \Gamma \vdash_{l'} \Uparrow_{l}^{l'} \nu^{+}(\sigma;\ A \vdash T) \equiv \nu^{+}(\Uparrow_{l}^{l'} \sigma;\ \Uparrow_{l}^{l'} A \vdash \Uparrow_{l}^{l'} T) \ \mathtt{LSig}^{0} \quad \Gamma \vdash_{l'} \Uparrow_{l}^{l'} \nu\pi_1(\sigma) \equiv \nu\pi_1(\Uparrow_{l}^{l'} \sigma) \ \mathtt{LSig}^{n} \\ \Gamma \vdash_{l'} \Uparrow_{l}^{l'} \nu\pi_1'(\sigma) \equiv \nu\pi_1'(\Uparrow_{l}^{l'} \sigma) \quad \Gamma, \Uparrow_{l}^{l'} \nu\pi_1'(\sigma) \vdash_{l} \Uparrow_{l}^{l'} \nu\pi_2(\sigma) \equiv \nu\pi_2(\Uparrow_{l}^{l'} \sigma)\end{array}$$

$$\frac{}{\Gamma \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})} \qquad \frac{\Gamma \vdash \ell : \mathbb{L}(\nu^{\bullet})}{\Gamma \vdash \ell \equiv \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})} \qquad \frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \quad \Gamma, A \vdash t : T}{\Gamma \vdash \mu^{+}(\ell, t) : \mathbb{L}(\nu^{+}(\sigma;\ A \vdash T))}\ [\textsc{l/add}]$$

$$\frac{\Gamma \vdash \ell : \mathbb{L}(\sigma)}{\Gamma \vdash \mu^{+}(\mu\pi_1(\ell), \mu\pi_2(\ell)) \equiv \ell : \mathbb{L}(\sigma)} \qquad \frac{\Gamma \vdash \ell : \mathbb{L}(\sigma)}{\Gamma \vdash \mu\pi_1(\ell) : \mathbb{L}(\nu\pi_1(\sigma)) \quad \Gamma, \nu\pi_1'(\sigma) \vdash \mu\pi_2(\ell) : \nu\pi_2(\sigma)}\ [\textsc{l/proj}]$$

$$\frac{\Gamma \vdash \ell : \mathbb{L}(\sigma) \quad \Gamma \vdash A \quad \Gamma, A \vdash t : T}{\Gamma \vdash \mu\pi_1(\mu^{+}(\ell, t)) \equiv \ell : \mathbb{L}(\sigma) \quad \Gamma, A \vdash \mu\pi_2(\mu^{+}(\ell, t)) \equiv t : T}$$

$$\frac{\Gamma \vdash \gamma : \Theta}{\begin{array}{c}\Gamma \vdash \mu^{\bullet}[\gamma] \equiv \mu^{\bullet} : \mathbb{L}(\nu^{\bullet}) \quad \Gamma \vdash (\mu^{+}(\ell, t))[\gamma] \equiv \mu^{+}(\ell[\gamma], t\big[\gamma^{\uparrow}\big]) : \mathbb{L}(\nu^{+}(\sigma;\ A \vdash T))[\gamma] \\ \Gamma \vdash (\mu\pi_1(\ell))[\gamma] \equiv \mu\pi_1(\ell[\gamma]) : \mathbb{L}(\nu\pi_1(\sigma))[\gamma] \quad \Gamma, \nu\pi_1'(\sigma[\gamma]) \vdash (\mu\pi_2(\ell))\big[\gamma^{\uparrow}\big] \equiv \mu\pi_2(\ell[\gamma]) : \nu\pi_2(\sigma)\end{array}}$$

$$\frac{\Gamma \vdash_{i} A \quad \Gamma, A \vdash_{i} B \quad \Gamma \vdash_{j} T}{\Gamma \vdash_{i \sqcup j} \mathtt{CaseTy}(A, B, R) \equiv \Pi(A, \Pi(\Pi(B, R\big[\mathtt{p}^2\big]), R\big[\mathtt{p}^2\big]))}\ [\textsc{tyeq/casety}]$$

$$\frac{\Gamma \vdash_{m} \tau \ \mathtt{WSig}^{N+1} \quad \Gamma \vdash_{j} R}{\Gamma \vdash_{m \cup j} \mathtt{RecSig}(\tau, R) \equiv \nu^{+}(\mathtt{RecSig}(\mathtt{w}^{-}(\tau), R);\ \pi_2 \vdash \mathtt{CaseTy}(\mathtt{w}\pi_1^0(\tau), \mathtt{w}\pi_2^0(\tau), R)) \ \mathtt{LSig}^{n+1}}$$

$$\frac{\Gamma \vdash_{m} \tau \ \mathtt{WSig}^{0} \quad \Gamma \vdash_{j} R}{\Gamma \vdash_{m \cup j} \mathtt{RecSig}(\tau, R) \equiv \nu^{\bullet} \ \mathtt{LSig}^{0}} \qquad \frac{\Gamma \vdash \tau \ \mathtt{WSig}^{N} \quad \Gamma \vdash \ell : \mathbb{L}(\mathtt{RecSig}(\tau, R)) \quad j < N}{\Gamma \vdash \mathtt{R}\pi^{j}(\ell) : (\mathtt{CaseTy}(\mathtt{w}\pi_1^{j}(\tau), \mathtt{w}\pi_2^{j}(\tau), R))[\pi_1]}$$

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

$$\frac{\Gamma \vdash \tau\ \mathtt{WSig}^N \quad j < N \quad \Gamma \vdash \ell \,:\, \mathbb{L}(\mathtt{RecSig}(\tau, R))}{\Gamma \vdash \mathtt{R}\pi^j(\ell) \,:\, \mathtt{CaseTy}(\mathtt{w}\pi_1^j(\tau), \mathtt{w}\pi_2^j(\tau), R)}$$

$$\Gamma \vdash \mathtt{R}\pi^{n+1}(\ell) \equiv \mathtt{R}\pi^n(\mu\pi_1(\ell)) \,:\, (\mathtt{CaseTy}(\mathtt{w}\pi_1^{n+1}(\tau), \mathtt{w}\pi_2^{n+1}(\tau), R))[\pi_1]$$

$$\Gamma \vdash \mathtt{R}\pi^0(\ell) \equiv \mu\pi_2(\ell)\big[(\mathtt{p}^0, \mathbb{p}(\mu\pi_1(\ell)))\big] \,:\, (\mathtt{CaseTy}(\mathtt{w}\pi_1^0(\tau), \mathtt{w}\pi_2^0(\tau), R))[\pi_1]$$

$$\frac{\Gamma \vdash h \,:\, \mathbb{L}(\mathtt{RecSig}(\tau, R))}{\Gamma \vdash \mathtt{Wrec}(\tau, h, \mathtt{Wsup}_j(\tau, a, b)) \equiv \mathtt{app}(\mathtt{app}(\mathtt{R}\pi^j(h))[(\mathtt{p}^0, a)])[(\mathtt{p}^0, \lambda(\mathtt{Wrec}(\tau, h[\pi_1], b)))] \,:\, R}$$

[PJWK/FORMATION]
$$\frac{\Gamma \vdash A_2 \quad \Gamma \vdash A_1 \quad \Gamma, A_2 \vdash f \,:\, A_1\big[\mathtt{p}^1\big]}{\begin{array}{c}\mathtt{ProjWk}\ A_2\ A_1\ f \cong \\ \{(M, f_1, f_2) \mid \Gamma, A_1 \vdash M, \quad \Gamma, \Sigma(A_1, M) \vdash f_1 \,:\, A_2\big[\mathtt{p}^1\big], \quad \Gamma, A_2 \vdash f_2 \,:\, \Sigma(A_1, M)\big[\mathtt{p}^1\big], \\ f_2\big[\mathtt{p}^1, f_1\big] \equiv \mathtt{var}_1 \quad f_1\big[\mathtt{p}^1, f_2\big] \equiv \mathtt{var}_1 \quad f \equiv \mathtt{fst}\ \mathtt{var}_1\big[\mathtt{p}^1, f_2\big] \quad \}\end{array}}$$

$$\frac{\Gamma \vdash h \,:\, \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash \ell \,:\, \mathbb{L}(\sigma_1)}{\Gamma \vdash \mathtt{inh}(h, \ell) \,:\, \mathbb{L}(\sigma_2)} \qquad \Gamma \vdash \mathtt{inh}(h, \ell)[\gamma] \equiv \mathtt{inh}(h[\gamma], \ell[\gamma]) \,:\, ..$$

$$\Gamma \vdash {}^I\mathtt{Id} \,:\, \sigma \twoheadrightarrow \sigma \qquad \Gamma \vdash {}^I\mathtt{Id}[\gamma] \equiv {}^I\mathtt{Id} \quad \Gamma \vdash \mathtt{inh}({}^I\mathtt{Id}, m) \equiv m \,:\, ..$$

$$\frac{\Gamma \vdash h \,:\, \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma, A_2 \vdash t \,:\, T}{\Gamma \vdash {}^I\mathtt{Ext}(h, t) \,:\, \sigma_1 \twoheadrightarrow (\nu^+(\sigma_2;\ s_2 \vdash T))}$$

$$\Gamma \vdash {}^I\mathtt{Ext}(h, t)[\gamma] \equiv {}^I\mathtt{Ext}(h[\gamma], t\big[\gamma^\uparrow\big]) \quad \Gamma \vdash \mathtt{inh}({}^I\mathtt{Ext}(h, t), \ell) \equiv \mu^+(\mathtt{inh}(h, \ell), t) \,:\, ..$$

$$\frac{\Gamma \vdash h \,:\, \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma, A \vdash T}{\Gamma \vdash {}^I\mathtt{Inherit}(h) \,:\, \nu^+(\sigma_1;\ A \vdash T) \twoheadrightarrow \nu^+(\sigma_2;\ A \vdash T)}$$

$$\Gamma \vdash {}^I\mathtt{Inherit}(h)[\gamma] \equiv {}^I\mathtt{Inherit}(h[\gamma]) \quad \Gamma \vdash \mathtt{inh}({}^I\mathtt{Inherit}(h), \mu^+(\ell, t)) \equiv \mu^+(\mathtt{inh}(h, \ell), t) \,:\, ..$$

$$\frac{\Gamma \vdash h \,:\, \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma, A \vdash T \quad \Gamma, A, T \vdash t \,:\, T[\pi_1]}{\Gamma \vdash {}^I\mathtt{Ov}^1(h, t) \,:\, (\nu^+(\sigma_1;\ A \vdash T)) \twoheadrightarrow (\nu^+(\sigma_2;\ A \vdash T))}$$

$$\Gamma \vdash {}^I\mathtt{Ov}^1(h, t^*)[\gamma] \equiv {}^I\mathtt{Ov}^1(h[\gamma], t\big[\gamma^\uparrow\big]) \quad \Gamma \vdash \mathtt{inh}({}^I\mathtt{Ov}^1(h, t^*), \mu^+(\ell, t)) \equiv \mu^+(\mathtt{inh}(h, \ell), t^*[\mathtt{id}, t]) \,:\, ..$$

$$\frac{\Gamma \vdash h \,:\, \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma, A_1 \vdash t_1 \,:\, T_1 \quad \Gamma, A_1 \vdash t_2 \,:\, T_2}{\Gamma \vdash {}^I\mathtt{Ov}^2(h, t_1, t_2) \,:\, (\nu^+(\sigma_1;\ A_1 \vdash \mathbb{S}(t_1))) \twoheadrightarrow (\nu^+(\sigma_2;\ A_2 \vdash \mathbb{S}(t_2)))}$$

$$\Gamma \vdash {}^{I}\mathtt{Ov}^2(h,t_1,t_2)[\gamma] \equiv {}^{I}\mathtt{Ov}^2(h[\gamma],t_1\big[\gamma^{\uparrow}\big],t_2\big[\gamma^{\uparrow}\big])$$
$$\Gamma \vdash \mathtt{inh}({}^{I}\mathtt{Ov}^2(h,t_1,t_2),\mu^{+}(\ell,t)) \equiv \mu^{+}(\mathtt{inh}(h,\ell),t_2) \;:\; ..$$

[INH/NEST]
$$\frac{\Gamma \vdash h \;:\; \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma, A_1 \vdash \tau_1 \;\; \mathtt{LSig} \qquad \Gamma, A_2 \vdash \tau_2 \;\; \mathtt{LSig} \quad \Gamma, A_2 \vdash \uparrow \;:\; A_1\big[\mathtt{p}^1\big] \quad \mathtt{ProjWk}\; A_2\; A_1\; \uparrow \quad \Gamma, A \vdash h_{inner} \;:\; \tau_1\big[\mathtt{p}^1,\uparrow\big] \twoheadrightarrow \tau_2}{\Gamma \vdash {}^{I}\mathtt{Nest}(h,\uparrow,h_{inner}) \;:\; v^{+}(\sigma_1;\; A_1 \vdash \mathbb{L}(\tau_1)) \twoheadrightarrow v^{+}(\sigma_2;\; A_2 \vdash \mathbb{L}(\tau_2))}$$

$$\Gamma \vdash {}^{I}\mathtt{Nest}(h,\uparrow,h_{inner})[\gamma] \equiv {}^{I}\mathtt{Nest}(h[\gamma],\uparrow\big[\gamma^{\uparrow}\big],h_{inner}\big[\gamma^{\uparrow}\big]) \;:\; v^{+}(\sigma_1;\; A \vdash \mathbb{L}(\tau_1))[\gamma] \twoheadrightarrow v^{+}(\sigma_2;\; A \vdash \mathbb{L}(\tau_2))[\gamma]$$
$$\Gamma \vdash \mathtt{inh}({}^{I}\mathtt{Nest}(h,\uparrow,h_{inner}),\mu^{+}(\ell,t)) \equiv \mu^{+}(\mathtt{inh}(h,\ell),\mathtt{inh}(h_{inner},(t)\big[\mathtt{p}^1,\uparrow\big])) \;:\; ..$$

$$\frac{\Gamma \vdash_l \sigma \;\; \mathtt{LSig}^n}{\Gamma \vdash_{l+1} \mathbb{L}^{+}(\sigma)}$$

[EL-INTRO]
$$\frac{\Gamma \vdash n \;\; \mathtt{Nat} \quad \Gamma \vdash h \;:\; \sigma \twoheadrightarrow \sigma' \quad \Gamma \vdash o \;:\; \mathbb{L}(\sigma)}{\Gamma \vdash \mathtt{iL}^{+}(n,\sigma',h,o) \;:\; \mathbb{L}^{+}(\sigma)}$$

[EL-ELIM]
$$\frac{\Gamma \vdash_l \sigma \;\; \mathtt{LSig}^n \quad \Gamma \vdash o \;:\; \mathbb{L}^{+}(\sigma)}{\Gamma \vdash o.\mathtt{size}\;\; \mathtt{Nat} \quad \Gamma \vdash_l o.\mathtt{sig}\;\; \mathtt{LSig}^{o.\mathtt{size}} \quad \Gamma \vdash o.\mathtt{inh} \;:\; \sigma \twoheadrightarrow o.\mathtt{sig} \quad \Gamma \vdash \mathtt{eL}^{+}(o) \;:\; \mathbb{L}(\sigma)}$$

$$\Gamma \vdash \mathtt{iL}^{+}(n,\sigma',h,o)[\gamma] \equiv \mathtt{iL}^{+}(n[\gamma],\sigma'[\gamma],h[\gamma],o[\gamma]) \;:\; .. \quad \Gamma \vdash o.\mathtt{size}[\gamma] \equiv o[\gamma].\mathtt{size}\;\; \mathtt{Nat}$$
$$\Gamma \vdash_l o.\mathtt{sig}[\gamma] \equiv o[\gamma].\mathtt{sig}\;\; \mathtt{LSig}^{o.\mathtt{size}} \quad \Gamma \vdash \mathtt{iL}^{+}(n,\sigma',h,o).\mathtt{size} \equiv n\;\; \mathtt{Nat}$$
$$\Gamma \vdash o.\mathtt{inh}[\gamma] \equiv o[\gamma].\mathtt{inh} \;:\; \sigma[\gamma] \twoheadrightarrow o.\mathtt{sig}[\gamma] \quad \Gamma \vdash \mathtt{eL}^{+}(o)[\gamma] \equiv \mathtt{eL}^{+}(o[\gamma]) \;:\; ..$$
$$\Gamma \vdash_l \mathtt{iL}^{+}(n,\sigma',h,o).\mathtt{sig} \equiv \sigma'\;\; \mathtt{LSig}^{o.\mathtt{size}} \quad \Gamma \vdash \mathtt{iL}^{+}(n,\sigma',h,o).\mathtt{inh} \equiv h \;:\; \sigma \twoheadrightarrow o.\mathtt{sig}$$
$$\Gamma \vdash \mathtt{eL}^{+}(\mathtt{iL}^{+}(n,\sigma',h,o)) \equiv o \;:\; \mathbb{L}(\sigma)$$

[INH/COMP]
$$\frac{\Gamma \vdash h \;:\; \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h' \;:\; \sigma_2 \twoheadrightarrow \sigma_3}{\Gamma \vdash h \circ h' \;:\; \sigma_1 \twoheadrightarrow \sigma_3}$$

$$\frac{\Gamma \vdash i \;:\; \sigma_1 \twoheadrightarrow \sigma_2}{\Gamma \vdash i \circ {}^{I}\mathtt{Id} \equiv i \quad \Gamma \vdash {}^{I}\mathtt{Id} \circ i \equiv i}$$

$$\frac{\Gamma \vdash i \;:\; \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash j \;:\; \sigma_2 \twoheadrightarrow \sigma_3 \quad \Gamma, A \vdash t \;:\; T}{\Gamma \vdash i \circ {}^{I}\mathtt{Ext}(j,t) \equiv {}^{I}\mathtt{Ext}(i \circ j,t)}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_2 \twoheadrightarrow \sigma_3 \quad \Gamma, A \vdash t : T}{\Gamma \vdash {}^I\mathtt{Ext}(h_1, t) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Ext}(h_1 \circ h_2, t)}$$

$$\begin{gathered}
\Gamma \vdash {}^I\mathtt{Inherit}(h_1) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Inherit}(h_1 \circ h_2) \\
\Gamma \vdash {}^I\mathtt{Ov}^1(h_1, t) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Ov}^1(h_1 \circ h_2, t) \\
\Gamma \vdash {}^I\mathtt{Ov}^2(h_1, t_1, t_2) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_2) \\
\Gamma \vdash {}^I\mathtt{Ov}^2(h_1, t_1, t_2) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_2) \\
\Gamma \vdash {}^I\mathtt{Nest}(h_1, \restriction, i) \circ {}^I\mathtt{Inherit}(h_2) \equiv {}^I\mathtt{Nest}(h_1 \circ h_2, \restriction, i) \\
\Gamma \vdash {}^I\mathtt{Ext}(h_1, t) \circ {}^I\mathtt{Ov}^1(h_2, t^*) \equiv {}^I\mathtt{Ext}(h_1 \circ h_2, t^*[\mathtt{id}, t]) \\
\Gamma \vdash {}^I\mathtt{Inherit}(h_1) \circ {}^I\mathtt{Ov}^1(h_2, t^*) \equiv {}^I\mathtt{Ov}^1(h_1 \circ h_2, t^*) \\
\Gamma \vdash {}^I\mathtt{Ov}^1(h_1, t) \circ {}^I\mathtt{Ov}^1(h_2, t^*) \equiv {}^I\mathtt{Ov}^1(h_1 \circ h_2, t^*[\mathtt{p}^1, t]) \\
\Gamma \vdash {}^I\mathtt{Ov}^2(h_1, t_1, t_2) \circ {}^I\mathtt{Ov}^1(h_2, t^*) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_2) \\
\Gamma \vdash {}^I\mathtt{Nest}(h_1, \restriction, i) \circ {}^I\mathtt{Ov}^1(h_2, t^*) \equiv {}^I\mathtt{Nest}(h_1 \circ h_2, \restriction, i) \\
\Gamma \vdash {}^I\mathtt{Ext}(h_1, t) \circ {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \equiv {}^I\mathtt{Ext}(h_1 \circ h_2, t_2) \\
\Gamma \vdash {}^I\mathtt{Inherit}(h_1) \circ {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_2) \\
\Gamma \vdash {}^I\mathtt{Ov}^1(h_1, t) \circ {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_2) \\
\Gamma \vdash {}^I\mathtt{Ov}^2(h_1, t_1, t_2) \circ {}^I\mathtt{Ov}^2(h_2, t_2, t_3) \equiv {}^I\mathtt{Ov}^2(h_1 \circ h_2, t_1, t_3) \\
\Gamma \vdash {}^I\mathtt{Ext}(h_1, t) \circ {}^I\mathtt{Nest}(h_2, \restriction, i) \equiv {}^I\mathtt{Ext}(h_1 \circ h_2, \mathtt{inh}(i, t)[\mathtt{p}^1, \restriction]) \\
\Gamma \vdash {}^I\mathtt{Inherit}(h_1)t \circ {}^I\mathtt{Nest}(h_2, \restriction, i) \equiv {}^I\mathtt{Nest}(h_1 \circ h_2, \restriction, i) \\
\Gamma \vdash {}^I\mathtt{Ov}^1(h_1, t) \circ {}^I\mathtt{Nest}(h_2, \restriction, i) \equiv {}^I\mathtt{Nest}(h_1 \circ h_2, \restriction, i) \\
\Gamma \vdash {}^I\mathtt{Nest}(h_1, \restriction, i) \circ {}^I\mathtt{Nest}(h_2, \restriction', i') \equiv {}^I\mathtt{Nest}(h_1 \circ h_2, \restriction[\mathtt{p}^1, \restriction'], i[\mathtt{p}^1, \restriction'] \circ i')
\end{gathered}$$

[MIXIN/FORMATION]

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3}{(h_1 \oplus\!\!\!\!\!) \, h_2) \in \left\{ (\sigma_4, h_3, h_4) \;\middle|\; \Gamma \vdash \sigma_4 \;\; \mathtt{LSig}, \Gamma \vdash h_3 : \sigma_2 \twoheadrightarrow \sigma_4, \Gamma \vdash h_4 : \sigma_3 \twoheadrightarrow \sigma_4 \right\}}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3}{{}^I\mathtt{Id} \oplus\!\!\!\!\!) \, h_2 \equiv (\sigma_3, h, {}^I\mathtt{Id}) \quad h_1 \oplus\!\!\!\!\!) \, {}^I\mathtt{Id} \equiv (\sigma_2, {}^I\mathtt{Id}, h)}$$

$$\frac{h_1 \oplus\!\!\!\!\!) \, h_2 = (\tau, hh_2, hh_3)}{\begin{gathered}{}^I\mathtt{Ext}(h_1, t_1) \oplus\!\!\!\!\!) \, h_2 \equiv (\_, {}^I\mathtt{Inherit}(hh_2), {}^I\mathtt{Ext}(hh_3, t_1)) \\ h_2 \oplus\!\!\!\!\!) \, {}^I\mathtt{Ext}(h_1, t_1) \equiv (\_, {}^I\mathtt{Ext}(hh_3, t_1), {}^I\mathtt{Inherit}(hh_2))\end{gathered}}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus\!\!\!\!\!) \, h_2 = (\tau, hh_2, hh_3)}{{}^I\mathtt{Inherit}(h_1) \oplus\!\!\!\!\!) \, {}^I\mathtt{Inherit}(h_2) \equiv (\_, {}^I\mathtt{Inherit}(hh_2), {}^I\mathtt{Inherit}(hh_3))}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus\!\!\!\!\!) \, h_2 = (\tau, hh_2, hh_3)}{\begin{gathered}{}^I\mathtt{Inherit}(h_1) \oplus\!\!\!\!\!) \, {}^I\mathtt{Ov}^1(h_2, t) \equiv (\_, {}^I\mathtt{Ov}^1(hh_2, t), {}^I\mathtt{Inherit}(hh_3)) \\ {}^I\mathtt{Ov}^1(h_2, t) \oplus\!\!\!\!\!) \, {}^I\mathtt{Inherit}(h_1) \equiv (\_, {}^I\mathtt{Inherit}(hh_3), {}^I\mathtt{Ov}^1(hh_2, t))\end{gathered}}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus\!\!\!\!\!) \, h_2 = (\tau, hh_2, hh_3)}{\begin{gathered}{}^I\mathtt{Inherit}(h_1) \oplus\!\!\!\!\!) \, {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \equiv (\_, {}^I\mathtt{Ov}^2(hh_2, t_1, t_2), {}^I\mathtt{Inherit}(hh_3)) \\ {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \oplus\!\!\!\!\!) \, {}^I\mathtt{Inherit}(h_1) \equiv (\_, {}^I\mathtt{Inherit}(hh_3), {}^I\mathtt{Ov}^2(hh_2, t_1, t_2))\end{gathered}}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus h_2 = (\tau, hh_2, hh_3)}{\begin{array}{c} {}^I\mathtt{Inherit}(h_1) \oplus {}^I\mathtt{Nest}(h_2, \upharpoonright, i) \equiv (\_, {}^I\mathtt{Nest}(hh_2, \upharpoonright, i), {}^I\mathtt{Inherit}(hh_3)) \\ {}^I\mathtt{Nest}(h_2, \upharpoonright, i) \oplus {}^I\mathtt{Inherit}(h_1) \equiv (\_, {}^I\mathtt{Inherit}(hh_3), {}^I\mathtt{Nest}(hh_2, \upharpoonright, i)) \end{array}}$$

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus h_2 = (\tau, hh_2, hh_3)}{{}^I\mathtt{Ov}^1(h_1, t_1) \oplus {}^I\mathtt{Ov}^1(h_2, t_2) \equiv (\_, {}^I\mathtt{Ov}^1(hh_2, t_2), {}^I\mathtt{Ov}^1(hh_3, t_2))}$$

$$\frac{\begin{array}{c} \Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \\ h_1 \oplus h_2 = (\tau, hh_2, hh_3) \quad \Gamma, A_1 \vdash \mathbb{S}(t_1) \quad \Gamma, A_2 \vdash \mathbb{S}(t_2) \quad \Gamma, A, \mathbb{S}(t_1) \vdash t_1' : \mathbb{S}(t_1)[\mathtt{p}^1] \end{array}}{\begin{array}{c} {}^I\mathtt{Ov}^1(h_1, t_1') \oplus {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \equiv (\_, {}^I\mathtt{Ov}^2(hh_2, t_1, t_2), {}^I\mathtt{Inherit}(hh_3)) \\ {}^I\mathtt{Ov}^2(h_2, t_1, t_2) \oplus {}^I\mathtt{Ov}^1(h_1, t_1') \equiv (\_, {}^I\mathtt{Ov}^2(hh_3, t_2, t_1), {}^I\mathtt{Inherit}(hh_2)) \end{array}}$$

$$\frac{\begin{array}{c} \Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \\ \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad h_1 \oplus h_2 = (\tau, hh_2, hh_3) \quad \Gamma, A_1 \vdash \mathbb{S}(t_1) \quad \Gamma, A_2 \vdash \mathbb{S}(t_2) \quad \Gamma, A_3 \vdash \mathbb{S}(t_3) \end{array}}{{}^I\mathtt{Ov}^2(h_1, t_1, t_3) \oplus {}^I\mathtt{Ov}^2(h_2, t_1, t_3) \equiv (\_, {}^I\mathtt{Ov}^2(hh_2, t_1, t_3), {}^I\mathtt{Inherit}(hh_3))}$$

$$\frac{\begin{array}{c} \Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3 \quad \Gamma \vdash i_1 : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma \vdash i_2 : \tau_1 \twoheadrightarrow \tau_3 \\ h_1 \oplus h_2 = (\sigma_4, h_3, h_4) \quad (g_1, g_2) = \mathtt{ProjWkMx}(\upharpoonright_1, \upharpoonright_2) \quad i_1[\mathtt{p}^1, g_1] \oplus i_2[\mathtt{p}^1, g_2] = (\tau_4, i_3, i_4) \end{array}}{{}^I\mathtt{Nest}(h_1, \upharpoonright_1, i_1) \oplus {}^I\mathtt{Nest}(h_2, \upharpoonright_2, i_2) \equiv (\_, {}^I\mathtt{Nest}(h_3, g_1, i_3), {}^I\mathtt{Nest}(h_4, g_2, i_4))}$$

[Signature/mixin]

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3}{\mathcal{S}^{\oplus}_{\sigma_1}(h_1, h_2) \coloneqq \Gamma \vdash (h_1 \oplus h_2).{}_0 \ \mathtt{LSig}}$$

[inh/quantified/apply]

$$\frac{\Gamma \vdash \ell : \mathbb{L}^+(\sigma_1) \quad \Gamma \vdash i : \sigma_1 \twoheadrightarrow \sigma_2 \quad \mathcal{M} \coloneqq (\ell.\mathtt{inh} \oplus i)}{\mathtt{inh}^{\oplus}(i, \ell) \coloneqq \Gamma \vdash \mathtt{inh}(\mathcal{M}.{}_1, \mathtt{inh}(\ell.\mathtt{inh}, \mathtt{eL}^+(\ell))) : \mathcal{S}^{\oplus}(\ell.\mathtt{inh}, i)}$$

[Lkg/Concatenation]

$$\frac{\Gamma \vdash \ell_1 : \mathbb{L}^+(\sigma) \quad \Gamma \vdash \ell_2 : \mathbb{L}^+(\sigma)}{\ell_1 \oplus \ell_2 \coloneqq \Gamma \vdash \mathtt{inh}^{\oplus}(\ell_2.\mathtt{inh}, \ell_1) : \mathcal{S}^{\oplus}_{\sigma}(\ell_1.\mathtt{inh}, \ell_2.\mathtt{inh})}$$

[inh/lifting]

$$\frac{\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash i : \sigma_1 \twoheadrightarrow \sigma_3 \quad \mathcal{M} \coloneqq (h \oplus i)}{\mathtt{lift}_h(i) \coloneqq \Gamma \vdash \mathcal{M}.{}_1 : \sigma_2 \twoheadrightarrow \mathcal{S}^{\oplus}_{\sigma_1}(h, i)}$$

[inh/mixin/diagonal]

$$\frac{\Gamma \vdash h_1 : \sigma_1 \twoheadrightarrow \sigma_2 \quad \Gamma \vdash h_2 : \sigma_1 \twoheadrightarrow \sigma_3}{(h_1 \oplus\!\rangle\, h_2) \coloneqq \Gamma \vdash h_1 \circ (h_1 \oplus h_2).{}_1 : \sigma_1 \twoheadrightarrow \mathcal{S}^{\oplus}_{\sigma_1}(h_1, h_2)}$$

Contextual natural numbers. We use natural numbers to index the length of linkage signatures. However, we allow not only literal natural numbers, but also $\Gamma \vdash x.\mathtt{size}\ \ \mathtt{Nat}$ as a dynamic (i.e., contextual) natural number depending on variables in the context. So we introduce a judgment form $\Gamma \vdash n\ \ \mathtt{Nat}$ for *contextual natural numbers*. The judgment form indicates that n is a valid natural-number term in context Γ.

Later, we will introduce W-type signature, also indexed by length. For them, the length is plain natural number N.

Linkage signatures and linkages. Each linkage signature can be considered as a list of pairs of types $\{(A_i, T_i) \mid A_i \vdash T_i\}$ ([LSIG/ADD],[LSIG/PROJ]), where T_i is the type of each field.

Similarly, each linkage can be considered as a list of terms $\{t_i \mid A_i \vdash t_i : T_i\}$ ([L/ADD],[L/PROJ]), directly reflecting the internal representation used in the Rocqet implementation, where each field (e.g., t_i) is stored independently and quantified over the prior fields (e.g., A_i). We can thus call A_i the *abstracted prior fields*.

However, in our implementation of Rocqet, this A_i is largely determined by the prior terms; in our FMLTT$^{2.0}$ design, we do not enforce this restriction on A_i, *for now*. Thus, each linkage is simply a list of *independent* fields, which makes removing and inserting new fields particularly simple in the metatheory.

Later, we will show how, in FMLTT$^{2.0}$, each linkage can be compiled to a module as in Rocqet, realized by a function $\mathbb{L}(\sigma) \to \mathbb{P}(\sigma)$, where $\mathbb{P}(\sigma)$ is a nested Sigma type modeling modules. The restriction on A_i will be the requirement for such compilation procedure.

Linkage transformers. Compared to the Rocqet implementation, we choose a nameless representation for linkage for uniformity with the de Bruijn indices representation in FMLTT$^{2.0}$. However, this nameless representation presents challenges to linkage concatenation, because we have no clue which two fields should be aligned during concatenation.

To encode linkage concatenation, we introduce linkage transformer $\cdot \vdash h : \sigma_1 \twoheadrightarrow \sigma_2$. A linkage transformer directly corresponds to a *trait* in Rocqet. It can be considered as a "delta" linkage or a functor—given a parent linkage $\ell : \mathbb{L}(\sigma_1)$, the linkage transformer h can enrich ℓ to get the derived linkage $\mathtt{inh}(h, \ell)$. Just like functors, linkage transformers can be composed sequentially $h_1 \circ h_2$ (i.e. [INH/COMP]). Just like traits, linkage transformers can also be mixed $h_3 \oplus\!\!\!\!\!\!\cdot\;\; h_4$ (i.e. [MIXIN/FORMATION]).

To support $\circ$ and $\oplus\!\!\!\!\!\!\cdot$, linkage transformers are inductively constructed, by

- ${}^I\mathtt{Id}$ identity,
- ${}^I\mathtt{Ext}(\cdot,\cdot)$ field extension,
- ${}^I\mathtt{Ov}^1(\cdot,\cdot)$ type-preserving overriding,
- ${}^I\mathtt{Ov}^2(\cdot,\cdot,\cdot)$ singleton overriding, and
- ${}^I\mathtt{Nest}(\cdot,\cdot,\cdot)$ nested linkage transformer.

We provide two kinds of overriding operation. ${}^I\mathtt{Ov}^1(\cdot,\cdot)$ can override term but cannot change type; ${}^I\mathtt{Ov}^2(,,)$ can alter the type in the signature but only works with singleton types. ${}^I\mathtt{Nest}(\cdot,\cdot,\cdot)$ is used when nested family needs inheritance.

When designing the computation behavior of $\circ$ and $\oplus\!\!\!\!\!\!\cdot$, we have these two properties in mind:

- If $h_1 \circ h_2 = h_3$, then $\mathtt{inh}(h_2, \mathtt{inh}(h_1, o)) \equiv \mathtt{inh}(h_1 \circ h_2, o)$.
- If $h_1 \oplus\!\!\!\!\!\!\cdot\;\; h_2 = (\_, h_3, h_4)$, then $h_1 \circ h_3 \equiv h_2 \circ h_4$ constitutes a commutative diagram.

But apparently these two properties *do not hold generally*; they hold only in some places. Some weird behaviors in $\circ$ and $\oplus\!\!\!\!\!\!\cdot$ are simply the incompetence of fulfilling these two extensional property.

Finally, the notation of $\oplus\!\!\!\!\!\!\cdot$ should hint on its asymmetry, especially when overriding is involved. For example, when $h_1 \oplus\!\!\!\!\!\!\cdot\;\; h_2$ has ${}^I\mathtt{Ov}^1(\cdot,\cdot)$ in both h_1 and h_2, the final result will take h_2's overriding.

Existential linkages and linkage concatenation. Consider a linkage transformer $\cdot \vdash h : \nu^\bullet \twoheadrightarrow \sigma$. h is quite limited as it can only be applied to the empty linkage $\mathtt{inh}(h, \mu^\bullet)$ and give us $\mathbb{L}(\sigma)$.

We actually want to apply it to arbitrary linkage $\mathbb{L}(\sigma')$, and after the application, we want to get, intuitively speaking, $\mathbb{L}(\sigma' + \sigma)$.

Such a *quantified inheritance* operation can be encoded as the result of mixin composition $\oplus\!\!\!\!\!\!\diagup$. Imagine a term of $\mathbb{L}(\sigma')$ as a linkage transformer $\cdot \vdash h' : \nu^\bullet \twoheadrightarrow \sigma'$. Then the desired $\mathbb{L}(\sigma' + \sigma)$ is the result of $h' \oplus\!\!\!\!\!\!\diagup\ h$.

To streamline this operation, we introduce a new type $\mathbb{L}^+(\tau)$, which is basically an *existential linkage* that packs a signature τ' and a linkage transformer $\vdash ? : \tau \twoheadrightarrow \tau'$ ([eL-intro], [eL-elim]). This new type and the mixin operation $\oplus\!\!\!\!\!\!\diagup$ can derive multiple helper functions, including *linkage concatenation* [Lkg/Concatenation], *quantified inheritance application* [inh/quantified/apply], [inh/lifting], and etc.

Nested inheritance and projective weakening. ${}^I\texttt{Nest}(h_0, \restriction, h_1)$ has the most involved introduction rule among all the rules of linkage transformer. It requires $\restriction$ to be a *projective weakening* $\texttt{ProjWk}\restriction$, which is essential to make ${}^I\texttt{Nest}(h_0, \restriction, h_1)$ support weakening from A_2 to A_1 in [inh/nest], and supports mixin and composition of nested inheritance at the same time.

$\texttt{ProjWk}\ A_2\ A_1\ f$ (e.g. [PJWK/formation]) is another judgment. It says a term $\Gamma, A_2 \vdash f : A_1\left[\mathtt{p}^1\right]$ is *Projective weakening* if $A_2 \cong \Sigma(A_1, M)$ for some type M, and f is equivalent to projecting A_1 out of $\Sigma(A_1, M)$. Those complicated rules $f_2\left[\mathtt{p}^1, f_1\right] \equiv \mathtt{var}_1, f_1\left[\mathtt{p}^1, f_2\right] \equiv \mathtt{var}_1$ is simply a native way to say $A_2 \cong \Sigma(A_1, M)$. [PJWK/formation] is written in a way to implicitly specify that the derivation of $\texttt{ProjWk} \cdots$ is isomorphic to three derivations of other judgments. Here, we are implicit about the introduction rules, elimination rules, or computation rules—we simply acknowledge their existence and use them when needed.

Now we will prove a theorem, to justify the composition rule and mixin rule between two linkage transformers in our syntax rules:

Theorem 1 ($\texttt{ProjWk} \cdots$ preserves composition and mixin).

Comp $(H_1 : \texttt{ProjWk}\ A_2\ A_1\ f_1)$ and $(H_2 : \texttt{ProjWk}\ A_3\ A_2\ f_2)$ implies $\texttt{ProjWk}\ A_3\ A_1\ f_1\left[\mathtt{p}^1, f_2\right]$

Mixin Given $(h_2 : \texttt{ProjWk}\ A_2\ A_1\ f_2)$ and $(h_3 : \texttt{ProjWk}\ A_3\ A_1\ f_3)$,
we have a function $\texttt{ProjWkMx}(f_2, f_3) = (g_2, g_3)$,
s.t. $\texttt{ProjWk}\ \Sigma(A_1, h_2.M \times h_3.M)\ A_2\ g_2$
and $\texttt{ProjWk}\ \Sigma(A_1, h_2.M \times h_3.M)\ A_2\ g_2$
and $f_2\left[\mathtt{p}^1, g_2\right] \equiv f_3\left[\mathtt{p}^1, g_3\right]$
(Apparently, $h_2.M$ is the chosen M specified in the $\texttt{ProjWk}\ A_2\ A_1\ f_2$)

Proof. To prove *Comp*, we simply notice that $A_3 \cong \Sigma(\Sigma(A_1, H_1.M), H_2.M) \cong \Sigma(A_1, \Sigma(H_1.M, H_2.M))$. We omit the verification of equalties.

To prove *Mixin*, the definition of g_2 can be deduced from $\Gamma, \Sigma(A_1, h_2.M \times h_3.M) \vdash \ldots : \Sigma(A_1, h_2.M)$ as $A_2 \cong \Sigma(A_1, h_2.M)$. Similarly for the definition of g_3. Then we simply compute and verify all equations. □

With this theorem, the computation rule of ${}^I\texttt{Nest}(\cdot,\cdot,\cdot) \circ {}^I\texttt{Nest}(\cdot,\cdot,\cdot)$ and ${}^I\texttt{Nest}(\cdot,\cdot,\cdot) \oplus\!\!\!\!\!\!\diagup\ {}^I\texttt{Nest}(\cdot,\cdot,\cdot)$ makes sense.

W-types and W-type signatures encoding inductive types. We follow the same W-type formulation as Jin et al. [27].

W-types [38] are a succinct way to model inductive types in MLTT. Together with the identity type $\texttt{Eq}(\cdot,\cdot)$, they can express a whole host of inductive types [25] with multiple constructors.

We alter the formulation of W-types to get closer to the inductive facility in Rocq. The list of constructors of an inductive type in Rocq corresponds to a *W-type signature*—a signature $\Gamma \vdash \tau\ \texttt{WSig}^n$ consititues of n pairs of types $\left\{ (A_i, B_i) \;\middle|\; A_i \vdash B_i \right\}_{i \in n}$ ([wsig/empty] and [wsig/add]),

each pair modeling one constructor of the inductive type. We can project the i-th pair of types of the signature τ using the rules $\Gamma \vdash \mathtt{w}\pi_1^i(\tau)$ and $\Gamma \vdash \mathtt{w}\pi_2^i(\tau) : \mathtt{w}\pi_1^i(\tau) \to \mathbb{U}$.

For each pair of types identifying a constructor, the first type models the non-inductive arguments of the constructor, and the second type models the *arity* of the inductive arguments. We use a concrete example to show their correspondence: the inductive type with four constructors (`tm_unit`, `tm_var`, `tm_abs` and `tm_app`) is modeled by W-type $\tau_{\mathtt{tm}}$ in the following way, where $\mathbb{0}$ is the bottom type $\bot$, $\mathbb{1}$ the unit type $\top$, $\mathbb{2}$ the boolean type, and $T_{\mathtt{id}}$ a type encoding of `id`:

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

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

W-type elimination via recursion. W-types are eliminated with the form $\mathtt{Wrec}(\tau, \ell, t)$, where t is of a W-type $\mathtt{El}(\mathtt{W}(\tau))$, and ℓ is essentially an n-tuple of case handlers for the n constructors in τ ([TM/WREC]).

Each case handler has a type of the form $\mathtt{CaseTy}(A, B, T)$, where T is the motive of the recursion ([TYEQ/CASETY]); for simplicity, we model only non-dependent motives. The collection of case handlers ℓ encodes those defined and inherited by an `FRecursion` command in Rocqet.

We organize case handlers as a linkage for code reuse and composition—when the inductive type (W-type) is inherited and combined from different parent families, the corresponding case handlers from the parent families can be mixed together to recover a complete recursive function.

A.3 A Proof-Relevant Logical-Relations Model for Canonicity

Now we prove canonicity (and consistency) for FMLTT$^{2.0}$ using a logical-relations model. We follow the reducibility argument of Kaposi et al. [29], Coquand [8], and Sterling [56] to construct our model.

The elimination principle in use to construct these prior models are based on QIITs, categories with families [15], and the generalized algebraic theory [6], respectively. Without exposing the reader to too many technical details, our metalanguage should be understood as an instance of any of the above logical frameworks—the difference is that quotienting is manual in our formulation, whereas it is automatic within those logical frameworks. Thus, our elimination principle will be formulated as a simple induction on the derivation tree, but formally speaking, theoretically supported by the elimination principle of QIIT.

We state the canonicity theorem first:

Theorem 2 (Canonicity). If $\cdot \vdash t : \mathbb{B}$, then either $\cdot \vdash t \equiv \mathtt{tt} : \mathbb{B}$ or $\cdot \vdash t \equiv \mathtt{ff} : \mathbb{B}$.

Canonicity is a key criterion for a dependent type theory to be considered as a programming language or as a computational foundation for mathematics. Since if this theorem is proven in a computable metalogic, then by the Curry—Howard correspondence, the canonicity theorem provides a big-step interpreter for closed terms of the boolean type.

Canonicity theorem is about **characterizing well-typed closed terms**. The canonicity model is trying to prove this characterization holds on all possible closed terms, by induction on derivation.

A.3.1 Canonicity Model of the MLTT fragment. We start with interpreting the MLTT fragment.

First, we need the mathematical setup to interpret universe levels, following Sterling [56, Assumption 5.2]:

Assumption A.1 (Set-theoretic Universe Assumption). *We assume a transfinite hierarchy of Grothendieck universes* $\mathtt{Set}_i$ *for* $i \in \{0, 1, ..., \omega\}$ *in our ambient metalogic.*

This assumption is setting up a powerful set-theoretic environment for us. We can roughly consider each Grothendieck universe $\mathtt{Set}_i$ as the $\mathtt{Set}_i$ in Agda:

- Each $\mathtt{Set}_i$ is closed under dependent function types and dependent pair types. For example, later, for our interpretation of dependent function types, when we have denotations $[\![A]\!]_C, [\![B]\!]_C \in \mathtt{Set}_i$, we will have $[\![\Pi(A,B)]\!]_C \in \mathtt{Set}_i$.
- The universe hierarchy is cumulative, as $\mathtt{Set}_i \in \mathtt{Set}_{i+1}$ and $\mathtt{Set}_i \subseteq \mathtt{Set}_{i+1}$.
- Thus, if $[\![A]\!]_C \in \mathtt{Set}_i$, $[\![B]\!]_C \in \mathtt{Set}_j$, we will have $[\![\Pi(A,B)]\!]_C \in \mathtt{Set}_i \cup \mathtt{Set}_j = \mathtt{Set}_{i \sqcup j}$.

Like most logical-relations proofs, we interpret each judgment and inductively interpret each derivation. We are working in an intrinsic setting; thus, even if we omit contexts for brevity, each syntax piece is actually still well-typed.

Our canonicity model for the base MLTT fragment follows the constructions in Coquand [8] and Sterling [56] in that it utilizes the facilities of the ambient metalogic; for example, we use dependent functions and dependent tuples in the ambient metalogic $\mathtt{Set}_i$ to interpret dependent functions and tuples in FMLTT$^{2.0}$.

$$
\begin{aligned}
[\![\Gamma \vdash]\!]_C \text{ is a function} &: \left\{ \gamma \;\middle|\; \cdot \vdash \gamma : \Gamma \right\} \to \mathtt{Set}_\omega \\
&\quad \text{(i.e., sets indexed by closed substitution)} \\
[\![\Gamma \vdash_j T]\!]_C \text{ is a dependent function} &: \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma' \in [\![\Gamma\vdash]\!]_C(\gamma)} \left\{ t \;\middle|\; \cdot \vdash t : T[\gamma] \right\} \to \mathtt{Set}_j \\
[\![\Gamma \vdash \delta : \Delta]\!]_C \text{ is a dependent function} &: \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma' \in [\![\Gamma\vdash]\!]_C(\gamma)} [\![\Delta \vdash]\!]_C(\delta \circ \gamma) \\
[\![\Gamma \vdash t : T]\!]_C \text{ is a dependent function} &: \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma' \in [\![\Gamma\vdash]\!]_C(\gamma)} [\![\Gamma \vdash T]\!]_C(\gamma)(\gamma')(t[\gamma]) \\
[\![\Gamma \vdash T[\sigma]]\!]_C(\gamma)(\gamma')(t) &= [\![T]\!]_C(\sigma \circ \gamma)([\![\sigma]\!](\gamma)(\gamma'))(t) \\
[\![\Gamma \vdash \Uparrow_a^b T]\!]_C(\gamma)(\gamma')(t) &= [\![T]\!]_C(\gamma)(\gamma')(t) \\
[\![\Gamma \vdash \top]\!]_C(\gamma)(\gamma')(t) &= \{\star\} \quad \text{a singleton set} \\
[\![\Gamma \vdash \bot]\!]_C(\gamma)(\gamma')(t) &= \emptyset \\
[\![\Gamma \vdash \mathbb{B}]\!]_C(\gamma)(\gamma')(t) &= \begin{cases} \{\star^1\} & \text{if } t \equiv \mathtt{tt} \\ \{\star^2\} & \text{if } t \equiv \mathtt{ff} \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash \mathtt{Eq}(a,b)]\!]_C(\gamma)(\gamma')(t) &= \begin{cases} \{\star\} & \text{if } t \equiv \mathtt{refl}(a[\gamma]) \text{ and } a[\gamma] \equiv b[\gamma] \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash \Pi(A,B)]\!]_C(\gamma)(\gamma')(t) &= \prod_{\cdot\vdash u:A[\gamma]} \prod_{u' \in [\![A]\!]_C(\gamma)(\gamma')(u)} [\![B]\!]_C(\gamma,u)((\gamma',u'))(\mathtt{app}(t)[\mathtt{id},u]) \\
[\![\Gamma \vdash \Sigma(A,B)]\!]_C(\gamma)(\gamma')(t) &= \sum_{u' \in [\![A]\!]_C(\gamma)(\gamma')(\mathtt{fst}\ t)} [\![B]\!]_C(\gamma, \mathtt{fst}\ t)((\gamma',u'))(\mathtt{snd}\ t) \\
[\![\cdot \vdash]\!]_C(\gamma) &= \{\star\} \\
[\![\Gamma, T \vdash]\!]_C(\gamma_t) &= \left\{ (\gamma', t') \;\middle|\; \gamma' \in [\![\Gamma \vdash]\!]_C(\pi_1\gamma_t), t' \in [\![\cdot \vdash T[\gamma]]\!]_C(\pi_1\gamma_t)(\gamma')(\pi_2\gamma_t) \right\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \pi_1 \delta : \Delta]\!]_C(\gamma)(\gamma') &= [\![\delta]\!]_C(\gamma)(\gamma')[0] \quad \text{get the first element of the tuple} \\
[\![\Gamma \vdash \pi_2 \delta : \Delta]\!]_C(\gamma)(\gamma') &= [\![\delta]\!]_C(\gamma)(\gamma')[1] \quad \text{get the second element of the tuple} \\
[\![\Gamma \vdash \delta, t : \Delta]\!]_C(\gamma)(\gamma') &= ([\![\delta]\!]_C(\gamma)(\gamma'), [\![t]\!]_C(\gamma)(\gamma')) \\
[\![\Gamma \vdash \mathtt{id} : \Gamma]\!]_C(\gamma)(\gamma') &= \gamma' \\
[\![\Gamma \vdash \epsilon : \cdot]\!]_C(\gamma)(\gamma') &= \star \\
[\![\Gamma \vdash \delta_1 \circ \delta_2 : \Delta]\!]_C(\gamma)(\gamma') &= [\![\delta_1]\!]_C(\delta_2 \circ \gamma)([\![\delta_2]\!]_C(\gamma)(\gamma')) \\
[\![\Gamma \vdash t[\sigma] : T[\sigma]]\!]_C(\gamma)(\gamma') &= [\![t]\!]_C(\sigma \circ \gamma)([\![\sigma]\!]_C(\gamma)(\gamma')) \\
[\![\Gamma \vdash () : \top]\!]_C(\gamma)(\gamma') &= \star \\
[\![\Gamma \vdash \mathtt{tt} : \mathbb{B}]\!]_C(\gamma)(\gamma') &= \star^1 \\
[\![\Gamma \vdash \mathtt{ff} : \mathbb{B}]\!]_C(\gamma)(\gamma') &= \star^2 \\
[\![\Gamma \vdash \mathtt{if}(c, a, b) : T]\!]_C(\gamma)(\gamma') &= \begin{cases} [\![a]\!]_C(\gamma)(\gamma') & \text{if } [\![c]\!](\gamma)(\gamma') = \star^1 \\ [\![b]\!]_C(\gamma)(\gamma') & \text{if } [\![c]\!](\gamma)(\gamma') = \star^2 \end{cases} \\
[\![\Gamma \vdash \mathtt{refl}(t) : \mathtt{Eq}(t,t)]\!]_C(\gamma)(\gamma') &= \star \\
[\![\Gamma \vdash \mathtt{J}(w, t) : C[id, v, t]]\!]_C(\gamma)(\gamma') &= [\![w]\!]_C(\gamma)(\gamma') \qquad \text{given } \Gamma \vdash t : \mathtt{Eq}(u, v) \\
&\qquad\qquad \text{Since } [\![t]\!]_C(\gamma)(\gamma') \text{ witnesses } t[\gamma] \equiv \mathtt{refl}(u[\gamma]) \text{ and } u[\gamma] \equiv v[\gamma] \\
[\![\Gamma \vdash \lambda(t) : \Pi(A, B)]\!]_C(\gamma)(\gamma') &= \lambda u \lambda u'.[\![t]\!]_C(\gamma, u)(\gamma', u') \\
[\![\Gamma \vdash \mathtt{app}(t) : B]\!]_C(\gamma)(\gamma') &= [\![t]\!]_C(\pi_1 \gamma)(\gamma'[0])(\pi_2 \gamma)(\gamma'[1]) \\
[\![\Gamma \vdash \langle a, b \rangle : \Sigma(A, B)]\!]_C(\gamma)(\gamma') &= ([\![a]\!]_C(\gamma)(\gamma'), [\![b]\!]_C(\gamma)(\gamma')) \\
[\![\Gamma \vdash \mathtt{fst}\, t : T]\!]_C(\gamma)(\gamma') &= [\![t]\!]_C(\gamma)(\gamma')[0] \quad \text{extract the first element in the tuple} \\
[\![\Gamma \vdash \mathtt{snd}\, t : T]\!]_C(\gamma)(\gamma') &= [\![t]\!]_C(\gamma)(\gamma')[1] \\
[\![\Gamma \vdash_{j+1} \mathbb{U}_j]\!]_C(\gamma)(\gamma')(T) &= \left\{ t \,\middle|\, \cdot \vdash t : \mathtt{El}(T) \right\} \to \mathtt{Set}_j \\
[\![\Gamma \vdash \mathtt{c}(T) : \mathbb{U}_j]\!]_C(\gamma)(\gamma') &= [\![T]\!]_C(\gamma)(\gamma') \\
[\![\Gamma \vdash_j \mathtt{El}(T)]\!]_C(\gamma)(\gamma')(t) &= [\![T]\!]_C(\gamma)(\gamma')(t)
\end{aligned}
$$

Here, $\star$, $\star^1$, $\star^2$ are just some arbitrary fixed elements. Our interpretation respect all the equations in the syntax, especially [TM/LiftEq] because the interpretation of $[\![\Uparrow_a^b T]\!]_C = [\![T]\!]_C$.

Now that the canonicity model of MLTT is constructed, we can prove the fundamental theorem of MLTT, which corresponds to the fundamental lemma in a conventional logical relation proof.

Theorem 3 (Fundamental Theorem for MLTT fragement). If $\Gamma \vdash t : T$, then its semantic interpretation is a dependent function such that $[\![t]\!]_C : \prod_{\cdot \vdash \gamma : \Gamma} \prod_{\gamma' \in [\![\Gamma \vdash]\!]_C(\gamma)} [\![\Gamma \vdash T]\!]_C(\gamma)(\gamma')(t[\gamma])$.

A.3.2 Comparison to conventional Logical Relation Proof. This MLTT canonicity model actually shares a great similarity with the classical logical relation style termination/type-safety proof of STLC [55, §3.2], formulated in operational semantic. We will point out the similarity for better understanding of this MLTT canonicity model.

(1) Both proofs are carried out by an inductive interpretation/denotation of derivation—both proofs start with interpretation of types, contexts, and then terms.

(2) Both proofs use a set of closed terms as type denotations.
(3) Both proofs use a list of closed terms/substitutions as context denotation.
(4) Both proofs share similarity on interpretation of terms—with context denotations substituted, they belong to type denotations.

The biggest difference between the two proofs is that Skorstengaard [55, §3.2] uses *proof-irrelevant* logical relations while our MLTT canonicity model uses *proof-relevant* logical relations. "Proof-irrelevance" means, for the former one, term interpretation is merely a *proposition* (we cite from Skorstengaard [55, §3.2])

$$\Gamma \vDash e : \tau \coloneqq \forall \gamma \in \mathscr{G}[\![\Gamma]\!], \gamma(e) \in \mathscr{E}[\![\tau]\!]$$

while for the latter one, term interpretation is a *function*, which can also classify other values. For example $[\![\Gamma \vdash \mathsf{c}(\mathbb{B}) : \mathbb{U}]\!]_C$ is a function $[\![\Gamma \vdash \mathbb{B}]\!]_C$ that maps to a binary set. Compared to the proposition, this is more like a piece of data that carry sets. Coquand [8] points out this strategy works well with an infinite universe hierarchy and allows concise proofs for type theory with universes.

A.3.3 Consistency and Canonicity Theorems for the MLTT fragment. The first consequence of Theorem 3 is the consistency of the MLTT fragment—we cannot derive $\cdot \vdash t : \bot$. Otherwise, we would have an element in the empty set, $[\![\cdot \vdash t : \bot]\!]_C(\epsilon)(\star) \in [\![\bot]\!]_C(\epsilon)(\star)(t[\gamma]) = \emptyset$, a contradiction.

Theorem 4 (Consistency). The typing judgment $\cdot \vdash t : \bot$ is not derivable for any term t.

Next, let's use Theorem 3 on an arbitrary term $\cdot \vdash t : \mathbb{B}$, then we have $[\![t]\!]$ that witness

$$\prod_{\cdot \vdash \gamma : \cdot} \prod_{\gamma' \in [\![\cdot \vdash]\!]_C(\gamma)} [\![\Gamma \vdash \mathbb{B}]\!]_C(\gamma)(\gamma')(t[\gamma]) \quad \Longleftrightarrow \quad t \in \left\{ x \;\middle|\; x \equiv \mathtt{tt} \text{ or } x \equiv \mathtt{ff} \right\}$$

Theorem 5. For arbitrary closed boolean term t, it is (judgmentally) equal to either `tt` or `ff`.

To some extent, the canonicity theorem justifies MLTT as a basic type-safe programming language, resolving the aforementioned two concerns (†):

(1) If we consider judgmental equality as reduction rules, then the canonicity theorem is saying that well-typed boolean program never gets stuck and actually terminates. This also means that we have a *complete* set of judgmental equalities.
(2) Our canonicity proof is constructive. If this proof is mechanically formalized in a constructive proof assistant, then under Curry–Howard correspondence, our theorem can act as a (big-step) program interpreter that can execute arbitrary closed boolean program.

The above two theorems are essential properties for any dependent type theory. Now, we will aim to extend this canonicity model to our FMLTT$^{2.0}$, at the same time making sure that these two properties still hold.

A.3.4 Dependent Function, Dependent Pair, and Type Connectives in the MLTT Canonicity Model. Given the above model for MLTT, there should be a function Π^c such that $\Pi^c([\![A]\!]_C, [\![B]\!]_C) = [\![\Pi(A, B)]\!]_C$. Type-theoretically speaking, this Π^c is the *internal dependent function type* of the above model. We hope to use this function as a helper when defining the logical-relations model for the rest of FMLTT$^{2.0}$. However, such a function Π^c is not yet possible because the definition $[\![\Pi(A, B)]\!]_C$ is not based solely on $[\![A]\!]_C$ and $[\![B]\!]_C$, but also on the syntax $\Gamma \vdash A$ and $\Gamma, A \vdash B$.

Thus, we define a new denotation $[\![S]\!]^\bullet_C := (S, [\![S]\!]_C)$ that also returns the syntax piece S.[6] Then, we can have a function $\Pi^\bullet$ such that $\Pi^\bullet([\![A]\!]^\bullet_C, [\![B]\!]^\bullet_C) = [\![\Pi(A,B)]\!]^\bullet_C$ now that the syntax is available. Similarly, there are functions $\Sigma^\bullet$, $(a,^\bullet b)$, and $\gamma,^\bullet t$ for dependent pair types, dependent pairs, and substitution extension (and more for other constructions).

Now, when given $S^\bullet$ (i.e., the syntax and its semantic interpretation), we use S or $(S^\bullet)\downarrow$ to mean the former (syntax part) and we use $S^\blacktriangleright$ or $(S^\bullet)^\blacktriangleright$ to mean the latter (the semantic part).

We need more *internal* type-theoretic constructions, as helper functions. We define the following:

$$\mathtt{Con}^\blacktriangleright\ \Gamma := \left\{\gamma \,\middle|\, \cdot \vdash \gamma\ :\ \Gamma\right\} \to \mathtt{Set}_\omega \qquad \mathtt{Con}^\bullet := \sum_{\Gamma\vdash} \mathtt{Con}^\blacktriangleright\ \Gamma$$

$$\mathtt{Ty}^\blacktriangleright_j\ \Gamma^\bullet\ T := \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma'\in\Gamma^\blacktriangleright(\gamma)} \left\{t \,\middle|\, \cdot \vdash t\ :\ T[\gamma]\right\} \to \mathtt{Set}_j \qquad \mathtt{Ty}^\bullet{}_j\ \Gamma^\bullet := \sum_{\Gamma\vdash_j T} \mathtt{Ty}^\blacktriangleright_j\ \Gamma^\bullet\ T$$

$$\mathtt{Tm}^\blacktriangleright\ \Gamma^\bullet\ T^\bullet\ t := \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma'\in\Gamma^\blacktriangleright} T^\blacktriangleright(\gamma)(\gamma')(t[\gamma]) \qquad \mathtt{Tm}^\bullet\ \Gamma^\bullet\ T^\bullet := \sum_{\Gamma\vdash t:T} \mathtt{Tm}^\blacktriangleright\ \Gamma^\bullet\ T^\bullet\ t$$

$$\mathtt{Sub}^\blacktriangleright\ \Gamma^\bullet\ \Delta^\bullet\ \delta := \prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma'\in\Gamma^\blacktriangleright(\gamma)} \Delta^\blacktriangleright(\delta\circ\gamma) \qquad \mathtt{Sub}^\bullet\ \Gamma^\bullet\ \Delta^\bullet := \sum_{\Gamma\vdash\delta:\Delta} \mathtt{Sub}^\blacktriangleright\ \Gamma^\bullet\ \Delta^\bullet\ \delta$$

These sets are collecting the syntax and semantic interpretation together. For each well-formed type $\Gamma \vdash_j T$, we have its denotation $[\![T]\!]_C \in \mathtt{Ty}^\blacktriangleright_j\ [\![\Gamma]\!]^\bullet_C\ T$ and $[\![T]\!]^\bullet_C \in \mathtt{Ty}^\bullet{}_j\ [\![\Gamma]\!]^\bullet_C$; for each well-typed term $\Gamma \vdash t\ :\ T$, we have its denotation $[\![t]\!]_C \in \mathtt{Tm}^\blacktriangleright\ [\![\Gamma]\!]^\bullet_C\ [\![T]\!]^\bullet_C\ t$ and $[\![t]\!]^\bullet_C \in \mathtt{Tm}^\bullet\ [\![\Gamma]\!]^\bullet_C\ [\![T]\!]^\bullet_C$; etc. The audience should notice that these structure are just the type of the interpretation of judgments in the MLTT canonicity model.

Notice that the input type $\prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma'\in\Gamma^\blacktriangleright(\gamma)}$ is part of $\mathtt{Ty}^\bullet$, $\mathtt{Tm}^\bullet$, and $\mathtt{Sub}^\bullet$. A useful fact about them is : given a pair of arbitrary $\cdot \vdash \gamma\ :\ \Gamma$ and $\gamma' \in \Gamma^\blacktriangleright(\gamma)$, we can consider (γ,γ') as an element of $\mathtt{Sub}^\bullet\ \cdot^\bullet\ \Gamma^\bullet$, and vice versa. Thus, we consider the pair (γ,γ') the equivalent form of an element $\gamma^\bullet \in \mathtt{Sub}^\bullet\ \cdot^\bullet\ \Gamma^\bullet$.

So sometimes instead of working on $\prod_{\cdot\vdash\gamma:\Gamma} \prod_{\gamma'\in\Gamma^\blacktriangleright(\gamma)}$ ..., we will work on $\prod_{\gamma^\bullet\in\mathtt{Sub}^\bullet\ \cdot^\bullet\ \Gamma^\bullet}$ ...

A.3.5 Canonicity Model for FMLTT$^{2.0}$. .

Interpreting Contextual Nat.

We want to show $\cdot \vdash n\ \mathtt{Nat}$ are bounds to be natural number, and thus we inductively define a predicate $\mathtt{Nat}_\varnothing{}^c\ \cdot\ :\ \left\{n \,\middle|\, \vdash n\ \mathtt{Nat}\right\} \to \mathtt{Prop}$

$$\frac{}{\mathtt{0}_\varnothing{}^c\ :\ \mathtt{Nat}_\varnothing{}^c\ 0} \qquad \frac{x\ :\ \mathtt{Nat}_\varnothing{}^c\ n}{\mathtt{S}_\varnothing{}^c\ x\ :\ \mathtt{Nat}_\varnothing{}^c\ n+1}$$

Then we can define the interpretation for judgments and derivations: ▸

$$[\![\Gamma \vdash x\ \mathtt{Nat}]\!]_C \text{ is a depedent function } : \prod_{\gamma^\bullet\in\mathtt{Sub}^\bullet\ \cdot^\bullet\ \Gamma^\bullet} \mathtt{Nat}_\varnothing{}^c\ x[\gamma]$$

$$[\![\Gamma \vdash x[\sigma]\ \ \mathtt{Nat}]\!]_C(\gamma^\bullet) = [\![x]\!]_C(\sigma^\bullet\circ^\bullet\gamma^\bullet)$$

$$[\![\Gamma \vdash 0\ \ \mathtt{Nat}]\!]_C(\gamma^\bullet) = \mathtt{0}_\varnothing{}^c$$

$$[\![\Gamma \vdash n+1\ \ \mathtt{Nat}]\!]_C(\gamma^\bullet) = \mathtt{S}_\varnothing{}^c\ [\![n]\!]_C(\gamma^\bullet)$$

[6] This is also called glued interpretation in Sterling [56].

We have verified that this interpretation respects the equality/quotient in the syntax.
We can subsequently define

$$\mathtt{Nat}^{\blacktriangleright}\ \Gamma^{\bullet}\ n \coloneqq \prod_{\gamma^{\bullet}\in \mathtt{Sub}^{\bullet}\ \therefore^{\bullet}\ \Gamma^{\bullet}} \mathtt{Nat}_{\varnothing}{}^{c}\ x[\gamma] \qquad\qquad \mathtt{Nat}^{\bullet}\ \Gamma^{\bullet} \coloneqq \sum_{\Gamma\vdash n\ \ \mathtt{Nat}} \mathtt{Nat}^{\blacktriangleright}\ \Gamma^{\bullet}\ n$$

, where for arbitrary $\Gamma \vdash n\ \ \mathtt{Nat}$, we have $[\![n]\!]_C \in \mathtt{Nat}^{\blacktriangleright}\ [\![\Gamma]\!]_C^{\bullet}\ n$ and $[\![n]\!]_C^{\bullet} \in \mathtt{Nat}^{\bullet}\ [\![\Gamma]\!]_C^{\bullet}$.

Interpreting Linkage Signature, Linkage and Linkage Type.

Similarly, to characterize closed signature, linkage and linkage type, we need to inductively classify them.

We mutually-recursively define the following indexed sets, by induction on parameter n, since we have $n^{\bullet} \in \mathtt{Nat}^{\bullet}\ \therefore^{\bullet}$ witness n to be a plain inductive natural number

$$\begin{aligned}
{}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ (\cdot)\ (\cdot) &: \prod_{n^{\bullet}\in\mathtt{Nat}^{\bullet}\ \therefore^{\bullet}} \left\{\sigma \;\middle|\; \vdash \sigma\ \ \mathtt{LSig}^{n}\right\} \to \mathtt{Set}_{i+1}\\
\mathbb{L}_{\varnothing}{}^{c}((\cdot)) &: \prod_{\sigma^{\blacktriangleright}\in\sum_{\sigma}{}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ n^{\bullet}\ \sigma} \to \mathtt{Ty}_{i}^{\blacktriangleright}\ \therefore^{\bullet}\ \mathbb{L}(\sigma)\\
{}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ 0\ \sigma &= \{\star\}\\
{}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ (n+1)\ \sigma &= {}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ n\ \nu\pi_1(\sigma)\\
&\quad \times \sum_{A^{\blacktriangleright}\in\mathtt{Ty}^{\blacktriangleright}\ \therefore^{\bullet}\ \nu\pi_1'(\sigma)} \mathtt{Ty}_{i}^{\blacktriangleright}\ (\therefore^{\bullet},^{\bullet}(\nu\pi_1'(\sigma), A^{\blacktriangleright}))\ \nu\pi_2(\sigma)\\
\mathbb{L}_{\varnothing}{}^{c}((\cdots : {}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ 0\ \cdots))(\gamma^{\bullet})(o) &= \left\{\star \;\middle|\; o \equiv \mu^{\bullet}\right\}\\
\mathbb{L}_{\varnothing}{}^{c}(((X,(\sigma^{\blacktriangleright},A^{\blacktriangleright},T^{\blacktriangleright})) : {}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ n+1\ \cdots))(\gamma^{\bullet})(o) &= \mathbb{L}_{\varnothing}{}^{c}((\nu\pi_1(X),\sigma^{\blacktriangleright}))(\gamma^{\bullet})(\mu\pi_1(o))\\
&\quad \times \mathtt{Tm}^{\blacktriangleright}\ (\therefore^{\bullet},^{\bullet}(\nu\pi_1'(\sigma),A^{\blacktriangleright}))\ (\nu\pi_2(\sigma),T^{\blacktriangleright})\ \mu\pi_2(o)
\end{aligned}$$

Now we can interpret derivation and judgment.

$$\begin{aligned}
[\![\Gamma \vdash_i \sigma\ \ \mathtt{LSig}^{n}]\!]_C \text{ is a dependent function } &: \prod_{\gamma^{\bullet}\in\mathtt{Sub}^{\bullet}\ \therefore^{\bullet}\ \Gamma^{\bullet}} {}^{i}\mathtt{LSig}_{\varnothing}{}^{c}\ [\![n]\!]_C^{\bullet}[\gamma^{\bullet}]^{\bullet}\ \sigma[\gamma]\\
[\![\Gamma \vdash s[\sigma]\ \ \mathtt{LSig}^{n}]\!]_C(\gamma^{\bullet}) &= [\![s]\!]_C([\![\sigma]\!]_C^{\bullet}\circ^{\bullet}\gamma^{\bullet})\\
[\![\Gamma \vdash \mathbb{L}(\sigma)]\!]_C(\gamma^{\bullet}) &= \mathbb{L}_{\varnothing}{}^{c}((\sigma[\gamma]\,,[\![\sigma]\!]_C(\gamma^{\bullet})))\ _\\
[\![\Gamma \vdash \nu^{+}(\sigma;\ A \vdash T)\ \ \mathtt{LSig}^{n+1}]\!]_C(\gamma^{\bullet}) &= ([\![\sigma]\!]_C(\gamma^{\bullet}),[\![A]\!]_C^{\bullet}[\gamma^{\bullet}]^{\blacktriangleright},[\![T]\!]_C^{\bullet}[\gamma^{\bullet\uparrow}]^{\blacktriangleright})\\
[\![\Gamma \vdash \nu\pi_1(\sigma)\ \ \mathtt{LSig}^{n}]\!]_C(\gamma^{\bullet}) &= [\![\sigma]\!]_C(\gamma^{\bullet}).{}_0\\
[\![\Gamma \vdash \nu\pi_1'(\sigma)]\!]_C(\gamma^{\bullet}) &= [\![\sigma]\!]_C(\gamma^{\bullet}).{}_1()\\
[\![\Gamma,\nu\pi_1'(\sigma) \vdash \nu\pi_2(\sigma)]\!]_C(\gamma^{\bullet}) &= [\![\sigma]\!]_C(\pi_1{}^{\bullet}\gamma^{\bullet}).{}_2(\mathtt{id}^{\bullet},^{\bullet}\pi_2{}^{\bullet}\gamma^{\bullet})\\
[\![\Gamma \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})]\!]_C(\gamma)(\gamma') &= \star\\
[\![\Gamma \vdash \mu^{+}(\ell,t) : \mathbb{L}(\nu^{+}(\sigma;\ A \vdash T))]\!]_C(\gamma)(\gamma') &= ([\![\ell]\!]_C(\gamma)(\gamma'),[\![t]\!]_C^{\bullet}[\gamma^{\bullet\uparrow}]^{\blacktriangleright})\\
[\![\Gamma \vdash \mu\pi_1(\ell) : \mathbb{L}(\nu\pi_1(\sigma))]\!]_C(\gamma^{\bullet}) &= [\![\ell]\!]_C(\gamma^{\bullet}).{}_0
\end{aligned}$$

$$[\![\Gamma, v\pi_1'(\sigma) \vdash \mu\pi_2(\ell) : \mathbb{L}(v\pi_1(\sigma))]\!]_C(\gamma^\bullet) = ([\![\ell]\!]_C(\pi_1{}^\bullet\gamma^\bullet))._1(\mathtt{id}^\bullet, \pi_2{}^\bullet\gamma^\bullet)$$

We have verified that this interpretation respects the equality/quotient in the syntax.
We can subsequently define

$${}^i\mathtt{LSig}^\blacktriangleright\ \Gamma^\bullet\ n\ \sigma \coloneqq \prod_{\gamma^\bullet \in \mathtt{Sub}^\bullet\ \cdot^\bullet\ \Gamma^\bullet} {}^i\mathtt{LSig}_\emptyset{}^c\ [\![n]\!]_C^\bullet[\gamma^\bullet]^\bullet\ \sigma[\gamma] \qquad {}^i\mathtt{LSig}^\bullet\ \Gamma^\bullet\ n \coloneqq \sum_{\Gamma \vdash_i \sigma\ \mathtt{LSig}^n} {}^i\mathtt{LSig}^\blacktriangleright\ \gamma^\bullet\ n\ \sigma$$

, where for arbitrary $\Gamma \vdash \sigma\ \mathtt{LSig}^n$, we have $[\![\sigma]\!]_C \in {}^i\mathtt{LSig}^\blacktriangleright\ \Gamma^\bullet\ n\ \sigma$ and $[\![\sigma]\!]_C^\bullet \in {}^i\mathtt{LSig}^\bullet\ \Gamma^\bullet\ n$.

Interpreting Linkage Transformer.

For the linkage transformer, we will take a different strategy. We first inductively define two indexed sets $\mathtt{Inh}^\bullet$ and $\mathtt{ProjWk}^\bullet$:

$$\frac{}{{}^I\mathtt{Id}^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma^\bullet} \qquad \frac{h^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \quad t^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet A^\bullet)\ T^\bullet}{{}^I\mathtt{Ext}^\bullet(h^\bullet, t^\bullet) : \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ v^{+\bullet}(\sigma'^\bullet, A^\bullet, T^\bullet)}$$

$$\frac{h^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \quad T^\bullet \in \mathtt{Ty}^\bullet\ (\Gamma^\bullet,^\bullet A^\bullet)}{{}^I\mathtt{Inherit}^\bullet(h^\bullet) \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ v^{+\bullet}(\sigma^\bullet, A^\bullet, T^\bullet)\ v^{+\bullet}(\sigma'^\bullet, A^\bullet, T^\bullet)}$$

$$\frac{h^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \quad T^\bullet \in \mathtt{Ty}^\bullet\ (\Gamma^\bullet,^\bullet A^\bullet) \quad t^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet A^\bullet,^\bullet T^\bullet)\ T^\bullet[(\mathtt{p}^1)^\bullet]^\bullet}{{}^I\mathtt{Ov}^{1\bullet}(h^\bullet, t^\bullet) \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ v^{+\bullet}(\sigma^\bullet, A^\bullet, T^\bullet)\ v^{+\bullet}(\sigma'^\bullet, A^\bullet, T^\bullet)}$$

$$\frac{h^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \quad t_1{}^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet A_1{}^\bullet)\ T_1{}^\bullet \quad t_2{}^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet A_2{}^\bullet)\ T_2{}^\bullet}{{}^I\mathtt{Ov}^{2\bullet}(h^\bullet, t_1{}^\bullet, t_2{}^\bullet) \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ v^{+\bullet}(\sigma^\bullet, A^\bullet, \mathbb{S}^\bullet(t_1{}^\bullet))\ v^{+\bullet}(\sigma'^\bullet, A^\bullet, \mathbb{S}^\bullet(t_1{}^\bullet))}$$

$$\begin{gathered}\mathtt{ProjWk}^\bullet\ A_2{}^\bullet\ A_1{}^\bullet \coloneqq \\ \{f^\bullet \mid \\ \{(M^\bullet, f_1{}^\bullet, f_2{}^\bullet) \mid M^\bullet \in \mathtt{Ty}^\bullet\ (\Gamma^\bullet,^\bullet A_1{}^\bullet), \quad f_1{}^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet \Sigma(A_1{}^\bullet, M^\bullet))\ A_2{}^\bullet[\mathtt{p}^1], \\ f_2{}^\bullet \in \mathtt{Tm}^\bullet\ (\Gamma^\bullet,^\bullet A_2{}^\bullet)\ \Sigma(A_1{}^\bullet, M^\bullet)[(\mathtt{p}^1)^\bullet]^\bullet, \quad f_2{}^\bullet[(\mathtt{p}^1)^\bullet,^\bullet f_1{}^\bullet]^\bullet \equiv \mathtt{var}_1{}^\bullet \quad f_1{}^\bullet[(\mathtt{p}^1)^\bullet, f_2{}^\bullet]^\bullet \equiv \mathtt{var}_1{}^\bullet \\ f^\bullet \equiv \mathtt{fst}\ \mathtt{var}_1[(\mathtt{p}^1)^\bullet,^\bullet f_2{}^\bullet]^\bullet\} \\ \}\end{gathered}$$

$$\frac{h^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \quad \restriction^\bullet \in \mathtt{ProjWk}^\bullet\ A_2{}^\bullet\ A_1{}^\bullet \quad h_0{}^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet,^\bullet A_2{}^\bullet\ \tau_1{}^\bullet[(\mathtt{p}^1)^\bullet,^\bullet \restriction^\bullet]^\bullet\ \tau_2{}^\bullet}{{}^I\mathtt{Nest}^\bullet(h^\bullet, \restriction^\bullet, h_0{}^\bullet) \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ v^{+\bullet}(\sigma^\bullet, A_1{}^\bullet, \mathbb{L}^\bullet(\tau_1{}^\bullet))\ v^{+\bullet}(\sigma'^\bullet, A_2{}^\bullet, \mathbb{L}^\bullet(\tau_2{}^\bullet))}$$

We can easily spot the similarity between these inductive rules and the syntax rules surrounding $\Gamma \vdash (\cdot) : \sigma \twoheadrightarrow \sigma'$. In fact, we can consider this inductive type, as solely just the six constructors ${}^I\mathtt{Id}$, ${}^I\mathtt{Inherit}()$, ${}^I\mathtt{Ext}(,)$, ${}^I\mathtt{Ov}^1(,)$, ${}^I\mathtt{Ov}^2(,,)$, ${}^I\mathtt{Nest}(,,)$, but working on $S^\bullet$ instead of S.

The similarity induces the following function, defined by induction/pattern matching on $\mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet$:

$$\begin{aligned}(\cdot)\!\downarrow &: \mathtt{ProjWk}^\bullet\ A_1{}^\bullet\ A_1{}^\bullet \to \left\{ f \,\middle|\, \mathtt{ProjWk}\ A_2\ A_1\ f \right\} \\ (\cdot)\!\downarrow &: \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet \to \left\{ i \,\middle|\, \Gamma \vdash i : \sigma \twoheadrightarrow \sigma' \right\}\end{aligned}$$

$(\cdot)\!\downarrow$ formally connects $i^\bullet \in \mathtt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet$ with its "syntax mirror" $(i^\bullet)\!\downarrow \equiv \Gamma \vdash i : \sigma \twoheadrightarrow \sigma'$. Just as the notation suggests, we somehow have $S^\bullet$(the syntax and semantic compoenent) and S (syntax component). (Yet, we don't have the definition for $S^\blacktriangleright$ right now.)

Since $\texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet$ is ultimately an inductive type, we can define the following 5 dependent functions

$$\begin{aligned}
_[\cdot]^\bullet &: (f^\bullet \in \texttt{ProjWk}^\bullet\ A_2^\bullet\ A_1^\bullet) \to (\delta^\bullet \in \texttt{Sub}^\bullet\ \Delta^\bullet\ \Gamma^\bullet) \\
&\to \left\{ f'^\bullet \in \texttt{ProjWk}^\bullet\ (A_2^\bullet[\delta^\bullet]^\bullet)\ (A_1^\bullet[\delta^\bullet]^\bullet) \;\middle|\; (f'^\bullet)\!\downarrow \equiv (f^\bullet)\!\downarrow [\delta] \right\} \\
_[\cdot]^\bullet &: (h^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_2^\bullet) \to (\delta^\bullet \in \texttt{Sub}^\bullet\ \Delta^\bullet\ \Gamma^\bullet) \\
&\to \left\{ h'^\bullet \in \texttt{Inh}^\bullet\ \Delta^\bullet\ (\sigma_1^\bullet[\delta^\bullet]^\bullet)\ (\sigma_2^\bullet[\delta^\bullet]^\bullet) \;\middle|\; (h'^\bullet)\!\downarrow \equiv (h'^\bullet)\!\downarrow [\delta] \right\} \\
\texttt{inh}^\bullet((\cdot),(\cdot)) &: (i^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet) \to (h^\bullet \in \texttt{Tm}^\bullet\ \cdot^\bullet\ \mathbb{L}^\bullet(\sigma^\bullet)) \\
&\to \left\{ t^\bullet \in \texttt{Tm}^\bullet\ \cdot^\bullet\ \mathbb{L}^\bullet(\sigma'^\bullet) \;\middle|\; t \equiv \texttt{inh}((i^\bullet)\!\downarrow, h) \right\} \\
(\cdot)\circ^\bullet(\cdot) &: (i_1^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma'^\bullet) \to (i_2^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma'^\bullet\ \sigma''^\bullet) \\
&\to \left\{ i_3^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma^\bullet\ \sigma''^\bullet \;\middle|\; (i_3^\bullet)\!\downarrow \equiv (i_1^\bullet)\!\downarrow\ \circ (i_2^\bullet)\!\downarrow \right\} \\
(\cdot)\oplus^\bullet(\cdot) &: (i^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_2^\bullet) \to (i'^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_3^\bullet) \\
&\to \left\{ (\sigma_4^\bullet, i_3^\bullet, i_4^\bullet) \;\middle|\; (\sigma_4, (i_3^\bullet)\!\downarrow, (i_4^\bullet)\!\downarrow) \equiv (i^\bullet)\!\downarrow \oplus (i'^\bullet)\!\downarrow \right\}
\end{aligned}$$

, by induction/pattern matching on $\texttt{Inh}^\bullet\ (\cdot)\ (\cdot)\ (\cdot)$. Most importantly, they respect the "syntax mirror" $(\cdot)\!\downarrow$. The definition of these 5 functions directly come from their beta equations indicated in the syntax. In fact, those beta equations cover all possible cases, and give a complete definition (as we are using intrinsic typing).

Finally, we can define $S^\blacktriangleright$:

$$\begin{aligned}
\texttt{ProjWk}^\blacktriangleright\ A_2^\bullet\ A_1^\bullet\ f &:= \left\{ f^\bullet \in \texttt{ProjWk}^\bullet\ A_2^\bullet\ A_1^\bullet \;\middle|\; f \equiv (f^\bullet)\!\downarrow \right\} \\
\texttt{Inh}^\blacktriangleright\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_2^\bullet\ h &:= \left\{ i_0^\bullet \in \texttt{Inh}^\bullet\ \Gamma^\bullet\ (\sigma_1^\bullet)\ (\sigma_2^\bullet) \;\middle|\; h \equiv (i_0^\bullet)\!\downarrow \right\}
\end{aligned}$$

In the previous model constructions for linkage and signatures, we always define the semantic component $S^\blacktriangleright$ and then define their combination $S^\bullet$. However, when dealing with linkage transformer, we take another route, by first defining $S^\bullet$ and its syntax componenet $(S^\bullet)\!\downarrow$, then we use set comprehension (subset) to construct $S^\blacktriangleright$. It is easy to prove $\texttt{Inh}^\bullet\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_2^\bullet \cong \sum_h \texttt{Inh}^\blacktriangleright\ \Gamma^\bullet\ \sigma_1^\bullet\ \sigma_2^\bullet\ h$.

Now we interpret judgments and derivations around linkage transformer.

$$\begin{aligned}
[\![\texttt{ProjWk}\ A_2\ A_1\ f]\!]_C &: \texttt{ProjWk}^\blacktriangleright\ [\![A_2]\!]_C^\bullet\ [\![A_1]\!]_C^\bullet\ f \\
[\![\Gamma \vdash h : \sigma_1 \twoheadrightarrow \sigma_2]\!]_C &: \texttt{Inh}^\blacktriangleright\ [\![\Gamma]\!]_C^\bullet\ [\![\sigma_1]\!]_C^\bullet\ [\![\sigma_2]\!]_C^\bullet\ h \\
[\![\Gamma \vdash {}^I\texttt{Id} : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Id}^\bullet \\
[\![\Gamma \vdash {}^I\texttt{Ext}(h,t) : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Ext}^\bullet([\![h]\!]_C, [\![t]\!]_C^\bullet) \\
[\![\Gamma \vdash {}^I\texttt{Inherit}(h) : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Inherit}^\bullet([\![h]\!]_C) \\
[\![\Gamma \vdash {}^I\texttt{Ov}^1(h,t) : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Ov}^{1\bullet}([\![h]\!]_C, [\![t]\!]_C^\bullet) \\
[\![\Gamma \vdash {}^I\texttt{Ov}^2(h,t_1,t_2) : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Ov}^{2\bullet}([\![h]\!]_C, [\![t_1]\!]_C^\bullet, [\![t_2]\!]_C^\bullet) \\
[\![\Gamma \vdash {}^I\texttt{Nest}(h,\restriction,h_0) : \cdots \twoheadrightarrow \cdots]\!]_C &= {}^I\texttt{Nest}^\bullet([\![h]\!]_C, [\![\restriction]\!]_C^\bullet, [\![h_0]\!]_C) \\
[\![\Gamma \vdash \texttt{inh}(h,a) : \cdots]\!]_C &= \texttt{inh}^\bullet([\![h]\!]_C, [\![a]\!]_C^\bullet) \\
[\![\Gamma \vdash h_1 \circ h_2 : \cdots \twoheadrightarrow \cdots]\!]_C &= [\![h_1]\!]_C \circ^\bullet [\![h_2]\!]_C \\
[\![h_1 \oplus h_2]\!]_C &= [\![h_1]\!]_C \oplus^\bullet [\![h_2]\!]_C
\end{aligned}$$

$$[\![\Delta \vdash h[\delta] : \cdots \twoheadrightarrow \cdots]\!]_C = [\![h]\!]_C\Big[[\![\delta]\!]^\bullet_C\Big]^\bullet$$

We have verified that this interpretation respects the equality/quotient in the syntax, surrounding those linkage transformer.

The interpretation is trivially mapping the quotiented syntax to the highly-similar non-quotient inductive type we just defined. The triviality of this interpretation should reminisce the fact that, linkage transformer itself has limited interaction with the core calculus and can be considered as an external library out of the core calculus as in Jin et al. [27]. In FMLTT$^{2.0}$, we move it into the core calculus simply because the existential linkage type $\mathbb{L}^+(\sigma)$ requires linkage transformer, while $\mathbb{L}^+(\sigma)$ cannot be encoded outside of core calculus.

Interpreting Existential Linkage Type.

$$\begin{aligned}
[\![\Gamma \vdash_{i+1} \mathbb{L}^+(\sigma)]\!]_C(\gamma)(\gamma')(o) &= \left\{ N^\bullet, \sigma'^\bullet, h^\bullet, o_i^\bullet \;\middle|\; o \equiv \mathtt{iL}^+(N, \sigma', h, o_i) \right\} \\
&\quad \text{where } N^\bullet \in \mathtt{Nat}^\bullet \cdot^\cdot, \sigma'^\bullet \in {}^i\mathtt{LSig}^\bullet \cdot^\cdot N^\bullet, h^\bullet \in \mathtt{Inh}^\bullet \cdot^\cdot ([\![\sigma]\!]^\bullet_C[\gamma^\bullet]^\bullet)\, \sigma'^\bullet, \\
&\quad o_i^\bullet \in \mathtt{Tm}^\bullet \cdot^\cdot \mathbb{L}^\bullet(\sigma'^\bullet) \\
[\![\Gamma \vdash \mathtt{iL}^+(N, \sigma', h, o_i) : \mathbb{L}^+(\sigma)]\!]_C(\gamma)(\gamma') &= ([\![N]\!]^\bullet_C[\gamma^\bullet]^\bullet, [\![\sigma']\!]^\bullet_C[\gamma^\bullet]^\bullet, [\![h]\!]^\bullet_C[\gamma^\bullet]^\bullet, [\![o_i]\!]^\bullet_C[\gamma^\bullet]^\bullet) \\
[\![\Gamma \vdash o.\mathtt{size\ Nat}]\!]_C(\gamma^\bullet) &= (([\![o]\!]_C(\gamma^\bullet)).{}_0)^\blacktriangleright() \\
[\![\Gamma \vdash o.\mathtt{sig\ LSig}^{o.\mathtt{size}}]\!]_C(\gamma^\bullet) &= (([\![o]\!]_C(\gamma^\bullet)).{}_1)^\blacktriangleright() \\
[\![\Gamma \vdash o.\mathtt{inh} : \sigma \twoheadrightarrow o.\mathtt{sig}]\!]_C(\gamma^\bullet) &= (([\![o]\!]_C(\gamma^\bullet)).{}_2)^\blacktriangleright() \\
[\![\Gamma \vdash \mathtt{eL}^+(o) : \mathbb{L}(\sigma)]\!]_C(\gamma^\bullet) &= (([\![o]\!]_C(\gamma^\bullet)).{}_3)^\blacktriangleright()
\end{aligned}$$

We have verified that this interpretation respects the equality/quotient in the syntax, surrounding those existential linkage type $\mathbb{L}^+(\sigma)$.

Interpreting W-type. For W-type, we take a similar approach as dealing with linkage transformer. We first define

$$\begin{aligned}
{}^j\mathtt{WSig}^\bullet\, \Gamma^\bullet\, n &\coloneqq \mathtt{Vector}^n \sum_{A^\bullet \in \mathtt{Ty}^\bullet_j\, \Gamma^\bullet} \mathtt{Ty}^\bullet_j\, (\Gamma^\bullet, {}^\cdot A^\bullet) \\
(\cdot)\!\downarrow\; &: {}^j\mathtt{WSig}^\bullet\, \Gamma^\bullet\, n \to \left\{ \tau \;\middle|\; \Gamma \vdash_j \tau\ \mathtt{WSig}^n \right\} \\
(\{(A_i^\bullet, T_i^\bullet)\})\!\downarrow\; &\coloneqq \mathtt{w}^+(\mathtt{w}^+(\cdots \mathtt{w}^+(\mathtt{w}^\bullet, A_1, T_1) \cdots, A_{n-1}, T_{n-1}), A_n, T_n) \\
{}^j\mathtt{WSig}^\blacktriangleright\, \Gamma^\bullet\, n\, \tau &\coloneqq \left\{ w^\bullet \;\middle|\; (w^\bullet)\!\downarrow \equiv \tau \right\}
\end{aligned}$$

, where $\mathtt{Vector}^n$ is a list with length n.

Similarly, we first define $S^\bullet$ and its syntax componenet $(S^\bullet)\!\downarrow$, then we use set comprehension (subset type) to construct $S^\blacktriangleright$. It is easy to prove ${}^j\mathtt{WSig}^\bullet\, \Gamma^\bullet\, n \cong \sum_\tau {}^j\mathtt{WSig}^\blacktriangleright\, \Gamma^\bullet\, n\, \tau$.

Now we interpret the judgments and derivations.

$$\begin{aligned}
[\![\Gamma \vdash_j \tau\ \mathtt{WSig}^n]\!]_C &: {}^j\mathtt{WSig}^\blacktriangleright\, [\![\Gamma]\!]^\bullet_C\, n\, \tau \\
&\quad \text{i.e., a list of 2-tuple of length } n, \text{ with syntax on} \\
[\![\Gamma \vdash \mathtt{w}^\bullet\ \mathtt{WSig}^0]\!]_C &= \mathtt{nil}
\end{aligned}$$

$$\llbracket \Gamma \vdash \mathtt{w}^{+}(\tau, A, B)\ \ \mathtt{WSig}^{n+1} \rrbracket_C = (\llbracket A \rrbracket_C^{\bullet}, \llbracket B \rrbracket_C^{\bullet}) :: \llbracket \tau \rrbracket_C$$

$$\llbracket \Gamma \vdash \tau[\gamma]\ \ \mathtt{WSig}^{n} \rrbracket_C \text{ is done by point-wise/component-wise substitution}$$

$$\mathtt{w}\pi_1^{j\bullet}(\tau^{\bullet}) = (\ j\text{-th element of }\ \tau^{\bullet})[0]$$

$$\llbracket \Gamma \vdash \mathtt{w}\pi_1^{j}(\tau) \rrbracket_C = \mathtt{w}\pi_1^{j\bullet}(\llbracket \tau \rrbracket_C^{\bullet})$$

$$\mathtt{w}\pi_2^{j\bullet}(\tau^{\bullet}) = (\ j\text{-th element of }\ \llbracket \tau \rrbracket_C^{\bullet})[1]$$

$$\llbracket \Gamma, \mathtt{w}\pi_1^{j}(\tau) \vdash \mathtt{w}\pi_2^{j}(\tau) \rrbracket_C = \mathtt{w}\pi_2^{j\bullet}(\tau^{\bullet})$$

$$\llbracket \Gamma \vdash \mathtt{w}^{-}(\tau)\ \ \mathtt{WSig}^{n} \rrbracket_C = \mathtt{tl}\ \llbracket \tau \rrbracket_C$$

Next we can interpret the related terms and types for W-type.

$$\llbracket \Gamma \vdash \mathtt{W}(\tau)\ :\ \mathbb{U} \rrbracket_C(\gamma)(\gamma')(t) = W^C\ (\llbracket \tau \rrbracket_C^{\bullet}[\gamma^{\bullet}])\ t$$

$$\llbracket \Gamma \vdash \mathtt{Wsup}_i(\tau, a, b)\ :\ \mathtt{El}(\mathtt{W}(\tau)) \rrbracket_C(\gamma)(\gamma') = W^C\ sup\ i\ \ (\llbracket a \rrbracket_C^{\bullet}[\gamma^{\bullet}])\ \ (\llbracket b \rrbracket_C^{\bullet}[\gamma^{\uparrow\bullet}])$$

$$\mathtt{CaseTy}^{\bullet}(A^{\bullet}, B^{\bullet}, R^{\bullet}) = \Pi^{\bullet}(A^{\bullet}, \Pi^{\bullet}(\Pi^{\bullet}(B^{\bullet}, R^{\bullet}[(\mathtt{p}^2)^{\bullet}]), R^{\bullet}[(\mathtt{p}^2)^{\bullet}]))$$

$$\llbracket \Gamma \vdash \mathtt{CaseTy}(\mathtt{A}, \mathtt{B}, \mathtt{R}) \rrbracket_C = (\mathtt{CaseTy}^{\bullet}(\llbracket \mathtt{A} \rrbracket_C^{\bullet}, \llbracket \mathtt{B} \rrbracket_C^{\bullet}, \llbracket \mathtt{R} \rrbracket_C^{\bullet}))^{\blacktriangleright}$$

we define $\mathtt{RecSig}(,)$ by induction on the signature, via $RS^{\bullet}$

$$RS^{\bullet}\ \mathtt{nil}\ R = \llbracket v^{\bullet} \rrbracket_C^{\bullet}$$

$$RS^{\bullet}\ ((A, B) :: tl)\ R = v^{+\bullet}(RS^{\bullet}\ tl\ R, \pi_2{}^{\bullet}, \mathtt{CaseTy}^{\bullet}(A, B, R))$$

$$\llbracket \Gamma \vdash \mathtt{RecSig}(\tau, \mathtt{R}) \rrbracket_C = RS^{\blacktriangleright}\ \llbracket \tau \rrbracket_C\ \llbracket \mathtt{R} \rrbracket_C$$

$$\mathtt{R}\pi^{j\blacktriangleright}(\ell^{\bullet}) = \text{ take the } j\text{-th field from } \ell^{\bullet}$$

$$\llbracket \mathtt{R}\pi^{j}(\ell) \rrbracket_C = \mathtt{R}\pi^{j\bullet}(\llbracket \ell \rrbracket_C^{\bullet})$$

$$\llbracket \Gamma \vdash \mathtt{Wrec}(\tau, \ell, t)\ :\ T \rrbracket_C(\gamma)(\gamma') = W^C\ rec\ \ \llbracket \tau \rrbracket_C^{\bullet}[\gamma^{\bullet}]\ \ (\lambda w.\llbracket \mathtt{R} \rrbracket_C(\gamma)(\gamma')(\mathtt{Wrec}(\tau[\gamma], \ell[\gamma], w)))\ \ f^r\ \ t[\gamma]\ \ (\llbracket t \rrbracket_C(\gamma)(\gamma'))$$

where $f^r\ j\ a^{\bullet}\ b\ b^c = \mathtt{let}\ \rho^{\bullet} \in \mathtt{Tm}^{\bullet}\ \ (\cdot^{\bullet}, {}^{\bullet}\ B^{\bullet}[\mathtt{id}^{\bullet}, {}^{\bullet} a^{\bullet}]^{\bullet})\ \ (R^{\bullet}[\gamma^{\bullet}]^{\bullet}[(\mathtt{p}^9)^{\bullet}]^{\bullet})$

$$\mathtt{s.t.}\ \rho^{\bullet} := (\mathtt{Wrec}(\tau, \ell[\gamma \circ \mathtt{p}^1], b), b^c)\ \ \mathtt{in}$$

$$\mathtt{app}^{\bullet}(\ \mathtt{app}^{\bullet}(\ \mathtt{R}\pi^{j\bullet}(\llbracket \ell \rrbracket_C^{\bullet}[\gamma^{\bullet}])\ )\ [\mathtt{id}^{\bullet}, {}^{\bullet} a^{\bullet}]^{\bullet}\ )\ [\mathtt{id}^{\bullet}, {}^{\bullet}\ \lambda^{\bullet}(\rho^{\bullet})\ ]^{\bullet}\ (\epsilon)(\star)$$

given $\Gamma \vdash \ell\ :\ \mathtt{RecSig}(\tau, \mathtt{R})$

where $R^{\bullet} = \llbracket \mathtt{R} \rrbracket_C^{\bullet}, B^{\bullet} = \mathtt{w}\pi_2^{j\bullet}(\llbracket \tau \rrbracket_C^{\bullet})$

Notice these constructions rely on the inductive defined set W^C, its constructor $W^C sup$ and its eliminator $W^C rec$, which we will show in the following. These three concepts are defined using the inductive facility in the ambient logic. For example, the interpretation of $\mathtt{Wsup}_i(\tau, a, b)$ is defined using $W^C sup$, taking three arguments as input: i, a glued $(\llbracket a \rrbracket^\bullet_C [\gamma^\bullet]^\bullet)$ and a glued $(\llbracket b \rrbracket^\bullet_C [\gamma^{\uparrow\bullet}]^\bullet)$. As mentioned earlier, $\gamma^\bullet$ is an equivalent form of (γ, γ').

Finally, we show how W^C, $W^C sup$ and $W^C rec$ are defined.

(Note: we also use $(a \in A \to B(a))$ as another notation for dependent function $\prod_{a \in A} B(a)$).

$$\begin{aligned}
&\mathtt{Inductive}\ W^C : (\tau^\bullet \in {}^i\mathtt{WSig}^\bullet\ \llbracket\cdot\rrbracket^\bullet_C\ N) \to \left\{ t \,\middle|\, \cdot \vdash t : \mathtt{El}(\mathtt{W}(\tau)) \right\} \to \mathtt{Set}_{i+1} \quad \mathtt{where} \\
&\quad W^C sup : j < N \to a^\bullet \in \mathtt{Tm}^\bullet \cdot^\bullet \mathtt{w}\pi_1^{j\bullet}(\tau^\bullet) \\
&\qquad \to b^\bullet \in (\mathtt{Tm}^\bullet\ (\cdot^\bullet, {}^\bullet\ \mathtt{w}\pi_2^{j\bullet}(\tau^\bullet)[\mathtt{id}^\bullet, a^\bullet]^\bullet\ )\ \mathtt{El}^\bullet(\mathtt{W}^\bullet(\tau^\bullet))[(\mathtt{p}^1)^\bullet]^\bullet\ ) \\
&\qquad \to W^C\ \tau^\bullet\ \mathtt{Wsup}_j(\tau, a, b)
\end{aligned}$$

and its eliminator $W^C rec$

$$\begin{aligned}
W^C rec\ &: (\tau^\bullet \in {}^i\mathtt{WSig}^\bullet\ \llbracket\cdot\rrbracket^\bullet_C\ N) \to (P : \left\{ t \,\middle|\, \cdot \vdash t : \mathtt{El}(\mathtt{W}(\tau)) \right\} \to \mathtt{Set}_k) \\
&\to \Bigg(j < N \to a^\bullet \in \mathtt{Tm}^\bullet \cdot^\bullet \mathtt{w}\pi_1^{j\bullet}(\tau^\bullet) \\
&\quad \to \left\{ b \,\middle|\, (\cdot,\ \mathtt{w}\pi_2^j(\tau)[\mathtt{id}, a]\ ) \vdash b : \mathtt{El}(\mathtt{W}(\tau))[\mathtt{p}^1] \right\} \\
&\quad \to \left(\gamma^\bullet \in \mathtt{Sub}^\bullet \cdot^\bullet\ (\cdot^\bullet, {}^\bullet\ \mathtt{w}\pi_2^{j\bullet}(\tau^\bullet)[\mathtt{id}^\bullet, {}^\bullet a^\bullet]\ ) \to P\,(b[\gamma]) \right) \\
&\quad \to P\,(\mathtt{Wsup}_j(\tau, a, b)) \Bigg) \\
&\to \cdot \vdash t : \mathtt{El}(\mathtt{W}(\tau)) \to W^C\ \tau^\bullet\ t \to P\,t \\
W^C rec\ \tau^\bullet\ P\ f\ t\ &(W^C sup\ a^\bullet\ b^\bullet) = f\ a^\bullet\ b\ (\lambda\gamma^\bullet. W^C rec\ \tau^\bullet\ P\ f\ (b[\gamma])\ (b^\blacktriangleright\ \gamma^\bullet))
\end{aligned}$$

Note that in $W^C sup$, the $b^\bullet$ uses the definition of $\llbracket \mathtt{W}(\tau) \rrbracket_C$, which after unfolding, recursively references W^C in a strictly positive position. We do not distinguish (b, b^c) and $b^\bullet$ for simplicity.

The idea of the proof of W-type is, as mentioned, mirroring the facility of the inductive type in the ambient logic into FMLTT$^{2.0}$. The main difference between W^C in the ambient logic and $\mathtt{W}(\cdot)$ in the FMLTT$^{2.0}$, is that W^C is only witnessing those reducible *closed terms*. Thus when using W^C to model $\mathtt{W}(\cdot)$, we need to do closed substitution properly.

Again, we omit validating that our model respects the equational/quotient rules (β, η, and substitution) here.

Now, our fundamental theorem can cover the whole FMLTT$^{2.0}$.

Theorem 6 (Fundamental Theorem for FMLTT$^{2.0}$). If $\Gamma \vdash t : T$, then its semantic interpretation is a dependent function such that $\llbracket t \rrbracket_C : \prod_{\cdot \vdash \gamma : \Gamma} \prod_{\gamma' \in \llbracket \Gamma \vdash \rrbracket_C(\gamma)} \llbracket \Gamma \vdash T \rrbracket_C(\gamma)(\gamma')(t[\gamma])$.

A.3.6 Canonical Forms in FMLTT$^{2.0}$. Besides the canonical forms for $\mathbb{B}$ (e.g. $\mathtt{tt}$ and $\mathtt{ff}$), we also have the following canonical forms for other type:

Theorem 7 (Canonical Forms).

- If $\cdot \vdash t : \texttt{El}(\texttt{W}(\tau))$ and $\cdot \vdash \tau\ \texttt{WSig}^n$, then $\cdot \vdash t \equiv \texttt{Wsup}_j(\tau, a, b) : \texttt{El}(\texttt{W}(\tau))$ for some $\cdot \vdash a : A$, $B[(\texttt{id}, a)] \vdash b : \texttt{El}(\texttt{W}(\tau))$, and $j < n$
- If $\cdot \vdash t : \mathbb{B}$ then $\cdot \vdash t \equiv \texttt{tt} : \mathbb{B}$ or $\cdot \vdash t \equiv \texttt{ff} : \mathbb{B}$
- If $\cdot \vdash t : \mathbb{L}(v^\bullet)$ then $\cdot \vdash t \equiv \mu^\bullet : \mathbb{L}(v^\bullet)$
- If $\cdot \vdash t : \mathbb{L}(\sigma)$ with $\cdot \vdash \sigma\ \texttt{LSig}^n$, then $\cdot \vdash t \equiv \mu^+(o, t) : \mathbb{L}(\sigma)$ for some $\cdot \vdash o : \mathbb{L}(\mu\pi_1(\sigma))$ and $v\pi_1'(\sigma) \vdash t : \mu\pi_2(\sigma)$
- If $\cdot \vdash t : \Sigma(A, B)$ then $\cdot \vdash t \equiv (a, b) : \Sigma(A, B)$ with $\cdot \vdash a : A$ and $\cdot \vdash b : B[(\texttt{id}, a)]$

A.4 Compile Linkage to Module in FMLTT$^{2.0}$

The structure of linkage in FMLTT$^{2.0}$ is flexible enough such that every field can be arbitrarily modified, as it is ultimately just a list of terms. However, it is far from the programming interface provided by Rocqet, where the programmer mainly interact with the compiled module of the linkage.

In this section, we show how such compilation procedure can be encoded in FMLTT$^{2.0}$: we will introduce several new constructs like $\texttt{Seals}$, $\mathbb{P}(\cdot)$, $\mathbb{p}(\cdot)$, that are based on the core calculus of FMLTT$^{2.0}$.

We start by defining two (dependent) functions $\texttt{Seals}$ and $\mathbb{P}(\cdot,\cdot)$ mutually recursively. They all take derivations of (different) judgments as input, but $\texttt{Seals}$ will output new judgment while $\mathbb{P}(\cdot,\cdot)$ will out put new types. They are inductively defined, based on the length of the linkage signature, which we restrict[7] to natural numbers $n \in \mathbb{N} \cong \left\{ x \,\middle|\, \cdot \vdash x\ \texttt{Nat} \right\}$ due to the canonicity result.

Moreover, we have $n \in \mathbb{N} \mapsto n[\epsilon] \in \left\{ x \,\middle|\, \Gamma \vdash x\ \texttt{Nat} \right\}$, so we will consider $n \in \mathbb{N} \subseteq \left\{ x \,\middle|\, \Gamma \vdash x\ \texttt{Nat} \right\}$ for simplicity. For notational simplicity, we will also omit some arguments when using these two functions.

$$
\begin{aligned}
\texttt{Seals} &: \forall(n : \mathbb{N}), \left\{ \sigma \,\middle|\, \Gamma \vdash_i \sigma\ \texttt{LSig}^n \right\} \to \texttt{Set} \\
\mathbb{P}(\cdot) &: \forall(n : \mathbb{N}), \left\{ \sigma \,\middle|\, \Gamma \vdash_i \sigma\ \texttt{LSig}^n \right\} \to \texttt{Seals}(\sigma) \to \left\{ A \,\middle|\, \Gamma \vdash_i A \right\} \\
\texttt{Seals}(\{n = 0\}, \sigma) &= \{\star\} \\
\texttt{Seals}(\{n = n' + 1\}, \sigma) &= \sum_{\texttt{sls} : \texttt{Seals}(v\pi_1(\sigma))} \left\{ t \,\middle|\, \Gamma, \mathbb{P}(v\pi_1(\sigma), \texttt{sls}) \vdash t : v\pi_1'(\sigma)\left[\texttt{p}^1\right] \right\} \\
\mathbb{P}(\{n = 0\}, \sigma, \_) &= \top \\
\mathbb{P}(\{n = n' + 1\}, \sigma, (S, f)) &= \Sigma(\mathbb{P}(v\pi_1(\sigma), S) \quad , v\pi_2(\sigma)\left[\texttt{p}^1, f\right])
\end{aligned}
$$

By inspection, we can see $\mathbb{P}(\cdot)$ is a deeply nested existential type, which represents Rocq modules. Similarly, we will construct the compilation procedure $\mathbb{p}(\cdot)$, inductively on the (literal) length of the signature:

$$
\begin{aligned}
\mathbb{p}(\cdot) : \ & \forall(n : \mathbb{N}), \left\{ \sigma \,\middle|\, \Gamma \vdash_i \sigma\ \texttt{LSig}^n \right\} \to \texttt{Seals}(\sigma) \to \left\{ t \,\middle|\, \Gamma \vdash t : \mathbb{L}(\sigma) \right\} \to \left\{ t' \,\middle|\, \Gamma \vdash t' : \mathbb{P}(\sigma) \right\} \\
& \mathbb{p}(\{n = 0\}, \sigma, \_, \_) = () \\
& \mathbb{p}(\{n = n' + 1\}, \sigma, (S, f), t) = (u, \mu\pi_2(t)[\texttt{id}, f[\texttt{id}, u]]) \\
& \qquad \text{where } u = \mathbb{p}(v\pi_1(\sigma), S, \mu\pi_1(t))
\end{aligned}
$$

[7] That means these two functions can only work on linkage with literal length. They cannot handle dynamic length like $\Gamma \vdash \sigma.\texttt{size Nat}$. Thus, these two functions can handle closed signature/linkages, and open signature/linkage with known length.

Recall that a given signature σ_n is a list of types $\{A_i \vdash T_i\}_{i \le n}$, so we have a list of signatures as well $\sigma_1, \sigma_2, .., \sigma_i$ each indicating first i fields in the signature σ_n. Intuitively, $\mathbb{P}(\sigma_i)$ computes a specific nested Sigma type and `Seals` (σ_i) shows how to map from $\mathbb{P}(\sigma_j)$ to A_j for each $j < i$.

Also recall that $\ell \in \mathbb{P}(\sigma_n)$ is intuitively a list of terms $\left\{ t_i \;\middle|\; \Gamma, A_i \vdash t_i : T_i \right\}$. Then utilizing those mappings in `sls : Seals`(σ_i) and the list of terms in $\ell \in \mathbb{P}(\sigma_i)$, we can instantiate each A_i and get a term $\mathbb{p}(\ell, \texttt{sls})$ of the deeply nested Sigma type $\mathbb{P}(\sigma, \texttt{sls})$.

B Using FMLTT$^{2.0}$ to Model Families and Inheritance

B.1 Example: Linkage encoding in FMLTT$^{2.0}$

The above formulation of compilation might be too abstract. Here we give a concrete example on a family and its compilation to modules in FMLTT$^{2.0}$. This example is (the skeleton of) the base STLC family, which includes an inductive type and recursion and one normal fields utilizing the recursion function.

This example forms as a template as how FMLTT$^{2.0}$ can encode Rocqet's family facility. Our example will use named syntax for readability. We will show the derivation in between the Rocq code.

```
Family Λ→.
```
$$\Lambda^{\rightarrow}_{.0} = \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})$$
```
  FInductive tm : Set := | tm_unit : tm | tm_abs  : id → tm → tm.
```
$$\Lambda^{\rightarrow}_{.1} = \vdash \mu^{+}(\Lambda^{\rightarrow}_{.0}, \star) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \mathcal{W}(\tau_1)))$$
```
  FRecursion subst ...
```
$$\Lambda^{\rightarrow}_{.2.0} = \texttt{tm} : \mathcal{J}^{c}(\tau_1) \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})$$
```
    Case tm_unit := λ x t. tm_unit
```
$$\Lambda^{\rightarrow}_{.2.1} = \texttt{tm} : \mathcal{J}^{c}(\tau_1) \vdash \mu^{+}(\Lambda^{\rightarrow}_{.2.0}, \lambda\, x\, t.\texttt{tm.unit}) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \texttt{CaseTy}(\cdots, , \mathcal{R})))$$
```
    Case tm_lam  := λ x t. ...
```
$$\Lambda^{\rightarrow}_{.2.2} = \texttt{tm} : \mathcal{J}^{c}(\tau_1) \vdash \mu^{+}(\Lambda^{\rightarrow}_{.2.1}, \cdots) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \texttt{CaseTy}(\cdots, , \mathcal{R})))$$
```
  End subst.
```
$$\Lambda^{\rightarrow}_{.2} = \vdash \mu^{+}(\Lambda^{\rightarrow}_{.1}, \Lambda^{\rightarrow}_{.2.2}) : \mathbb{L}(\nu^{+}(\cdots; \mathcal{J}^{c}(\tau_1) \vdash \texttt{RecSig}(\tau_1, \mathcal{R})))$$
$$\Lambda^{\rightarrow}_{.3.0} = \{\texttt{tm} : \mathbb{S}(\mathcal{W}(\tau_1)), \texttt{rL} : \mathbb{L}(\texttt{RecSig}(\tau_1, \mathcal{R}))\}^{\Sigma} \vdash \star : \mathbb{S}(\lambda\, w.\texttt{Wrec}(\tau_1, \texttt{rL}, w))$$
$$\Lambda^{\rightarrow}_{.3} = \vdash \mu^{+}(\Lambda^{\rightarrow}_{.2}, \Lambda^{\rightarrow}_{.3.0}) : \mathbb{L}(\nu^{+}(\cdots; \{\texttt{tm} : \mathbb{S}(\mathcal{W}(\tau_1)), \texttt{rL} : \mathbb{L}(\texttt{RecSig}(\tau_1, \mathcal{R}))\}^{\Sigma} \vdash \mathbb{S}(\lambda\, w.\texttt{Wrec}(\cdots, \cdots, w))))$$
```
  FDefinition test := subst tm_unit "v" tm_unit.
```
$$\Lambda^{\rightarrow}_{.4.0} = \{\texttt{tm} : \mathcal{J}^{c}(\tau_1), R : \texttt{tm} \to \mathcal{R}\}^{\Sigma} \vdash R(\texttt{tm.unit}, \text{"}\,v\,\text{"}, \texttt{tm.unit}) : \texttt{tm}$$
$$\Lambda^{\rightarrow}_{.4} = \vdash \mu^{+}(\Lambda^{\rightarrow}_{.3}, \Lambda^{\rightarrow}_{.4.0}) : \mathbb{L}(\nu^{+}(\cdots; \{\texttt{tm} : \mathcal{J}^{c}(\tau_1), R : \texttt{tm} \to \mathcal{R}\}^{\Sigma} \vdash \texttt{tm}))$$
```
End Λ→.
```
$$\Lambda^{\rightarrow} = \Lambda^{\rightarrow}_{.4}$$

Some comments are needed for explaining notation and this linkage construction.

(1) We use $\{\cdots\}^{\Sigma}$ as an easy shorthand for dependent pairs.
For example $\{\texttt{tm} : \mathbb{U}, \texttt{tm\_unit} : \texttt{tm}, \texttt{tm\_abs} : \texttt{ident} \to \texttt{tm} \to \texttt{tm}\}^{\Sigma}$ is exactly $\Sigma(\texttt{tm} : \mathbb{U}, \Sigma(\texttt{tm\_unit} : \texttt{tm}, \texttt{ident} \to \texttt{tm} \to \texttt{tm}))$, a type supports two constructors `tm_unit` and `tm_abs`.

(2) τ_1 is the W-type signature of an inductive type with only two constructors, `tm_unit` and `tm_abs`. For this example, $\tau_1 = \texttt{w}^{+}(\texttt{w}^{\bullet}, \top, \bot), \texttt{ident}, \top)$

(3) $\mathscr{I}^c(\tau_1)$ is the inductive type without exposing its eliminator. In other words, $\mathscr{I}^c(\tau_1) \cong \{\mathtt{tm} : \mathbb{U}, \mathtt{tm\_unit} : \mathtt{tm}, \mathtt{tm\_abs} : \mathtt{ident} \to \mathtt{tm} \to \mathtt{tm}\}^\Sigma$. We automatically have $\mathtt{W}(\tau) \to \mathscr{I}^c(\tau)$ for arbitrary τ.
(4) Compared to $\mathtt{W}(\tau_1)$, $\mathscr{I}^c(\tau_1)$ does not have recursor; while we can recurse on a term of $t : \mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_1))$ using $\mathtt{Wrec}(\tau_1, \cdot, .)$
(5) Compared to $\mathtt{W}(\cdot)$, $\mathscr{I}^c(\cdot)$ has a mapping between related inductive signatures. For example, if $\tau_2 = \mathtt{w}^+(\tau_1, \top, \bot)$ to support $\mathtt{tm\_true}$ constructor, then we will have a map $\mathscr{I}^c(\tau_2) \to \mathscr{I}^c(\tau_1)$, which maps out the two of the constructors and the type in $\mathscr{I}^c(\tau_2)$.
(6) $\mathscr{R}$ is short for a type of the form $\mathtt{ident} \to \mathtt{tm} \to \mathtt{tm}$. So $\mathtt{tm} \to \mathscr{R}$ is exactly the signature of substitution

To make this example more readable, we focus on the type and signature of our linkage to draw a big picture on the components of the encoding. We also use $\{\cdot\}^{\mathtt{Sig}}$ to indicate the linkage signature.

We have $\cdot \vdash \Lambda^{\to} : \mathbb{L}(\sigma^{\to})$, where

$$
\begin{aligned}
\sigma^{\to} = \{&\mathtt{tm} : && \top \vdash \mathbb{S}(\mathtt{W}(\tau_1));\\
&\mathtt{rL} : && \{\mathtt{tm} : \mathscr{I}^c(\tau_1)\}^\Sigma \vdash \mathtt{RecSig}(\tau_1, \mathtt{tm} \to \mathscr{R})\\
&\mathtt{subst} : && \{\mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_1)), \mathtt{rL} : \mathbb{L}(\mathtt{RecSig}(\tau_1, \mathscr{R}))\}^\Sigma \vdash \mathbb{S}(\lambda\, w.\mathtt{Wrec}(\tau_1, \mathtt{rL}, w))\\
&\mathtt{test} : && \{\mathtt{tm} : \mathscr{I}^c(\tau_1), \mathtt{R} : \mathtt{tm} \to \mathscr{R}\}^\Sigma \vdash \mathtt{tm}\\
\}^{\mathtt{Sig}}&
\end{aligned}
$$

Some more remarks on this linkage encoding in FMLTT$^{2.0}$:

(1) To construct a recursive function, we always split the definition of case handlers (e.g., $\mathtt{rL}$) and the utilization of eliminators (e.g., $\mathtt{subst}$ using $\mathtt{Wrec}(\cdot, \cdot,)$) into two fields. Because the former can be considered as normal fields for further code reuse and inheritance, while the later is closely related to a particular inductive type $\mathtt{W}(\tau)$, and cannot be reused once inductive signature is changed.
(2) When defining inductive type (e.g., $\mathtt{tm}$) and utilizing eliminator (e.g., $\mathtt{subst}$), we always make the definition of these fields explicit at type level using singleton type $\mathbb{S}(\cdot)$. This is for later extension—the extension of these two kinds of fields are encoded using ${}^I\mathtt{Ov}^2(\cdot, \cdot, \cdot)$. In other words, there is no internal structure of inductive type and the utilization of eliminator that can be reused—during inheritance, we simply replace them using enriched fields. This behavior is aligned with the implementation of Rocqet—to type-check an extended inductive type, the only way is to reflect the source code and manually add new constructors and re-type-check all the constructors.
(3) The case handlers (e.g., $\mathtt{rL}$) can be fully reused since they are not closely related to any particular inductive type. Particularly, they can be reused when τ_1 is extended to $\mathtt{w}^+(\tau_1, \cdots, \cdots)$.
(4) For other normal fields (i.e., those fields that are not inductive type definition or using eliminators) (e.g., $\mathtt{rL}$ and $\mathtt{test}$), they will abstract prior fields by treating prior inductive type definition as $\mathscr{I}^c(\tau)$ and prior recursive functions as normal functions (e.g., $\{\mathtt{tm} : \mathscr{I}^c(\tau_1), \mathtt{R} : \mathtt{tm} \to \mathscr{R}\}$). This makes these fields (e.g., $\mathtt{rL}$ and $\mathtt{test}$) reusable in extended inductive definition because we have a map $\mathscr{I}^c(\mathtt{w}^+(\tau, \cdots, \cdots)) \to \mathscr{I}^c(\tau)$.

B.2 Example : Compilation encoding in FMLTT$^{2.0}$

Now we are clear about how FMLTT$^{2.0}$ encodes some basic family examples, we will construct a sealing $f : \mathtt{Seals}(\sigma^{\to})$ to show our linkage encoding can be compiled to a module (Sigma type). At the same time, we will have $\mathbb{P}(\sigma^{\to}, f)$ to see the final module signature of the compilation.

We will denote $\sigma_i^{\rightarrow}$ as the signature of first i fields. And we can do the following computation

$$\texttt{Seals}(\sigma_1^{\rightarrow}) = \sum_{f_0 \in \{\star\}} \left\{ t \,\middle|\, \vdash t : \top \right\}$$
$$\text{where} \quad f_1 = (\star, \vdash () : \top) \in \texttt{Seals}(\sigma_1^{\rightarrow})$$
$$\mathbb{P}(\sigma_1^{\rightarrow}, f_1) = \{\_ : \top, \texttt{tm} : \mathbb{S}(\texttt{W}(\tau_1))\}^{\Sigma}$$
$$\texttt{Seals}(\sigma_2^{\rightarrow}) = \sum_{f_1 \in \texttt{Seals}(\sigma_1^{\rightarrow})} \left\{ t \,\middle|\, \mathbb{P}(\sigma_1^{\rightarrow}, f_1) \vdash t : \mathscr{I}^c(\tau_1) \right\}$$
$$\text{where} \quad f_2 = (f_1, \mathbb{P}(\sigma_1^{\rightarrow}, f_1) \vdash \langle \texttt{tm}, \cdots \rangle : \mathscr{I}^c(\tau_1)) \in \texttt{Seals}(\sigma_2^{\rightarrow})$$
$$\mathbb{P}(\sigma_2^{\rightarrow}, f_2) = \{\_ : \top, \texttt{tm} : \mathbb{S}(\texttt{W}(\tau_1)), \texttt{rL} : \texttt{RecSig}(\tau_1, \texttt{tm} \to \mathscr{R})\}^{\Sigma}$$
$$\texttt{Seals}(\sigma_3^{\rightarrow}) = \sum_{f_2 \in \texttt{Seals}(\sigma_2^{\rightarrow})} \left\{ t \,\middle|\, \mathbb{P}(\sigma_2^{\rightarrow}, f_2) \vdash t : \{\texttt{tm} : \mathbb{S}(\texttt{W}(\tau_1)), \texttt{rL} : \mathbb{L}(\texttt{RecSig}(\tau_1, \mathscr{R}))\}^{\Sigma} \right\}$$
$$\text{where} \quad f_3 = (f_2, \mathbb{P}(\sigma_2^{\rightarrow}, f_2) \vdash \langle \texttt{tm}, \texttt{rL} \rangle : \cdots) \in \texttt{Seals}(\sigma_3^{\rightarrow})$$
$$\mathbb{P}(\sigma_3^{\rightarrow}, f_3) = \{\_ : \top, \texttt{tm} : \mathbb{S}(\texttt{W}(\tau_1)), \texttt{rL} : \texttt{RecSig}(\tau_1, \texttt{tm} \to \mathscr{R}), \texttt{subst} : \mathbb{S}(\lambda\, w.\texttt{Wrec}(\tau_1, \texttt{rL}, w))\}^{\Sigma}$$
$$\texttt{Seals}(\sigma_4^{\rightarrow}) = \sum_{f_3 \in \texttt{Seals}(\sigma_3^{\rightarrow})} \left\{ t \,\middle|\, \mathbb{P}(\sigma_3^{\rightarrow}, f_3) \vdash t : \{\texttt{tm} : \mathscr{I}^c(\tau_1), \texttt{subst} : \texttt{tm} \to \mathscr{R}\}^{\Sigma} \right\}$$
$$\text{where} \quad f_4 = (f_3, \mathbb{P}(\sigma_3^{\rightarrow}, f_3) \vdash \langle \texttt{tm}, \texttt{subst} \rangle : \cdots) \in \texttt{Seals}(\sigma_4^{\rightarrow})$$
$$\mathbb{P}(\sigma_4^{\rightarrow}, f_4) = \{\cdots, \texttt{tm} : \mathbb{S}(\texttt{W}(\tau_1)), \texttt{rL} : \texttt{RecSig}(\tau_1, \texttt{tm} \to \mathscr{R}), \texttt{subst} : \mathbb{S}(\lambda\, w.\texttt{Wrec}(\tau_1, \texttt{rL}, w)), \texttt{test} : \texttt{tm}\}^{\Sigma}$$

Most f_i are simply just picking correct terms from the context; when inductive type is involved, we also need to pick and abstract corresponding constructors. So in Rocqet implementation, the construction of f_i is automatic.

The compilation result has the type $\mathbb{P}(\sigma_4^{\rightarrow}, f_4)$, carrying information as we expect.

B.3 Example: Nested Inheritance encoding in FMLTT$^{2.0}$

Here we show how our FMLTT$^{2.0}$ encodes a more complete example in Rocqet—we focus on four families, base STLC family, two extension on tuple and boolean respectively, and their composition. These four families are group into two families. The first base family has been illustrated in the earlier sealing example. For the sake of presentation simplicity, some of the derivations might be omitted and inferrable according to the context. In the derivation, we still use named convention for better readability.

We will show the derivation in between the code.

```
Family Λ¹. (* The first language family, includes arrow and boolean type. *)
  Family Λ→.
```
$\Lambda^{\rightarrow}_{.0} = \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})$
```
  FInductive tm : Set := | tm_unit : tm | tm_abs  : id → tm → tm.
```
$\Lambda^{\rightarrow}_{.1} = \vdash \mu^{+}(\Lambda^{\rightarrow}_{.0}, \star) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \mathit{W}(\tau_1)))$
```
  FRecursion subst ...
```
$\Lambda^{\rightarrow}_{.2.0} = \mathit{tm} : \mathscr{I}^c(\tau_1) \vdash \mu^{\bullet} : \mathbb{L}(\nu^{\bullet})$
```
    Case tm_unit := λ x t. tm_unit
```
$\Lambda^{\rightarrow}_{.2.1} = \mathit{tm} : \mathscr{I}^c(\tau_1) \vdash \mu^{+}(\Lambda^{\rightarrow}_{.2.0}, \lambda\, x\, t.\mathit{tm.unit}) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \mathit{CaseTy}(\cdots, , \mathscr{R})))$
```
    Case tm_lam  := λ x t. ...
```
$\Lambda^{\rightarrow}_{.2.2} = \mathit{tm} : \mathscr{I}^c(\tau_1) \vdash \mu^{+}(\Lambda^{\rightarrow}_{.2.1}, \cdots) : \mathbb{L}(\nu^{+}(\cdots; \top \vdash \mathit{CaseTy}(\cdots, , \mathscr{R})))$
```
  End subst.
```

$$\Lambda^{\to}{}_{.2} = \vdash \mu^{+}(\Lambda^{\to}{}_{.1}, \Lambda^{\to}{}_{.2.2}) : \mathbb{L}(\nu^{+}(\cdots; \mathcal{J}^{c}(\tau_1) \vdash \mathit{RecSig}(\tau_1, \mathscr{R})))$$

$$\Lambda^{\to}{}_{.3.0} = \{\mathit{tm} : \mathbb{S}(W(\tau_1)), \mathit{rL} : \mathbb{L}(\mathit{RecSig}(\tau_1, \mathscr{R}))\}^{\Sigma} \vdash \star : \mathbb{S}(\lambda\, w.\mathit{Wrec}(\tau_1, \mathit{rL}, w))$$

$$\Lambda^{\to}{}_{.3} = \vdash \mu^{+}(\Lambda^{\to}{}_{.2}, \Lambda^{\to}{}_{.3.0}) : \mathbb{L}(\nu^{+}(\cdots; \{\mathit{tm} : \mathbb{S}(W(\tau_1)), \mathit{rL} : \mathbb{L}(\mathit{RecSig}(\tau_1, \mathscr{R}))\}^{\Sigma} \vdash \mathbb{S}(\lambda\, w.\mathit{Wrec}(\cdots, \cdots, w))))$$

```
FDefinition test := subst tm_unit "v" tm_unit.
```

$$\Lambda^{\to}{}_{.4.0} = \{\mathit{tm} : \mathcal{J}^{c}(\tau_1), R : \mathit{tm} \to \mathscr{R}\}^{\Sigma} \vdash R(\mathit{tm.unit}, \text{`` v''}, \mathit{tm.unit}) : \mathit{tm}$$

$$\Lambda^{\to}{}_{.4} = \vdash \mu^{+}(\Lambda^{\to}{}_{.3}, \Lambda^{\to}{}_{.4.0}) : \mathbb{L}(\nu^{+}(\cdots; \{\mathit{tm} : \mathcal{J}^{c}(\tau_1), R : \mathit{tm} \to \mathscr{R}\}^{\Sigma} \vdash \mathit{tm}))$$

```
End Λ→.
```

$$\Lambda^{\to} = \Lambda^{\to}{}_{.4}$$

$$\Lambda^{1}{}_{.1} = \vdash \mu^{+}(\Lambda^{\to}{}_{.0}, \Lambda^{\to}) : \nu^{+}(\nu^{\bullet}; \top \vdash \mathbb{L}(\cdots))$$

```
Family ΛB extends Λ→.
  FInductive tm : Set := | tm_true .
```

$$\mathbf{I}^{\mathbb{B}}{}_{.1} = \vdash {}^{I}\mathit{Ov}^{2}({}^{I}\mathit{Id}, W(\tau_1), W(\tau_2)) : (\Lambda^{\to}{}_{.1})^{\sigma} \twoheadrightarrow \nu^{+}(\nu^{\bullet}; \top \vdash \mathbb{S}(W(\tau_2)))$$

```
  FRecursion subst.
```

$$\mathbf{I}^{\mathbb{B}}{}_{.2.0} = \mathit{tm} : \mathcal{J}^{c}(\tau_2) \vdash {}^{I}\mathit{Id} : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow (\Lambda^{\to}{}_{.2})^{\sigma}$$

```
    Case tm_true :=  λ x t. tm_true.
```

$$\mathbf{I}^{\mathbb{B}}{}_{.2.1} = \mathit{tm} : \mathcal{J}^{c}(\tau_2) \vdash {}^{I}\mathit{Ext}(\mathbf{I}^{\mathbb{B}}{}_{.2.0}, \lambda\, \_\ \_.\mathit{tm.true}) : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \top \vdash \mathit{CaseTy}(\cdots, \cdots, \mathscr{R}))$$

```
  End subst.
```

$$\mathbf{I}^{\mathbb{B}}{}_{.2} = \vdash {}^{I}\mathit{Nest}(\mathbf{I}^{\mathbb{B}}{}_{.1}, \cdots, \mathbf{I}^{\mathbb{B}}{}_{.2.1}) : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \mathcal{J}^{c}(\tau_2) \vdash \mathbb{L}(\mathit{RecSig}(\tau_2, \mathscr{R})))$$

$$\mathbf{I}^{\mathbb{B}}{}_{.3} = \vdash {}^{I}\mathit{Ov}^{2}(\mathbf{I}^{\mathbb{B}}{}_{.2}, \{\mathit{tm} : \mathbb{S}(W(\tau_2)), \mathit{rL} : \mathbb{L}(\mathit{RecSig}(\tau_2, \mathscr{R}))\}^{\Sigma}, \lambda\, w.\mathit{Wrec}(\tau_2, \mathit{rL}, w)) : (\Lambda^{\to}{}_{.3})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \cdots \vdash \cdots)$$

```
  (* Field test inherited. *)
```

$$\mathbf{I}^{\mathbb{B}}{}_{.4} = \vdash {}^{I}\mathit{Inherit}(\mathbf{I}^{\mathbb{B}}{}_{.3}) : (\Lambda^{\to}{}_{.4})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \{\mathit{tm} : \mathcal{J}^{c}(\tau_1), R : \mathit{tm} \to \mathscr{R}\}^{\Sigma} \vdash \mathit{tm})$$

```
  End ΛB.
```

$$\Lambda^{\mathbb{B}} = \ell : \mathbb{L}^{+}((\Lambda^{\to}{}_{.4})^{\sigma}) \vdash \mathit{inh}^{\oplus}(\mathbf{I}^{\mathbb{B}}{}_{.4}, \ell) : \mathbb{L}(\mathcal{S}^{\oplus}(\ell.\mathit{inh}, \mathbf{I}^{\mathbb{B}}{}_{.4}))$$

```
End Λ1.
```

$$\Lambda^{1} = \vdash \mu^{+}(\Lambda^{1}{}_{.1}, \Lambda^{\mathbb{B}}) : \nu^{+}(\cdots; \ell : \mathbb{L}^{+}((\Lambda^{\to}{}_{.4})^{\sigma}) \vdash \mathbb{L}(\mathcal{S}^{\oplus}(\ell.\mathit{inh}, \mathbf{I}^{\mathbb{B}}{}_{.4})))$$

```
Family Λ2 extends Λ1. (* We will extend language with pair/tuple here. *)
```

$$\mathbf{I}^{2}{}_{.0} = \vdash {}^{I}\mathit{Id} : \nu^{\bullet} \twoheadrightarrow \nu^{\bullet}$$

```
  Family Λ→. (* We use I^× to denote this linkage transformer in the derivation. *)
    FInductive tm : Set := | tm_pair : tm → tm → tm.
```

$$\mathbf{I}^{\times}{}_{.1} = \vdash {}^{I}\mathit{Ov}^{2}({}^{I}\mathit{Id}, W(\tau_1), W(\tau_3)) : (\Lambda^{\to}{}_{.1})^{\sigma} \twoheadrightarrow \nu^{+}(\nu^{\bullet}; \top \vdash \mathbb{S}(W(\tau_3)))$$

```
    FRecursion subst.
```

$$\mathbf{I}^{\times}{}_{.2.0} = \mathit{tm} : \mathcal{J}^{c}(\tau_3) \vdash {}^{I}\mathit{Id} : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow (\Lambda^{\to}{}_{.2})^{\sigma}$$

```
      Case tm_pair := λ a rec_a b rec_b x t. tm_pair (rec_a x t) (rec_b x t)
```

$$\mathbf{I}^{\times}{}_{.2.1} = \mathit{tm} : \mathcal{J}^{c}(\tau_3) \vdash {}^{I}\mathit{Ext}(\mathbf{I}^{\times}{}_{.2.0}, \cdots) : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \top \vdash \mathit{CaseTy}(\cdots, \cdots, \mathscr{R}))$$

```
    End subst.
```

$$\mathbf{I}^{\times}{}_{.2} = \vdash {}^{I}\mathit{Nest}(\mathbf{I}^{\times}{}_{.1}, \cdots, \mathbf{I}^{\times}{}_{.2.1}) : (\Lambda^{\to}{}_{.2})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \mathcal{J}^{c}(\tau_3) \vdash \mathbb{L}(\mathit{RecSig}(\tau_3, \mathscr{R})))$$

$$\mathbf{I}^{\times}{}_{.3} = \vdash {}^{I}\mathit{Ov}^{2}(\mathbf{I}^{\times}{}_{.2}, \{\mathit{tm} : \mathbb{S}(W(\tau_3)), \mathit{rL} : \mathbb{L}(\mathit{RecSig}(\tau_3, \mathscr{R}))\}^{\Sigma}, \lambda\, w.\mathit{Wrec}(\tau_3, \mathit{rL}, w)) : (\Lambda^{\to}{}_{.3})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \cdots \vdash \cdots)$$

```
    (* Field test inherited. *)
```

$$\mathbf{I}^{\times}{}_{.4} = \vdash {}^{I}\mathit{Inherit}(\mathbf{I}^{\times}{}_{.3}) : (\Lambda^{\to}{}_{.4})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \{\mathit{tm} : \mathcal{J}^{c}(\tau_1), R : \mathit{tm} \to \mathscr{R}\}^{\Sigma} \vdash \mathit{tm})$$

```
    End Λ→.
```

$$\mathbf{I}^{2}{}_{.1} = \vdash {}^{I}\mathit{Nest}(\mathbf{I}^{2}{}_{.0}, \cdots, \mathbf{I}^{\times}{}_{.4}) : (\Lambda^{1}{}_{.1})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \top \vdash \mathbb{L}(\cdots))$$

```
  (* Family ΛB inherited/mixin-ed. We use I^B× to denote this linkage tranformer. *)
```

$$\mathbf{I}^{\mathbb{B}\times}{}_{.1} = \vdash {}^{I}\mathit{Ov}^{2}({}^{I}\mathit{Id}, \cdots, \mathbb{S}(\mathtt{W}(\tau_4))) : (\Lambda^{\rightarrow}{}_{..1})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \top \vdash \mathbb{S}(\mathtt{W}(\tau_4)))$$

(* FRecursion `subst` inherited/mixin-ed. *)

$$\mathbf{I}^{\mathbb{B}\times}{}_{.2} = \vdash {}^{I}\mathit{Inherit}(\mathbf{I}^{\mathbb{B}\times}{}_{.1}) : (\Lambda^{\rightarrow}{}_{..2})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \cdots \vdash \mathbb{L}(\cdots))$$

$$\mathbf{I}^{\mathbb{B}\times}{}_{.3} = \vdash {}^{I}\mathit{Ov}^{2}(\mathbf{I}^{\mathbb{B}\times}{}_{.2}, \cdots, \cdots) : (\Lambda^{\rightarrow}{}_{..3})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \{\mathit{tm} : \mathbb{S}(\mathtt{W}(\tau_4)), \mathit{rL} : \mathbb{L}(\mathit{RecSig}(\tau_4, \mathscr{R}))\}^{\Sigma} \vdash \mathbb{S}(\lambda\, w.\mathit{Wrec}(\tau_4, \mathit{rL}, w)))$$

(* Field `test` inherited. *)

$$\mathbf{I}^{\mathbb{B}\times}{}_{.4} = \vdash {}^{I}\mathit{Inherit}(\mathbf{I}^{\mathbb{B}\times}{}_{.3}) : (\Lambda^{\rightarrow}{}_{..4})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \cdots \vdash \cdots)$$

$$\mathbf{I}^{\mathbb{B}\times} = \ell : \mathbb{L}^{+}((\Lambda^{\rightarrow}{}_{.4})^{\sigma}) \vdash \mathit{lift}_{\ell.\mathit{inh} \oplusplus \mathbf{I}^{\mathbb{B}}{}_{.4}}(\mathbf{I}^{\mathbb{B}\times}{}_{.4}) : \mathcal{S}^{\oplus}(\ell.\mathit{inh}, \mathbf{I}^{\mathbb{B}}{}_{.4}) \twoheadrightarrow \mathcal{S}^{\oplus}(\ell.\mathit{inh} \oplusplus \mathbf{I}^{\mathbb{B}}{}_{.4}, \mathbf{I}^{\mathbb{B}\times}{}_{.4})$$

$$\mathbf{I}^{2}{}_{.2} = \vdash {}^{I}\mathit{Nest}(\mathbf{I}^{2}{}_{.1}, , \mathbf{I}^{\mathbb{B}\times}) : (\Lambda^{1})^{\sigma} \twoheadrightarrow \nu^{+}(\cdots; \ell : \mathbb{L}^{+}((\Lambda^{\rightarrow}{}_{.4})^{\sigma}) \vdash \mathcal{S}^{\oplus}(\ell.\mathit{inh} \oplusplus \mathbf{I}^{\mathbb{B}}{}_{.4}, \mathbf{I}^{\mathbb{B}\times}{}_{.4}))$$

End Λ^2.

$$\Lambda^{2} = \vdash \mathit{inh}(\mathbf{I}^{2}{}_{.2}, \Lambda^{1}) : \mathbb{L}(\cdots)$$

Look at the signature of Λ^2:

$$\begin{aligned}
(\Lambda^{2})^{\sigma} = \{&\Lambda^{\times} : && \top \vdash \mathbb{L}((\Lambda^{\times})^{\sigma});\\
&\Lambda^{\mathbb{B}\times} : && \{\ell : \mathbb{L}^{+}((\Lambda^{\rightarrow})^{\sigma})\}^{\Sigma} \vdash \mathcal{S}^{\oplus}(\ell.\mathtt{inh} \oplusplus \mathbf{I}^{\mathbb{B}}{}_{.4}, \mathbf{I}^{\mathbb{B}\times}{}_{.4})\\
&\}^{\mathtt{Sig}} &&\\
(\Lambda^{\times})^{\sigma} = \{&\mathtt{tm} : && \top \vdash \mathbb{S}(\mathtt{W}(\tau_3));\\
&\mathtt{rL} : && \{\mathtt{tm} : \mathscr{T}^{c}(\tau_3)\}^{\Sigma} \vdash \mathtt{RecSig}(\tau_3, \mathtt{tm} \to \mathscr{R})\\
&\mathtt{subst} : && \{\mathtt{tm} : \mathbb{S}(\mathtt{W}(\tau_3)), \mathtt{rL} : \mathbb{L}(\mathtt{RecSig}(\tau_3, \mathscr{R}))\}^{\Sigma} \vdash \mathbb{S}(\lambda\, w.\mathtt{Wrec}(\tau_3, \mathtt{rL}, w))\\
&\mathtt{test} : && \{\mathtt{tm} : \mathscr{T}^{c}(\tau_3), \mathtt{R} : \mathtt{tm} \to \mathscr{R}\}^{\Sigma} \vdash \mathtt{tm}\\
&\}^{\mathtt{Sig}} &&
\end{aligned}$$

Now after the sealing and compilation, we can fill in the abstraction of $\Lambda^{2}{}_{..\Lambda^{\mathbb{B}\times}}$ (e.g. $\mathbb{L}^{+}((\Lambda^{\rightarrow})^{\sigma})$) with with $\mathbf{I}^{\times}{}_{.4}$, and resulting a linkage $\Lambda^{2}{}_{..\Lambda^{\mathbb{B}\times}}$ of signature $\mathcal{S}^{\oplus}(\mathbf{I}^{\times}{}_{.4} \oplusplus \mathbf{I}^{\mathbb{B}}{}_{.4}, \mathbf{I}^{\mathbb{B}\times}{}_{.4})$. After unfolding the definition and computation, we can see this linkage equips all the constructors and `subst` handles correctly and can be compiled.

Note that, when constructing $\mathbf{I}^{2}{}_{.2}$, besides the two expected linkage transformer $\mathbf{I}^{\times}{}_{.4}$ and $\mathbf{I}^{\mathbb{B}}{}_{.4}$, we also have an extra $\mathbf{I}^{\mathbb{B}\times}{}_{.4}$. The two $\mathbf{I}^{\times}{}_{.4}$ and $\mathbf{I}^{\mathbb{B}}{}_{.4}$ linkage transformers are reusing the main computational information (including the recursion body of `subst` and the last field). But the extra $\mathbf{I}^{\mathbb{B}\times}{}_{.4}$ is responsible for overriding the inductive signature $\mathtt{W}(\tau_4)$ and recursion call entrance $\mathtt{Wrec}(\cdots, \cdots, \cdots)$.