Certified Compilers à la Carte

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.

CCS Concepts: • **Software and its engineering** → *Compilers*; *Software verification*; *Inheritance*; *Polymorphism*; • **Theory of computation** → *Type theory*; *Logic and verification*.

Additional Key Words and Phrases: compiler verification, family polymorphism, extensibility, modularity, reuse, expression problem, mixins, late binding, dependent type theory, theorem proving, Rocq, CompCert, Rocqet.

ACM Reference Format:
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), 24 pages. https://doi.org/10.1145/3729261

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 [26] and CakeML [44], 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 [35, 32, 41, 20, 5, 36, 24]. The need for extensibility and composability is even more pressing in the case of certified

Authors' Contact Information: Oghenevwogaga Ebresafe, oghenevwogaga.ebresafe@uwaterloo.ca, Ian Zhao, i6zhao@uwaterloo.ca, Ende Jin, ende.jin@uwaterloo.ca, Arthur Bright, a2bright@uwaterloo.ca, Charles Jian, cjian@uwaterloo.ca, Yizhou Zhang, yizhou@uwaterloo.ca. All authors are with the School of Computer Science, University of Waterloo, Canada.

ACM 2475-1421/2025/6-ART162
https://doi.org/10.1145/3729261

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 [46], 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 [29].

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 [7, 23]. 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 [6, 7, 42, 21, 14, 18] to provide design patterns or language-level support for modularizing the development of mechanized proofs. Notably, our recent work [18] introduces an extensibility mechanism to the Rocq prover [39] (then called Coq), inspired by the object-oriented ideas of *late binding* and *family polymorphism* [12]. The resulting language design and implementation, called FPOP [18], 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.
- 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.

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

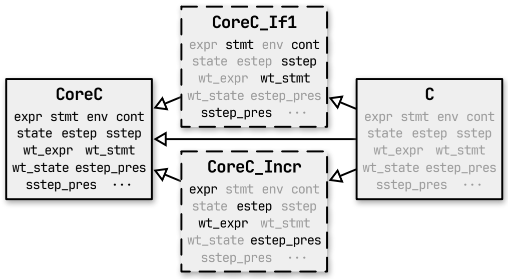
Rather than a single monolithic pass for all desugaring, a more modular approach follows the *nanopass* principle [41, 20]: structuring them as a collection of many small passes, each performing a single task. These smaller passes are easier to read, define, and verify. However, this approach can lead to excessive boilerplate code repeated across nanopasses. In fact, this repetition already occurs in some passes in CompCert and CakeML, as previously noted [7, 23], 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

[1]The et in Rocqet is French for &, suggesting à la carte composition of families and traits—also a nod to the J& language [33].
[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.



CoreC_If1
expr stmt env cont state estep sstep wt_expr wt_stmt wt_state estep_pres sstep_pres ⋯
CoreC
expr stmt env cont state estep sstep wt_expr wt_stmt wt_state estep_pres sstep_pres ⋯
C
expr stmt env cont state estep sstep wt_expr wt_stmt wt_state estep_pres sstep_pres ⋯
CoreC_Incr
expr stmt env cont state estep sstep wt_expr wt_stmt wt_state estep_pres sstep_pres ⋯

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 *)
```

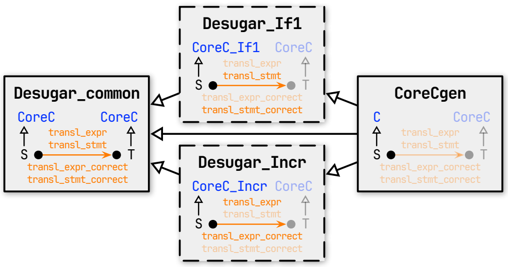
Figure 1. The source language C and its metatheories are mechanized by composing a shared component CoreC with extensions of CoreC (such as CoreC_If1 and CoreC_Incr).

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 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



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.

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 [18]—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 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 [41, 20] 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 [20, 23].

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 [32, 24].[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 [24], 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

[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.

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 `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 [46] 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* [31, 33, 47, 22]. 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* [25]) 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 [3, 9, 13, 1, 17, 34]. 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* [2]: 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 [18], 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., [15]).

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`.

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 [18], 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 [4]. 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`

```
                                        Comp/CoreC.v
1  ...
2  Module Type Comp°CoreC°Ctx.
3  End Comp°CoreC°Ctx.

5  Module Type Comp°CoreC°stmt°Ctx
6    (self[Comp]: Comp°CoreC°Ctx).
7  Include Comp°CoreC°expr°Ctx self[Comp].
8  Include Comp°CoreC°expr self[Comp].
9  End Comp°CoreC°stmt°Ctx.

11 Module Type Comp°CoreC°stmt
12   (self[Comp]: Comp°CoreC°Ctx)
13   (self[CoreC]: Comp°CoreC°stmt°Ctx
14                   self[Comp]).
15 Axiom stmt: Type.
16 Axiom Sskip: stmt
17 Axiom Sseq: stmt → stmt → stmt.
18 Axiom Sdo: self[CoreC].expr → stmt.
19 ...  (* declare other stmt constructors *)  ...
20 End Comp°CoreC°stmt.

22 Module Type Comp°CoreC°env°Ctx
23   (self[Comp]: Comp°CoreC°Ctx).
24 Include Comp°CoreC°stmt°Ctx self[Comp].
25 Include Comp°CoreC°stmt self[Comp].
26 End Comp°CoreC°env°Ctx.

28 Module Comp°CoreC°env
29   (self[Comp]: Comp°CoreC°Ctx)
30   (self[CoreC]: Comp°CoreC°env°Ctx
31                   self[Comp]).
32 Definition env := PTree.t
33   (self[Comp].block * self[Comp].type).
34 End Comp°CoreC°env.
```

```
35 ...  (* translate other fields of CoreC *)  ...

37 Module Type Comp°CoreC°Sig
38   (self[Comp]: Comp°CoreC°Ctx).
39 ...
40 Include Comp°CoreC°stmt self[Comp].
41 Include Comp°CoreC°env self[Comp].
42 ...
43 Module Type stmt°Art :=
44    Comp°CoreC°stmt.
45 ...
46 End Comp°CoreC°Sig.

48 Module Type Comp°CoreC
49   (self[Comp]: Comp°CoreC°Ctx).
50 Declare Module CoreC:
51    Comp°CoreC°Sig self[Comp].
52 End Comp°CoreC.

53                                  Comp/CoreC_If1.v
54 Module Type Comp°CoreC_If1°Ctx.
55 Include Comp°CoreC°Ctx.
56 Include Comp°CoreC.
57 End Comp°CoreC_If1°Ctx.
58 ...
59 Module Type Comp°CoreC_If1°stmt°Ctx
60   (self[Comp]: Comp°CoreC_If1°Ctx).
61 Include Comp°CoreC_If1°expr°Ctx
62          self[Comp].
63 Include Comp°CoreC_If1°expr self[Comp].
64 End Comp°CoreC_If1°stmt°Ctx.
```

```
65 Module Type Comp°CoreC_If1°stmt
66   (self[Comp]: Comp°CoreC_If1°Ctx)
67   (self[CoreC_If1]:
68     Comp°CoreC_If1°stmt°Ctx self[Comp]).
69 Include self[Comp].CoreC.stmt°Art
70          self[Comp] self[CoreC_If1].
71 Axiom Sif1:
72   self[CoreC_If1].expr → stmt → stmt.
73 End Comp°CoreC_If1°stmt.
74 ...

75 Module Type Comp°C°Ctx.         Comp/C.v
76 Include Comp°CoreC_Incr°Ctx.
77 Include Comp°CoreC_Incr.
78 End Comp°C°Ctx.
79 ...
80 Module Type Comp°C°stmt°Ctx
81   (self[Comp]: Comp°C°Ctx).
82 Include Comp°C°expr°Ctx self[Comp].
83 Include Comp°C°expr self[Comp].
84 End Comp°C°stmt°Ctx.

86 Module Type Comp°C°stmt
87   (self[Comp]: Comp°C°Ctx)
88   (self[C]: Comp°C°stmt°Ctx self[Comp]).
89 Include
90   self[Comp].Comp°CoreC.stmt°Art
91   self[Comp] self[C].
92 Include
93   self[Comp].Comp°CoreC_If1.stmt°Art
94   self[Comp] self[C].
95 End Comp°C°stmt.
96 ...
```

Figure 4. Translation of selected components from Figure 1.

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 [18], 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

```
1  …                                     Comp/Desugar_common.v
2  Module
3  Comp°Desugar_common°transl_stmt°Cases
4   (self[Comp]: Comp°Desugar_common°Ctx)
5   (self[Desugar_common]:
6    Comp°Desugar_common°S°Ctx self[Comp]).
7  Def transl_stmt°Sskip :=
8    ret self[Desugar_common].T.Sskip.
9  … (* translate other case handlers *) …
10 End
11 Comp°Desugar_common°transl_stmt°Cases.

13 Module Type
14 Comp°Desugar_common°transl_stmt
15  (self[Comp]: Comp°Desugar_common°Ctx)
16  (self[Desugar_common]: …).
17 Axiom transl_stmt :
18   self[Desugar_common].S.stmt →
19   res self[Desugar_common].T.stmt.
20 End Comp°Desugar_common°transl_stmt.

22 Module Type
23 Comp°Desugar_common°transl_stmt°CB
24  (self[Comp]: Comp°Desugar_common°Ctx)
25  (self[Desugar_common]: …).
26 Axiom transl_stmt_Sskip_eq:
27  self[Desugar_common].transl_stmt
28   self[Desugar_common].S.Sskip =
29  self[Desugar_common].transl_stmt°Sskip.
30 … (* axiomatize equalities for other cases *) …
31 End Comp°Desugar_common°transl_stmt°CB.
32 …
```

```
33 …                                        Comp/Desugar_If1.v
34 Module
35 Comp°Desugar_If1°transl_stmt°Cases
36  (self[Comp]: Comp°Desugar_If1°Ctx)
37  (self[Desugar_If1]: …).
38 Def transl_stmt°Sif1 e s IHs :=
39  do e' ←
40   self[Desugar_If1].transl_expr e;
41  do s' ← IHs;
42  ret (self[Desugar_If1].T.Sif1 e' s'
43       self[Desugar_If1].T.Sskip).
44 End Comp°Desugar_If1°transl_stmt°Cases.

46 Module Type
47 Comp°Desugar_If1°transl_stmt
48  (self[Comp]: Comp°Desugar_If1°Ctx)
49  (self[Desugar_If1]: …).
50 Include Comp°Desugar_common°transl_stmt
51  self[Comp] self[Desugar_If1].
52 End Comp°Desugar_If1°transl_stmt.

54 Module Type
55 Comp°Desugar_If1°transl_stmt°CB
56  (self[Comp]: Comp°Desugar_If1°Ctx)
57  (self[Desugar_If1]: …).
58 Axiom transl_stmt_Sif1_eq: ∀ e s IHs,
59  self[Desugar_If1].transl_stmt
60   (self[Desugar_If1].S.Sif1 e s) =
61  self[Desugar_If1].transl_stmt°Sif1
62   e s IHs.
63 End Comp°Desugar_If1°transl_stmt°CB.
64 …
```

```
65 …                                          Comp/CoreCgen.v
66 Module Type Comp°CoreCgen°Sig
67  (self[Comp]: Comp°CoreCgen°Ctx).
68 Module S := self[Comp].C.
69 Module T := self[Comp].CoreC.
70 …
71 Include
72  Comp°Desugar_common°transl_stmt°Cases
73  self[Comp].
74 Include
75  Comp°Desugar_If1°transl_stmt°Cases
76  self[Comp].
77 Module transl_stmt°Cases°Art := ….

79 Include
80  Comp°Desugar_common°transl_stmt°CB
81  self[Comp].
82 Include
83  Comp°Desugar_If1°transl_stmt°CB
84  self[Comp].
85 Module Type transl_stmt°CB°Art := ….
86 …
87 End Comp°CoreCgen°Sig.

89 Module Type Comp°CoreCgen
90  (self[Comp]: Comp°CoreCgen°Ctx).
91 Declare Module CoreCgen:
92  Comp°CoreCgen°Sig self[Comp].
93 End Comp°CoreCgen.
```

Figure 5. Translation of selected components from Figure 2.

`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.

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

```
1  Module Comp.
2   Module CoreC. … End CoreC.                      (* concretize CoreC *)
3
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.
13
14  Module Desugar_common. …          (* concretize Desugar_common *)
15  End Desugar_common.
16
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.
38
39  …                                    (* concretize other fields of Comp *)
40 End Comp.
```

```
41 Module CompX.
42  Module CoreC. … End CoreC.                      (* concretize CoreC *)
43
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.
54
55  Module Desugar_common. …          (* concretize Desugar_common *)
56  End Desugar_common.
57
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.
78
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.

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 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 [18] 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 [27] and builds on the formal model of FPOP [18], 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 appendices in the extended version of this paper [11].

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 [37]. 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.

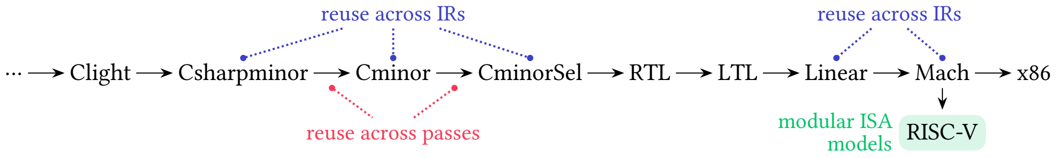

reuse across IRs
reuse across IRs
··· → Clight → Csharpminor → Cminor → CminorSel → RTL → LTL → Linear → Mach → x86
reuse across passes
modular ISA models
RISC-V

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

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 [40, 41, 20] 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. 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 [19]. 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.

```
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.
```

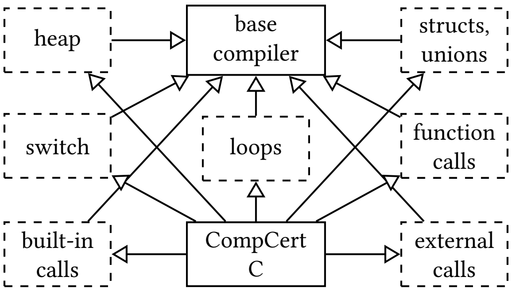


Figure 8. Rocqet enables à la carte composition of compiler extensions.

(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 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.

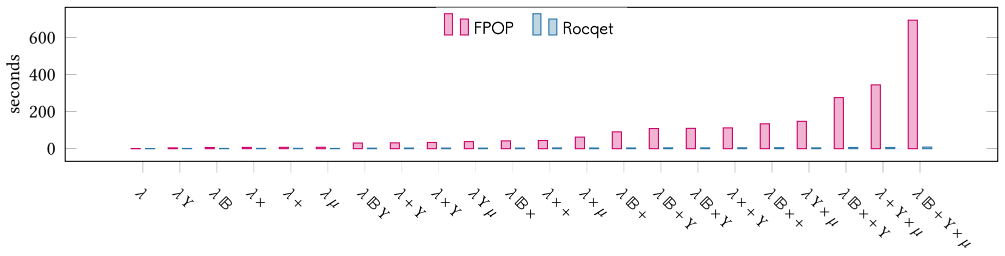


Figure 9. Proof-compilation times for different combinations of STLC extensions: FPOP vs. Rocqet.

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

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. [18] 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 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 [37].

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

[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

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.

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 [20]. 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 [38] 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 [26] and CakeML [44]. Tatlock and Lerner [45] and Gross et al. [16] 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 [6–8, 42, 21, 14, 18, 28], 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 [46] 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 [7, 8, 42, 21, 14] is inspired by the *data types à la carte* (DTC) encoding technique [43]. Notably, the *metatheory à la carte* work [7] 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 [31, 33, 47, 22] 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 [30] 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 available [10]. The latest version of Rocqet can be found at the following link:

https://github.com/rocqetry/rocqet

References

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

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

[10] 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

[11] Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. *Certified Compilers à la Carte (Extended Version)*. Technical Report CS-2025-02. School of Computer Science, University of Waterloo.

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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

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

[24] 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

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

[26] 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

[27] 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

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

[29] 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.

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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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.

[38] 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

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

[40] 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

[41] 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

[42] 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

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

[44] 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

[45] 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

[46] 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.

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

Received 2024-11-15; accepted 2025-03-06