PERSIMMON: Nested Family Polymorphism with Extensible Variant Types

ANASTASIYA KRAVCHUK-KIRILYUK, Harvard University, USA
GARY FENG, University of Waterloo, Canada
JONAS ISKANDER, Harvard University, USA
YIZHOU ZHANG, University of Waterloo, Canada
NADA AMIN, Harvard University, USA

Many obstacles stand in the way of modular, extensible code. Some language constructs, such as pattern matching, are not easily extensible. Inherited code may not be type safe in the presence of extended types. The burden of setting up design patterns can discourage users, and parameter clutter can make the code less readable. Given these challenges, it is no wonder that extensibility often gives way to code duplication. We present our solution: PERSIMMON, a functional system with nested family polymorphism, extensible variant types, and extensible pattern matching. Most constructs in our language are built-in "extensibility hooks," cutting down on the parameter clutter and user burden associated with extensible code. PERSIMMON preserves the relationships between nested families upon inheritance, enabling extensibility at a large scale. Since nested family polymorphism can express composable extensions, PERSIMMON supports mixins via an encoding. We show how PERSIMMON can be compiled into a functional language without extensible variants with our translation to Scala. Finally, we show that our system is sound by proving the properties of progress and preservation.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Extensible languages*; *Abstract data types*; *Polymorphism*; *Inheritance*; *Modules / packages*; *Reusability*; *Software evolution*; *Semantics*; • **Theory of computation** → *Abstraction*; *Type theory*; *Pattern matching*; *Object oriented constructs*; *Functional constructs*.

Additional Key Words and Phrases: Persimmon, nested inheritance, family polymorphism, extensibility, composable extensions

ACM Reference Format:
Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. PERSIMMON: Nested Family Polymorphism with Extensible Variant Types . *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 119 (April 2024), 27 pages. https://doi.org/10.1145/3649836

1 INTRODUCTION

Writing modular, extensible code is hard. The Expression Problem epitomizes the difficulty [Wadler et al. 1998]. It challenges the programmer to reconcile two conflicting objectives: adding new constructors to data types, and adding new functions over data types. Different programming styles enjoy different advantages in this task. The object-oriented (OO) programming style makes it easy to add new constructors (as classes), but adding new functions requires sweeping changes to all constructors. By contrast, the functional programming style makes it easy to add new functions,

Authors' addresses: Anastasiya Kravchuk-Kirilyuk, Harvard University, Cambridge, USA, akravchukkirilyuk@g.harvard.edu; Gary Feng, University of Waterloo, Waterloo, Canada, gary.feng@uwaterloo.ca; Jonas Iskander, Harvard University, Cambridge, USA, jonasiskander@college.harvard.edu; Yizhou Zhang, University of Waterloo, Waterloo, Canada, yizhou@uwaterloo.ca; Nada Amin, Harvard University, Cambridge, USA, namin@seas.harvard.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/4-ART119
https://doi.org/10.1145/3649836

but adding new constructors to types requires sweeping changes to all functions. When changes to existing code are infeasible, the programmer must duplicate code; either way, modularity is lost.

This conflict has driven language designers to devise new programming abstractions for code reuse and polymorphism. *Family polymorphism* is one such idea that originates in object-oriented programming [Ernst 2001; Igarashi et al. 2005; Zhang and Myers 2017]. It allows extension to happen at the level of *families* of related types. Code is *polymorphic* to the family it is nested within, so code defined in a base family can be safely reused by derived families. Virtual classes [Madsen et al. 1993], virtual types [Thorup 1997], and nested inheritance [Nystrom et al. 2004] are all forms of family polymorphism. Importantly, nested family polymorphism supports the inheritance and further binding of nested families, enabling large-scale extensibility and code reuse [Ernst 2003]. Complex software systems such as extensible compilers can be expressed with nested family polymorphism, with the source and target languages as extensible nested components (Figure 1). Mixins can be encoded via nested family polymorphism, supporting the *composition* of large nested systems.

The power of nested family polymorphism is yet to be fully realized in the design of functional languages, however. Although associated types [Chakravarty et al. 2005] in Haskell are inspired [Peyton Jones 2009] by virtual types, they do not provide the same level of extensibility that nested family polymorphism can offer in the OO setting. A recent system, FPOP, introduces top-level family polymorphism into the functional setting, but does not support the inheritance and extension of nested families [Jin et al. 2023]. Compositional programming [Zhang et al. 2021] does support nested inheritance, but limits type declarations to the top-level, lacking the expressive power types can achieve as mutually recursive family members.

Other systems may support nested inheritance, but not extensible variant types [Ernst et al. 2006; Igarashi et al. 2005; Nystrom et al. 2004; Zhang and Myers 2017]. Variant types (also known as algebraic data types) are central to functional programming; they are the primary way to allow variations in the data representation of a type. The elimination form of variants is pattern matching, which often results in more concise code than achievable in the OO style through the Visitor pattern [Gamma et al. 1994]. We consider it critical that a deeper integration of nested family polymorphism into functional languages should support extensible variant types. A language design should allow a derived family to add new constructors to variant types declared in its base family, and support the extension of pattern match expressions with new cases.

One difficulty in supporting extensible variant types is the tension between extensibility and type safety. Type safe code must check *exhaustivity* of pattern matching – there must exist a match case for each variant. In the presence of extensions, exhaustivity checking involves both the inherited definitions from the base family *and* their extensions in the derived family. Nested inheritance makes exhaustivity checking even harder, as we must allow for the possibility that pattern match expressions inherited from families *nested within* the base family may not be exhaustive in the derived family.

We reconcile this tension by introducing `cases` constructs, nominal pattern matching expressions that are polymorphic to their enclosing families. Since `cases` definitions are family members, they can be extended directly in the derived family, mirroring our extensible variant types. Our approach also simplifies exhaustivity checking even in the presence of nested inheritance: `cases` are checked to be exhaustive as part of a well-formedness check for family members. The type of a `cases` construct reflects all handled constructors of the scrutinee type, eliminating the need to access definitions from the base family.

1.1 Design Considerations

Various solutions exist for extensible, functional programming. Ours is not just another solution; it is concerned with additional design goals that harness the expressive power of extensibility

in systems with nested components. Specifically, we aim for a language design that meets the following goals *in addition* to the classic goal of **type safety**.

- ➢ **Extensibility at scale**. It is a pity that many solutions focus only on extensibility of *small* code units like classes, traits, and functions. Module and namespace mechanisms carry a convenient organizational advantage in large software developments. The ability to coevolve components of *arbitrarily large* code units (namely, modules that nest modules) will enable the programmer to create *extensible software frameworks* with ease.
- ➢ **Scalable extensibility**. In addition to extensibility *at scale* (i.e., supporting large code bases and nesting), a solution should also be *scalable*. Scalable solutions allow engineers to rapidly develop and extend code bases as software evolves. Little bookkeeping should be required of the programmer before an extension is introduced, and the effort to implement an extension should be proportional to the delta in program functionality. One common way to address extensibility is by explicitly parameterizing a unit of code with extensibility hooks. However, solutions of this flavor tend to require code to be written differently in the absence and presence of future extensions. They lead to *parameter clutter* for large code units with interdependent components, reducing scalability of the extensibility mechanism.
- ➢ **Mutual recursion**. The solution should support unrestricted, mutually recursive references between constructs in different components of the program. Complex systems with nested components – such as extensible compilers – rely on this feature.
- ➢ **Composable extensions**. In addition to being possible, extensions should be composable. The language should support composing extensions (and even families of extensions).
- ➢ **Idiomatic functional style**. The programming experience should not feel foreign to the working functional programmer. It should also be friendly to the novice programmer unaware of extensibility concerns.

1.2 Contributions

We make the following contributions in this work:

- We present a type-safe language design that supports nested family polymorphism and extensible variant types. The design is based on a simple functional core and is applicable to other functional languages with declared types.
- We showcase the expressive power of our design and its applicability to real programming challenges, such as extensible compilers. Our examples show that our language design meets our design goals.
- We pin down key aspects of the language design using a core calculus, PERSIMMON, providing a basis for integration of our family polymorphic mechanism into statically typed functional languages. We prove the soundness of the type system.
- We show how the powerful extensibility mechanism can be compiled into a functional language without extensible variants via our prototype compiler from PERSIMMON to Scala.

2 MOTIVATION

We begin with an example that illustrates how powerful (and practical!) the combination of nested family polymorphism and extensible variant types can be. Consider a compiler that can (1) transform simply-typed lambda calculus (STLC) terms into continuation passing style (CPS) and (2) transform CPS-converted terms using closure conversion. The target languages for CPS and closure conversion share some parts, so we would achieve better code reuse and modularity if both target languages extend some shared, intermediate language. Figure 1 shows the components of a base compiler, `BaseComp`: the source language (STLC), the shared language IL, the target language of CPS ($\mathsf{IL_K}$, which extends IL), and the target language of closure conversion ($\mathsf{IL_C}$, which extends IL).

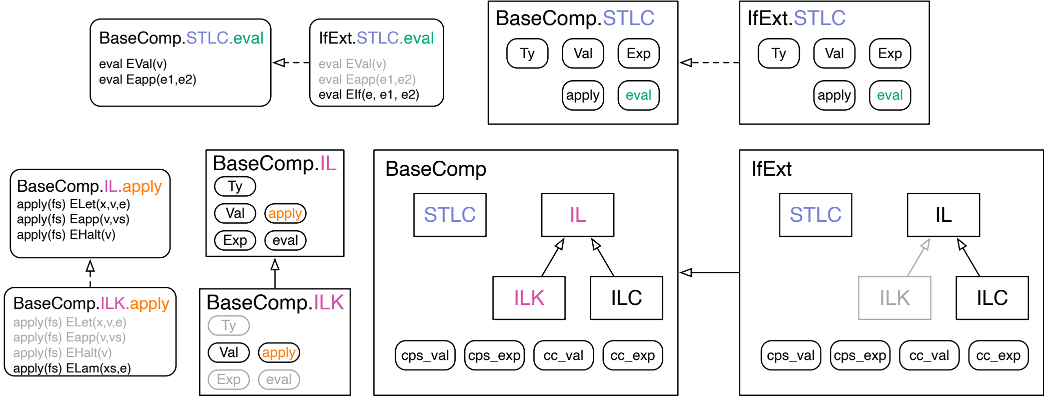

Figure 1. Motivating example: extensible compilers with nested inheritance.

Already, we recognize the need for extensible variant types. In the functional approach, the types, values, and expressions in the target languages IL_K and IL_C are represented as algebraic data types (ADTs). Since both IL_K and IL_C extend the intermediate language IL, the constructs in IL_K and IL_C are *extensions* of constructs in IL. Without extensible variant types, our capacity for code reuse is limited. For example, functions that operate on values `Val` in IL could not adapt to operate on extended values `Val` in IL_K or IL_C.

In our solution, we implement extensible variant types via family polymorphism, ensuring that constructs defined within each family are polymorphic to the family. For example, the ADT defining intermediate expressions `Exp` in IL may rely on the definition of values `Val`. However, when extending `Val` in IL_K, we need not redefine `Exp` – the inherited definition refers implicitly to the extended type `Val` via a relative path. Family polymorphism thus allows us to seamlessly reuse inherited code in a type safe way.

So far, we have only considered a single compiler and its language components. What if the source language STLC was extended with if-expressions? Without nested inheritance of compiler components, we may need to build a new compiler for each extension of STLC, making only minor changes between versions. We would much rather have an extensible compiler instead. Our solution makes this possible via *nested family polymorphism*: entire nested families can be extended, while preserving the structural and hierarchical relationships between them. We show the benefit of our solution in Figure 1. Consider two compilers: the `BaseComp` compiler for base STLC, and the `IfExt` compiler for STLC with if-expressions. Instead of being fully separate, the compiler `IfExt` extends `BaseComp`, and its component languages STLC, IL, and ILC extend their counterparts in `BaseComp`. Nested family polymorphism thus enables extensibility at the scale of large code units. We further discuss this example as a case study in Section 3.2.

We can take extensible compilers even further to showcase the importance of composable extensions. With composable extensions, we can create compilers for versions of STLC with arbitrary combinations of features (for example STLC with if-expressions, let-expressions, and references). This can be achieved with a mixin encoding, further detailed in Section 3.3.

3 NESTED FAMILY POLYMORPHISM, FUNCTIONALLY

In this section, we present the key features of Persimmon via case studies. We highlight extensible variant types, extensible pattern matching, nested families, and the mixin capabilities of Persimmon. The case studies also serve as a gentle introduction to our language. Here, we hope to develop an

```
1  Family STLCBase {
2    type Ty = TUnit | TNat | TArr(t1: Ty, t2: Ty)
3    type Val = Unit | Var(x: Str) | Lam(x: Str, e: Exp)
4    type Exp = EVal(v: Val) | EApp(e1: Exp, e2: Exp)
5    def eval : Exp -> Option Val =
6      case EVal(v) = Some v;
7      case EApp(e1, e2) =
8        (eval e1) >>= (λ v -> apply(e2) v)
9    def apply(e2: Exp) : Val -> Option Val =
10     case Lam(x, e) = eval (subst x e2 e);
11     case _ = None
12   ... (* subst, tc, print, etc. *)
13 }
```

```
14 Family STLCIf extends STLCBase {
15   type Ty += TBool
16   type Val += True | False
17   type Exp += EIf(e: Exp, e1: Exp, e2: Exp)
18   def eval : Exp -> Option Val +=
19     case EIf(e, e1, e2) =
20         (eval e) >>= (λ v -> branch(e1, e2) v)
21   def branch(e1: Exp, e2: Exp) : Val -> Option Val =
22     case True = eval e1;
23     case False = eval e2;
24     case _ = None
25   ... (* extensions to subst, tc, print, etc. *)
26 }
```

Figure 2. A base lambda calculus (left) and an extension (right) in extended Persimmon syntax.

intuition behind the different features of our language and how they are expressed in the type system, before introducing our formal calculus in Section 4.

3.1 Extensible Variant Types and Extensible Pattern Matching

First, we focus on the essentials of our extensibility solution: extensible variant types and pattern matching. Extensible pattern matching in Persimmon sets our solution apart from other family polymorphic systems with nesting such as Familia, which does not support extensible pattern matching [Zhang and Myers 2017]. We introduce these features in Persimmon with the classic example of a base lambda calculus (STLC) and an extension to STLC, shown in Figure 2. For convenience, we use an extended Persimmon syntax in this example.[1] Family `STLCBase` contains the base calculus with natural numbers and unit. Within the family, we declare algebraic data types (ADTs) with the keyword `type`. The ADTs `Ty`, `Val`, and `Exp` represent types, values, and expressions in the base calculus. Each ADT is defined as a set of constructors, where each constructor may have input fields specified in parentheses. For example, the constructor `Var` of type `Val` has a single field `x` of type `Str`, while the constructor `Unit` has no fields. Each function declared within the family has a name, an arrow type, and a definition. If the function involves pattern matching, such as the function `eval`, we specify each match case using the keyword `case`.[2] We support wildcard pattern match cases (marked with _) that match any constructors of the given ADT in the current family (and do not apply in a blanket fashion to extensions of that ADT). Note that the base code looks quite ordinary. This is one advantage of Persimmon: no prior setup is required in base families to enjoy extensibility in the derived families. Our code follows the functional programming style familiar to the user.

Family `STLCIf` in Figure 2 highlights the elegance of extensible variant types and pattern matching in Persimmon. `STLCIf` is an extension that adds booleans and if-expressions to the base calculus using our extensibility marker `+=`. For example, the type `Val` is extended with constructors `True` and `False`, and a new case for the if-expression is added to function `eval`. Persimmon ensures exhaustivity of pattern matching *at definition*: the new case for `eval` must be specified, otherwise the pattern match in the derived family will not be well-formed. We can also add new types and functionality in derived families, such as the function `branch` in `STLCIf`.

[1]We add a base type `Str` for strings, add option types, and omit type annotations on constructor variables within cases (these annotations could be inferred).

[2]We support the in-line case syntax for user convenience, while the underlying representation involves our extensible `cases` constructs, detailed in Section 4.

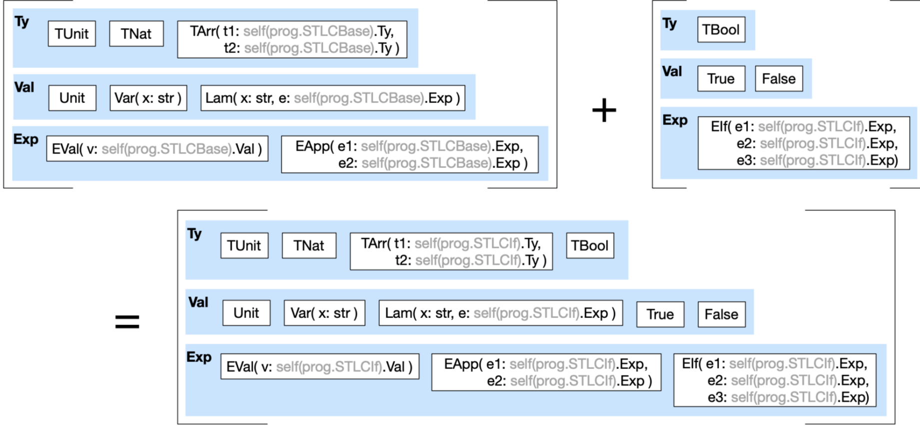

Figure 3. Example of linkage concatenation for types, combining both inherited and extended definitions.

Note that Persimmon code is highly modular: the base family only contains the base code, while the derived family only contains the extension code. Base code does not include any scaffolding for future extensions, and is not duplicated in the derived family. This *parsimonious* approach to extensible, user-written code is one inspiration for the name of our language, Persimmon.

Our minimalist approach to extensibility relies on *relative path types*. Relative path types ensure that all code is polymorphic to the enclosing family. Each type in Figure 2 has an implicit path prefix, which is inferred as the path to the immediate enclosing family. When code is inherited, any path prefix referring to the base family is replaced by the path prefix referring to the derived family. For example, consider type `Ty` which is inherited by family `STLCIf` and extended with an extra constructor. Figure 3 shows what happens under the hood: the type definition for `Ty` from family `STLCBase` (top left) is *concatenated* with the extension for `Ty` (top right), resulting in the full definition (bottom) for `Ty` in family `STLCIf`. Any self-referencing paths in the base code that refer to the parent family (for example, `self(prog.STLCBase)` in constructor `TArr`) are substituted with paths to the derived family, `self(prog.STLCIf)`. We substitute path prefixes in all inherited code, with the help of map-like data structures called *linkages* (detailed further in 4.4).

Our approach has multiple advantages. Due to relative path types and path substitution, code reuse is type-safe in Persimmon. Pattern matching is ensured to be exhaustive at definition. Persimmon code is modular and readable due to a minimal code overlap between the base and derived families. Most constructs in Persimmon are built-in extensibility hooks, eliminating parameter clutter. Finally, Persimmon reduces user effort associated with the setup of extensible frameworks, as compared to design patterns.

3.2 Nested Families and Inheritance

Here, we explore the powerful interaction between nested families and inheritance in Persimmon. We show how we use nested families to implement the extensible compilers example. We include the partial Persimmon code for this example in Figure 4, and an inheritance diagram of all nested components in Figure 1.[3]

Persimmon supports arbitrary nesting of families and preserves the nested structure upon inheritance. In Figure 4, family `BaseComp` represents the base compiler. The nested families within `BaseComp` represent the language components of the compiler: (1) the source language `STLC`, (2) the base intermediate language `IL`, (3) the target language of CPS, `ILK`, and (4) the target language of

[3]The partial implementation assumes built-in option types, `let` expressions, and pairs for convenience.

```
1  Family BaseComp {
2    Family STLC extends STLCBase {}

3    (* base intermediate language *)
4    Family IL {
5      type Ty = TUnit | TCont(ts: List Ty)
6      type Val = Unit | Var(x: Str)
7      type Exp = ELet(x: Str, v: Val, e: Exp) |
8           EApp(v: Val, vs: List Val) | EHalt(v: Val)
9      type Fun = MkFun(n: Str, xs: List Str, e: Exp)

10     def eval(fs: List Fun): Exp -> Option Val =
11       case ELet(x, v, e) = eval(fs) (subst x v e)
12       case EApp(v, vs) = apply(fs, vs) v
13       case EHalt(v) = Some v

14     def apply(fs: List Fun, vs: List Val): Val -> Option Val =
15       case _ = None
16     ... (* subst, tc, print, etc. *)
17   }

18   (* target language of CPS *)
19   Family ILK extends .IL {
20     type Val += Lam(xs: List Str, e: Exp)

21     def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
22       case Lam(xs,e) = eval(fs) (subst xs vs e)
23     ... (* subst, tc, print, etc. *)
24   }

25   (* target language of closure conversion *)
26   Family ILC extends .IL {
27     type Ty += TVar(α: Str) | TExist(α: Str, t: Ty)
28     type Val += Pack(t: Ty, v: Val) | Name(n: Str)
29     type Exp += EUnpack(α: Str, x: Str, v: Val, e: Exp)

30     def eval(fs: List Fun): Exp -> Option Val +=
31       case EUnpack(α,x,v,e) = unpack(fs,α,x,e) v

32     def unpack(fs: List Fun, α: Str, x: Str, e: Exp):
33       Val -> Option Val =
34         case Pack(t, v) = eval(fs) (subst x v (subst α t e))
35         case _ = None
36
37     def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
38       case Name(n) =
39         let (xs, e) = lookup(fs, n) in eval(fs) (subst xs vs e)
40       case Pack(t, v) = None
41     ... (* subst, tc, print, etc. *)
42   }

43   (* CPS translation *)
44   def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp =
45     ... (* cps_val cases *)
46   def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp =
47     ... (* cps_exp cases *)
48   (* closure conversion *)
49   def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) =
50     ... (* cc_val cases *)
51   def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) =
52     ... (* cc_exp cases *)
53 } (* end of BaseComp family *)
54
55 (* Compiler extension: add if-then-else to STLC and ILs *)
56 Family IfExt extends BaseComp {
57   Family STLC extends STLCIf {}

58   Family IL {
59     type Ty += TBool
60     type Val += Bool(b: B)
61     type Exp += EIf(v: Val, e1: Exp, e2: Exp)

62     def eval(fs: List Fun): Exp -> Option Val +=
63       ... (* new EIf case *)

64     def apply(fs: List Fun, vs: List Val):
65       Val -> Option Val += ... (* new Bool case *)
66     ... (* subst, tc, print, etc. *)
67   }

68   Family ILC extends .IL {
69     def unpack(fs: List Fun, α: Str, x: Str, e: Exp):
70       Val -> Option Val +=
71         case Bool(b) = None
72     ... (* subst, tc, print, etc. *)
73   }

74   def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp +=
75     ... (* new cases *)
76   def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp +=
77     ... (* new cases *)
78   def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) +=
79     ... (* new cases *)
80   def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) +=
81     ... (* new cases *)
82 }
```

Figure 4. A base STLC compiler and an extension in extended PERSIMMON syntax.

closure conversion, ILC. Both target languages ILK and ILC extend the intermediate language IL. ILK adds nested, open lambdas, while ILC adds existentials for abstracting closure environments.

The CPS translation from STLC to ILK is performed via functions cps_val and cps_exp on lines 44–47. Since the types of these functions are family polymorphic, we can safely reuse them in any extension to BaseComp that further binds families STLC or ILK (as long as any new pattern match cases are specified in the extension). We also get the guarantee that types from incompatible families will not be mixed – both STLC and ILK must belong to the same enclosing compiler family.

On line 56, family IfExt represents the compiler for STLC extended with if-expressions. All unchanged constructs, including those inside nested families, are inherited. Only the new constructs are added in the extension. For example, the nested family IfExt.IL further binds BaseComp.IL and adds boolean types, if-expressions, and the new match cases for eval and apply (omitted from figure). Family IfExt.ILC is extended in turn, since it defines a pattern match on the extended type IfExt.IL.Val. Nested family BaseComp.ILK is inherited as-is to become IfExt.ILK. Finally,

```
1 Family IfExt {
2   Family Base extends STLCBase {}
3   Family Derived extends Base {
4     (* ... contents of mixin IfExt, paths substituted *) }}
5 Family ArithExt {
6   Family Base extends STLCBase {}
7   Family Derived extends Base {
8     (* ... contents of mixin ArithExt, paths substituted *) }}
9 Family IfExtBuild extends IfExt {
10   Family Base extends STLCBase {}
11 }
12 Family ArithExtBuild extends ArithExt {
13   Family Base extends IfExtBuild.Derived {}
14 }
15 (* This family contains both if-expressions and arithmetic. *)
16 Family STLCIfArith extends ArithExtBuild.Derived {}
```

Figure 5. An encoding of mixins in Persimmon.

the translation functions are extended with new pattern match cases (omitted from figure). Figure 1 shows a detailed breakdown of all constructs that are inherited unchanged (in light grey) and constructs that are extended (in black).

Persimmon combines the benefits of nesting with the benefits of family polymorphism. We can define and inherit nested components, while preserving the structural and hierarchical relationships between them in the derived family. We have family polymorphic guarantees: inherited code is type safe for use in a derived family, and interactions between members of incompatible families are prohibited (for example, we could not call the function `BaseComp.IL.eval` on an instance of type `IfExt.IL.Exp` due to a mismatch in path prefixes to type `Exp`). The extensible compilers example in Figure 4 shows how we can enjoy all these benefits together in Persimmon.

3.3 Support for Mixins

In addition to linear extensions, Persimmon also supports composable extensions – *mixins* – with a simple encoding shown in Figure 5. Mixins allow us to compose functionality from parallel extensions without creating new inheritance relationships between the extensions. By supporting mixins via encoding, we avoid needlessly complicating the type system and duplicating language features. Nested family polymorphism in Persimmon is powerful enough to encode mixins, obviating the need for a native mixin construct. Consider the following example which uses the mixin syntax we would like to encode:

```
1 Mixin IfExt extends STLCBase { (* ... contents of mixin IfExt *) }
2 Mixin ArithExt extends STLCBase { (* ... contents of mixin ArithExt *) }
3 Family STLCIfArith extends STLCBase with IfExt, ArithExt {}
```

Suppose we want to extend `STLCBase` with multiple parallel features, such as if-expressions `IfExt` and arithmetic `ArithExt` (shown above). Ideally, we would define an extension for each feature only once (lines 1 and 2 above), and then *compose* those extensions (line 3) to yield arbitrary combinations of features. We use this example as a roadmap for our Persimmon encoding.

We encode mixins in Persimmon as shown in Figure 5 by combining linear extension with a flexible base for extension. Each family representing a mixin, such as `Family IfExt`, contains two nested families: a `Base` family, and a `Derived` family. The `Base` family can be further bound, which allows extensions to build on any version of `STLC`. The `Derived` family extends `Base` and contains the code relevant to the extension. This nested family structure ensures that the dependencies between extensions are flexible, and that the extensions are composable.

Lines 9–16 in Figure 5 show how we encode composition of extensions in Persimmon. This code is a direct translation from our roadmap example. We further bind the `Base` family for each subsequent extension, “stacking” the extensions on top of each other. `IfExtBuild.Base` extends base STLC, and `ArithExtBuild.Base` extends STLC with if-expressions. Finally, we make explicit that `STLCIfArith` extends `ArithExtBuild.Derived`, including both features.

While we do encode mixin composition via linear extension, our linearization follows the order of overwriting that is generally imposed by mixins. No inheritance relationship is created between the two parallel extensions, `IfExt` and `ArithExt`; they can be freely composed with other extensions. Our mixin encoding highlights the *parsimony of features* in our language: nested families in PERSIMMON are powerful enough to encode mixins, eliminating the need for a separate mixin construct. Finally, the encoding is completely automated – the programmer can enjoy the convenient mixin syntax as shown in the roadmap example.

4 THE PERSIMMON CALCULUS

In this section, we give a comprehensive overview of the PERSIMMON calculus as follows.

- First, we discuss the calculus syntax, highlighting the constructs that facilitate nested family polymorphism in our language: relative path types, nested families, and our extensible `cases` constructs (Section 4.1).
- Next, we present our type system (Section 4.2), which is based on static linkages: the map-like data structures that store type-level information about each family (further detailed in Section 4.4). Static linkages are a generalization of global class tables found in many type systems. Our type system directly references the contents of computed linkages, and thus is fairly straightforward. Our operational semantics is also streamlined by linkages; however, a different type of linkage is used that also stores definitions (Section 4.3).
- Finally, we discuss linkage operations and the benefits of linkages in detail (Section 4.4). Each family in our program (and the program itself) has a corresponding linkage. Linkages – as opposed to other data structures, such as an abstract syntax tree – make it easy to substitute paths inside inherited code so that it refers to the derived family. PERSIMMON supports nested inheritance, further binding of families, and extensible data types and pattern matching by nested *linkage concatenation*, a recursive operation that combines linkages for a base family and a derived family.

Underlying the linkage operations and the type system is the unifying notion of well-formedness: well-formed family definitions parse into well-formed linkages, and well-formedness of linkages is preserved by concatenation. Exhaustivity of pattern matching is a well-formedness condition, checked at program definition (Section 4.2).

4.1 Syntax

We show the full syntax of PERSIMMON in Figure 6. We follow a classic approach to family extension with relative path types [Igarashi et al. 2005]. In this section, we highlight our use of relative path types as well as the special shape of our `match` expressions, which forces the use of our extensible `cases` constructs within the match and thus enables extensible pattern matching in PERSIMMON. We also introduce the syntax of linkage structures we use for extension.

Paths, Types, and Expressions. In the context of a program, each family has a unique, fully qualified path that specifies the nesting depth of the family with respect to the program path, `prog`. The path `prog` is the prefix to all family paths. For example, family A' nested within family A is located at the path $\texttt{prog}.A.A'$. *Relative paths* reference the current family using the keyword `self` and adapt upon family extension. For example, inside the base family A the path $\texttt{self}(\texttt{prog}.A)$ refers to A, but upon inheritance into a derived family A'' the relative path is updated to refer to A''. This keeps inherited code up to date and compatible with the latest extension.

Path types are represented in PERSIMMON using syntax $a.R$, where a is the path to the family in which type name R is defined. Relative path types, such as $\texttt{self}(\texttt{prog}.A).T$, have a relative path prefix. When a relative path type is inherited, we update the relative path prefix to now refer to the

Family Name A **Relative Path** $sp ::= \texttt{prog} \mid \texttt{self}(a.A)$

Program Path $\texttt{prog}$ **Path** $a ::= sp \mid a.A$

Type $T, T' ::= \texttt{N} \mid \texttt{B} \mid a.R \mid T \to T' \mid \{(f_i : T_i)*\}$

Expression $e, g ::= n \mid b \mid x \mid a.m \mid a.c \mid g\ e \mid e.f \mid \lambda(x : T).e \mid \{(f_i = e_i)*\} \mid a.R(\{(f_i = e_i)*\})$
$\mid a.R(C\ \{(f_i = e_i)*\}) \mid \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \mid \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\}$

Value $v ::= n \mid b \mid \lambda(x : T).e \mid \{(f_i = v_i)*\} \mid a.R(\{(f_i = v_i)*\}) \mid a.R(C\ \{(f_i = v_i)*\})$

Path Context $\mathrm{K} ::= [\,] \mid sp :: \mathrm{K}$ **Linkage** $L ::= L_S \mid L_D$

Program p ::= famdef* e **Definition** def ::= famdef | typdef | adtdef | fundef | casesdef

famdef ::= **Family** A (**extends** $a.A'$)? {famdef* typdef* adtdef* fundef* casesdef*}

typdef ::= **type** $R = \{(f_i : T_i)*\}$ | **type** $R \mathrel{+}= \{(f_i : T_i = v_i)*\}$

adtdef ::= **type** $R\ (+)?\!= \overline{C_j\ \{(f_i : T_i)*\}}$

fundef ::= **val** $m : T \to T' = \lambda(x : T).e$

casesdef ::= **cases** $c\ \langle a.R \rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T)*\}\ (+)?\!= \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\}$

Figure 6. The Persimmon syntax.

derived family. A type's path prefix may also specify the exact family in which the type appears, such as prog.$A.T$. Finally, there is no raw ADT type in Persimmon – all ADTs in our system are path types with ADT definitions (for example, ADT `Exp` on line 7 in Figure 4).

Function calls ($a.m$) in Persimmon specify the path a to the family in which function m appears. Similarly, `cases` calls ($a.c$) select a `cases` definition c from the family at path a. We can view `cases` definitions as special function definitions, with a specific output type and the ability to extend the function body. Our `match` expressions have a special shape due to the separate `cases` constructs. For example, see the function `ev` and the corresponding `cases` construct `evc` on lines 5–7 in Figure 14. Inside a `match` expression, `match` e `with` $a.c$ $\{(f_{arg} = e_{arg})*\}$, the appropriate `cases` definition $a.c$ is applied to a record of arguments. The arguments represent a *match context* – any additional information needed for the pattern match, such as referenced variables. We don't generalize the left-hand side of the application to simplify the translation of our `match` expressions to other languages. By separating our `cases` definitions from their uses, we ensure that `cases` are easily extensible as family members regardless of how deeply their uses are nested.

Finally, we can create instances of path types ($a.R$) via instance expressions – $a.R(\{(f_i = e_i)*\})$ for named record types, and $a.R(C\ \{(f_i = e_i)*\})$ for ADTs, where C is a valid constructor of the type. To create a record type instance $a.R(\{(f_i = e_i)*\})$, the user must specify an input e_i for each field f_i. If a field is omitted, it must have a default value, otherwise the instance will not type-check. We require default values for all extensions of record types to ensure type safety of inherited instance expressions.

Programs and Definitions. A Persimmon program contains an arbitrary number of family definitions and a main expression. Family definitions can contain nested families, record types, ADTs, functions, and `cases` constructs. Extensions for inherited record types, ADTs, and `cases` constructs can be specified with marker +=, as opposed to the new definition marker =.[4] Extensions to record types must provide default values v_i for each field. For readability of ADT definitions, we

[4]The += syntax is a user convenience, but not a requirement. Since we do not allow overwriting of existing types, any named type that appears in the base family and in the derived family will be considered an extension in the derived family, regardless of the marker.

$$\left\{\begin{array}{lcl} \texttt{self} & = & a \\ \texttt{super} & = & a.A? \\ \texttt{NEST} & = & \{\ (A \mapsto L_S)*\ \} \\ \texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\ \texttt{DEFS} & = & \{\ (R \mapsto (f_k)*\ )*\ \} \\ \texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\ \texttt{FUNS} & = & \{\ (m \mapsto T)*\ \} \\ \texttt{CASES} & = & \{\ (c \mapsto (\langle T \rangle,\ T'))*\} \end{array}\right\}_{L_S} \qquad \left\{\begin{array}{lcl} \texttt{self} & = & a \\ \texttt{super} & = & a.A? \\ \texttt{NEST} & = & \{\ (A \mapsto L_D)*\ \} \\ \texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\ \texttt{DEFS} & = & \{\ (R \mapsto \{(f_k = v_k)*\}\ )*\ \} \\ \texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\ \texttt{FUNS} & = & \{\ (m \mapsto (T,\ e))*\ \} \\ \texttt{CASES} & = & \{\ (c \mapsto (\langle T \rangle,\ T',\ e))*\} \end{array}\right\}_{L_D}$$

Figure 7. Linkage syntax for static linkages L_S (left) and dynamic linkages L_D (right).

use an overline symbol instead of a Kleene star to represent a set of ADT constructors C_j with the corresponding input fields and their types.

Each `cases` construct c in our system is a function from a match context to a record of "case handlers" for constructors of the scrutinee type, $a.R$. Each constructor C_j is assigned a handler function, which takes as input the fields of that constructor. The output type of a `cases` construct explicitly names each handled constructor and the type of the corresponding handler function. While we use this detailed syntax for ease of type checking in our system, users can enjoy the convenient in-line syntax shown earlier.

Linkages. We differentiate between two kinds of linkages: *static linkages* L_S that store type-level information, and *dynamic linkages* L_D that store both type- and definition-level information (Figure 7). Both store the current family path (`self`), the parent family path (`super`), a map of record type definitions (`TYPES`), and a map of ADT definitions (`ADTS`). Static linkages keep track of all record type fields that have defaults. Dynamic linkages additionally store the default values for those fields. Static linkages store function and `cases` signatures, while dynamic linkages also store the bodies of those constructs.

4.2 Type System

Our type system is based on static linkages, which are a generalization of global class tables. We give a more detailed view of linkage computation and concatenation in Section 4.4. During type checking, we simply compute a static linkage L_S for any family path a on demand, and retrieve any desired types or function signatures from there. We retrieve definitions from *complete* linkages, which include all inherited, extended, and overwritten components for the family path at hand. For example, while the incomplete linkage for family path `prog.STLCIf` in Figure 2 maps the type `Ty` to the sole constructor `TBool`, the complete linkage maps `Ty` to four constructors: the inherited constructors `TUnit`, `TNat`, and `TArr`, and the extension `TBool`. We delegate the heavy-duty handling of extensibility to linkage computation, which simplifies type checking.

It is important to note that the on-demand approach to linkage computation has implications for the efficiency and modularity of type checking. Our theory may require a linkage for the same family path to be recomputed multiple times throughout the type checking process, negatively affecting performance. This is why in our implementation we cache the computed linkages for efficiency. Furthermore, the on-demand approach poses a conflict for separate type checking and compilation of *program fragments* as defined by Cardelli [1997]. A family in PERSIMMON is an example of a program fragment. Modular type checking of each such fragment would require a more sophisticated dependency analysis than the on-demand approach allows, along with a pre-computation of linkages for each family. This is not currently supported in PERSIMMON. We address how modular type checking and separate compilation could be supported in the future in Section 9.

$\boxed{\mathrm{K}; \Gamma \vdash e : T}$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(T) \qquad \mathrm{K}; (x : T, \Gamma) \vdash e : T'}{\mathrm{K}; \Gamma \vdash \lambda(x : T).e : T \to T'} \text{ (T-Lam)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad m \mapsto (T \to T') \in L_S.\text{FUNS}}{\mathrm{K}; \Gamma \vdash a.m : T \to T'} \text{ (T-FamFun)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \{(f_j : T_j)*\} \in L_S.\text{TYPES} \qquad R \mapsto (f_k)* \in L_S.\text{DEFS} \\ \forall i,\ \exists j,\ f_i = f_j \ \wedge\ \mathrm{K}; \Gamma \vdash e_i : T_j \qquad \forall j,\ f_j \in (f_i)* \ \vee\ f_j \in (f_k)*\end{array}}{\mathrm{K}; \Gamma \vdash a.R(\{(f_i = e_i)*\}) : a.R} \text{ (T-Constr)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\text{ADTS} \qquad C\ \{(f_k : T_k)*\} \in \overline{C_j\ \{(f_i : T_i)*\}} \qquad \forall k,\ \mathrm{K}; \Gamma \vdash e_k : T_k}{\mathrm{K}; \Gamma \vdash a.R(C\ \{(f_k = e_k)*\}) : a.R} \text{ (T-ADT)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad c \mapsto (\langle a'.R \rangle, \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}) \in L_S.\text{CASES}}{\mathrm{K}; \Gamma \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}} \text{ (T-Cases)}$$

$$\frac{\begin{array}{c}\mathrm{K}; \Gamma \vdash e : a'.R \qquad a' \rightsquigarrow L_S \qquad R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\text{ADTS} \\ \mathrm{K}; \Gamma \vdash a.c : \{(f_{arg} : T_{arg})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T)*\} \qquad \mathrm{K}; \Gamma \vdash \{(f_{arg} = e_{arg})*\} : \{(f_{arg} : T_{arg})*\}\end{array}}{\mathrm{K}; \Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} : T} \text{ (T-Match)}$$

$\boxed{\mathrm{K} \vdash \mathtt{WF}(a)}$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad A \in L_S.\text{NEST}}{\mathrm{K} \vdash \mathtt{WF}(a.A)} \text{ (WF-Path-Abs)} \qquad \frac{sp \in K}{\mathrm{K} \vdash \mathtt{WF}(sp)} \text{ (WF-Path-Self)}$$

Figure 8. Selected rules for type checking expressions.

Type Checking of Expressions. Figure 8 highlights the type-checking rules that best showcase the handling of extensibility in Persimmon, while the full relation can be found in the supplemental appendix. Any type-level information required for type checking is retrieved from a complete static linkage L_S for the appropriate family path a. We type check expressions with respect to a typing context Γ and a family path context K.[5] The context K keeps track of the nesting depth of the current expression within the program. Some expressions, such as functions and `cases` calls, are type checked by retrieving their type signatures directly from the linkage (rules `T-FamFun` and `T-Cases`). Since well-formed family definitions parse into well-formed linkages, and linkage concatenation preserves well-formedness, the retrieved signature reflects the true type of the expression.

An instance of a record type, $a.R(\{(f_i = e_i)*\})$, is well-typed if the linkage L_S for path a contains a definition for type R, and the inputs e_i are well-typed with respect to this definition (rule `T-Constr`). Any field f_j in the definition of R that does not have an input e_i within the instance expression must have a stored default value, or the instance will not type-check. ADT instances are checked similarly, while also ensuring that the constructor C used to create the instance is a valid constructor (rule `T-ADT`). ADTs do not take default field values, so a well-typed input e_k must be provided for every field f_k in constructor C. A pattern match expression will type-check if the type of the scrutinee e is some path type $a'.R$, which has an ADT definition in the complete static linkage for path a'. The `cases` call $a.c$ and the match context $\{(f_{arg} = e_{arg})*\}$ must be well-typed (rule `T-Match`).

Persimmon supports reflexivity of subtyping, subtyping of arrow types, depth and width subtyping of record types, and subtyping of path types (for conversion between a path type and the

[5]The syntax for K is shown in Figure 6. For singleton contexts we use the shorthand syntax $[sp]$, equivalent to $sp :: []$.

$\boxed{\mathrm{K};\Gamma \vdash \mathrm{p} : T}$

$$\frac{\mathrm{p} = \mathrm{famdef}_i{*}\ e \qquad \forall i,\ [\mathtt{prog}] \vdash \mathtt{WF}(\mathrm{famdef}_i) \qquad [\mathtt{prog}]; [] \vdash e : T}{[]; [] \vdash \mathrm{p} : T} \quad \text{(T-PROG)}$$

$\boxed{\mathrm{K} \vdash \mathtt{WF}(\mathrm{def})}$

$$\frac{\begin{array}{c}\mathrm{famdef} = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\mathrm{famdef}_n{*}\ \mathrm{typdef}_q{*}\ \mathrm{adtdef}_u{*}\ \mathrm{fundef}_w{*}\ \mathrm{casesdef}_z{*}\} \\ \mathtt{self}(sp.A) \rightsquigarrow L_S \qquad \mathtt{self}(sp.A) \notin \mathtt{ancestors}(L_S) \qquad \neg\mathtt{nested}(a.A', \mathtt{self}(sp.A)) \\ sp :: \mathrm{K} \vdash \mathtt{WF}(a.A') \qquad \mathrm{K}' = \mathtt{self}(sp.A) :: sp :: \mathrm{K} \qquad \mathrm{K}' \vdash \mathtt{EC}(L_S) \\ \forall n,\ \mathrm{K}' \vdash \mathtt{WF}(\mathrm{famdef}_n) \qquad \forall q,\ \mathrm{K}' \vdash \mathtt{WF}(\mathrm{typdef}_q) \qquad \forall u,\ \mathrm{K}' \vdash \mathtt{WF}(\mathrm{adtdef}_u) \\ \forall w,\ \mathrm{K}' \vdash \mathtt{WF}(\mathrm{fundef}_w) \qquad \forall z,\ \mathrm{K}' \vdash \mathtt{WF}(\mathrm{casesdef}_z)\end{array}}{sp :: \mathrm{K} \vdash \mathtt{WF}(\mathrm{famdef})} \quad \text{(WF-FAMDEF)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\})}{\mathrm{K} \vdash \mathtt{WF}(\textbf{type}\ R\ =\ \{(f_i : T_i){*}\})} \quad \text{(WF-TYPDEF)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\}) \qquad \forall i,\ \mathrm{K}; [] \vdash v_i : T_i}{\mathrm{K} \vdash \mathtt{WF}(\textbf{type}\ R\ \mathrel{+}=\ \{(f_i : T_i = v_i){*}\})} \quad \text{(WF-TYPDEF-EXT)}$$

$$\frac{\forall j,\ \mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\})}{\mathrm{K} \vdash \mathtt{WF}(\textbf{type}\ R\ \ (+)?{=}\ \ \overline{C_j\ \{(f_i : T_i){*}\}})} \quad \text{(WF-ADTDEF)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(T \to T') \qquad \mathrm{K}; [] \vdash \lambda(x : T).e : T \to T'}{\mathrm{K} \vdash \mathtt{WF}(\textbf{val}\ m\ :\ T \to T'\ =\ \lambda(x : T).e)} \quad \text{(WF-FUNDEF)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \quad a \rightsquigarrow L_S \quad R \mapsto \overline{C_i\ T_i} \in L_S.\mathtt{ADTS} \quad \forall j,\ \exists i,\ (C_j = C_i \wedge T_j = T_i) \quad \mathrm{K} \vdash \mathtt{WF}(T \to \{(C_j : T_j \to T'){*}\}) \\ \mathrm{K}; [] \vdash \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j){*}\} : T \to \{(C_j : T_j \to T'){*}\}\end{array}}{\mathrm{K} \vdash \mathtt{WF}(\textbf{cases}\ c\ \langle a.R\rangle\ :\ T \to \{(C_j : T_j \to T'){*}\}\ \ (+)?{=}\ \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j){*}\})} \quad \text{(WF-CASESDEF)}$$

$\boxed{\mathrm{K} \vdash \mathtt{EC}(L_S)}$

$$\frac{\begin{array}{c}\forall\ (c \mapsto (\langle a.R\rangle, T \to \{(C_j : T_j \to T'){*}\})) \in L_S.\mathtt{CASES},\ \mathrm{K} \vdash \mathtt{WF}(a) \wedge a \rightsquigarrow L'_S \wedge R \mapsto \overline{C_j\ T_j} \in L'_S.\mathtt{ADTS} \\ \forall\ A \in L_S.\mathtt{NEST},\ \ sp = L_S.\mathtt{self} \wedge \mathrm{K}' = \mathtt{self}(sp.A) :: \mathrm{K} \wedge \mathtt{self}(sp.A) \rightsquigarrow L''_S \wedge \mathrm{K}' \vdash \mathtt{EC}(L''_S)\end{array}}{\mathrm{K} \vdash \mathtt{EC}(L_S)} \quad \text{(EC-NEST)}$$

Figure 9. Type checking, well-formedness (WF), and exhaustivity checking (EC) of programs.

corresponding record type). The full rules are included in the appendix. An extended type from the derived family is not a subtype of the corresponding type from the base family, due to undesired interactions between relative path types and inheritance [Igarashi et al. 2005].

Typing and Well-Formedness of Programs. A program p is well-typed if every family definition within p is well-formed, and the main expression e is well-typed (rule T-Prog in Figure 9). At this topmost level of nesting, the linkage context K contains one path, prog, which is the path to p. To prevent circular inheritance, a well-formed family definition (WF-FamDef) cannot have its own family path as an ancestor, and cannot inherit from a nested family. All nested definitions within the family must also be well-formed. Since we use the linkage context to keep track of the nesting level, all nested definitions must be checked with respect to a linkage context K′, which extends K with the path to the current family. We maintain the convention that the head path in the linkage context points to the immediate wrapper family of the checked definition.

Importantly, we consider exhaustivity of pattern matching a well-formedness condition. We trigger an *exhaustivity check* from WF-FamDef to recursively check that pattern matching is exhaustive

in the current family definition, and any nested family definitions (rule `EC-Nest` in Figure 9). This check operates on complete static linkages, as opposed to program definitions, since we need to check the inherited `cases` constructs as well. Consider the following example:

```
1 Family A1 {
2     type T = C1 | C2
3     val f: T -> N = λ(t: T). match t with c {}
4     cases c <T>: {} -> {C1: {} -> N, C2: {} -> N} =
5       λ(_: {}). {C1 = λ(_:{}). 1, C2 = λ(_:{}).2}
6 }
```

```
7 Family A2 extends A1 {
8     type T += C3
9
10    // no match for C3!
11    val g: T -> N = λ(t: T). (f t)
12 }
```

Since family `A2` inherits the function `f` on line 3, the inferred relative path to `T` in the input type of `f` is updated via path substitution, giving `f` the input type `self(prog.A2).T` inside `A2`. The input `t` on line 11 has the same type, and the application in the body of `g` type-checks. However, the `cases` construct called by `f` has not been extended! To ensure exhaustivity of all `cases` constructs in a derived family, we perform an exhaustivity check on all inherited and newly defined `cases` constructs during well-formedness checking.

Since exhaustivity is checked separately, the rule for well-formedness of `cases` definitions (`WF-CasesDef` in Figure 9) only requires that the constructors C_j handled by the `cases` definition appear in the definition of scrutinee type, $a.R$, with the expected input types T_j. All other definitions (such as record types, ADTs, and functions) are well-formed if the types within these definitions are well-formed, and the expressions are well-typed. We also require that all types have unique names, `cases` and functions have unique headers, and that there are no duplicate constructor names in an ADT or duplicate fields in a record type. These repetitive checks are omitted in Figure 9.

4.3 Operational Semantics

Like our type system, our operational semantics delegates the heavy lifting to linkages. In operational semantics we use dynamic linkages L_D that contain both type-level and definition-level information. Our full reduction and substitution relations are included in the appendix. Most rules follow the convention of reducing subexpressions from left to right. Function calls $a.m$ and `cases` calls $a.c$ reduce directly to their definitions retrieved from the dynamic linkage L_D for family path a. The special shape of our `match` expressions means that we must perform the application of the `cases` call to the match context before we can project the required case handler.

4.4 Linkage Operations

Next, we discuss how *linkages* support extensibility in Persimmon. A linkage L is, essentially, a map of maps containing the information about a single family path. The static linkages L_S are used in static semantics, while the dynamic linkages L_D are used in dynamic semantics (see Figure 7 for a refresher on linkage syntax). We choose linkages for program representation as opposed to other options, such as an AST, for multiple reasons. Due to our use of relative path types, we must be able to easily perform *path substitution* when code is inherited (including constructs within nested families). Linkages are well-suited for this. *Linkage concatenation* – combining linkages from the base family and the derived family – is a natural fit for the modular extensibility of ADTs and pattern matching in Persimmon. Finally, our algorithmic linkage mechanism provides an easy way to look up names and signatures within any nested linkage, which helps support unrestricted, mutually recursive references between family members in Persimmon.

Linkage Computation. We compute a *complete* linkage L for family path a as shown in Figure 11. A complete linkage for a family path a includes all constructs inherited, extended, and newly defined at that family path, while an *incomplete* linkage only includes the constructs defined directly at

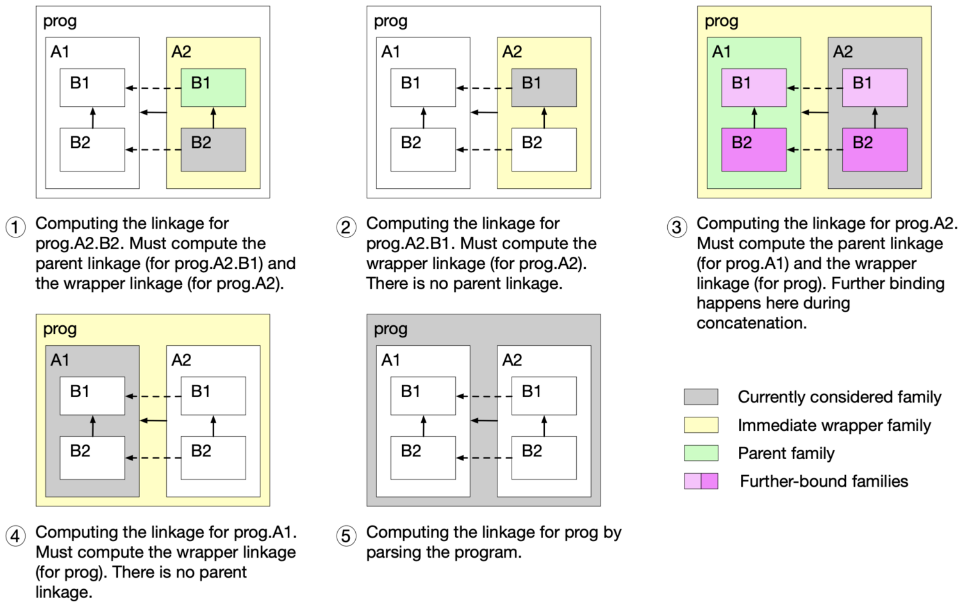

Figure 10. Linkage computation, intuitively. Each frame shows linkage computation for a single family (in grey), highlighting the recursive computation of the wrapper linkage (in yellow) and the parent linkage (in green).

that family path. Before diving into the details, let us build intuition for linkage computation with Figure 10. To keep succinct, we will use the phrase "a linkage for family A" to mean "a linkage for the family path to A". The frames in Figure 10 are diagrams of the code snippet in Figure 14, showing the nested family structure as well as the *extends* (solid) and *further binds* (dotted) links between families. In the code snippet, there are two top-level families, `A1` and `A2`. Family `A1` has two nested families, `B1` and `B2`, where `B2` extends `B1`. `A2` extends `A1` and further binds `B1` and `B2`. Each frame in Figure 10 represents linkage computation for a single family path. For example, frame (4) represents linkage computation for path `prog.A1` (highlighted in grey).

To compute a complete linkage for some family A, we must first recursively compute complete linkages for (i) the parent family of A, and (ii) the immediate wrapper family of A. The complete parent linkage will contain the constructs that must be inherited or extended. The complete wrapper linkage will let us retrieve the nested, incomplete linkage for family A – containing only the constructs that appear directly in A. Finally, we will concatenate the complete parent linkage with the incomplete linkage for A to obtain the complete linkage for A. For example, frame (1) in Figure 10 shows that to compute a complete linkage for the family path `prog.A2.B2` (highlighted in grey) we must compute the complete parent linkage (at path `prog.A2.B1`, in green) and the complete wrapper linkage (at path `prog.A2`, in yellow).

Importantly, linkages are computed *from the outside in*: linkages for wrapper families are always computed before linkages for nested families. Thus, nested family linkages within a complete linkage are themselves incomplete. Consider frame (4) in Figure 10. Linkage computation for family path `prog.A1` does not trigger linkage computation for the nested family paths, such as `prog.A1.B2`. Complete linkages for nested families must be computed on demand. The outside-in computation order also prevents linkage computation from running into an infinite loop, such as in the case of a nested family that extends its own wrapper family.

$\boxed{a \rightsquigarrow L}\ \boxed{a \rightsquigarrow_{\approx} L}$

$$\frac{\texttt{parse}_S(\texttt{p}) \;=\; L_S}{\texttt{prog} \rightsquigarrow L_S}\ \text{(L-Prog-S)} \qquad \frac{a.A \rightsquigarrow_{\approx} L}{\texttt{self}(a.A) \rightsquigarrow L}\ \text{(L-Self)} \qquad \frac{a.A \rightsquigarrow_{\approx} L \qquad L\,[a.A \,/\, \texttt{self}(a.A)] \;=\; L'}{a.A \rightsquigarrow L'}\ \text{(L-Sub)}$$

$$\frac{\texttt{parse}_D(\texttt{p}) \;=\; L_D}{\texttt{prog} \rightsquigarrow L_D}\ \text{(L-Prog-D)} \qquad \frac{a \rightsquigarrow L \qquad L'' = L.A \qquad L''.\texttt{super} \rightsquigarrow_{\approx} L' \qquad L' + L'' = L'''}{a.A \rightsquigarrow_{\approx} L'''}\ \text{(L-Nest)}$$

Figure 11. Rules for computing linkages L, parameterized by a program p.

Having established intuition for computing linkages, we now discuss the linkage computation rules in detail (see Figure 11). The rule `L-Nest` governs the linkage concatenation process (represented by +), as showcased by the frames in Figure 10. Here, we also make a distinction between *exact* linkage computation (marked $\rightsquigarrow$) and *inexact* linkage computation (marked $\rightsquigarrow_{\approx}$). *Exact* linkage computation for some path a produces a linkage that refers to the current family using exactly path a, while *inexact* linkage computation for some path $a.A$ results in a linkage that refers to the current family by path `self`$(a.A)$. This distinction is necessary because a family A can extend any family path, but linkage concatenation requires the parent linkage to refer to itself via a relative path for the purposes of path substitution.

As for the rest, rule `L-Sub` serves to translate between exact and inexact linkage computation by substituting the `self`-wrapped paths with their corresponding unwrapped versions within the computed linkage L. Rule `L-Self` computes the complete linkage L for a relative path. Finally, rules `L-Prog-S` and `L-Prog-D` compute the corresponding complete static or dynamic linkage for path `prog` by parsing the program p. Parsing also includes a process to "unfold" any wildcard cases within `cases` constructs. Each wildcard case is replaced by the explicit set of cases it implicitly covers within the given family, using the same case handler for each case. The wildcard case in a base family does not apply in a blanket fashion to any future extensions. Any derived families must provide explicit or implicit handling of all new cases for the match to be exhaustive.

We show a step-by-step example of linkage computation in Figure 12. In this example, the program consists of a family `A`, which nests families `B1` and `B2`. Family `B2` extends `B1`, as shown in the inheritance diagram in the bottom right corner. To compute the exact linkage for family path `prog.A.B2` (step 1), we must first apply the `L-Sub` rule, which will compute the corresponding inexact linkage and perform path substitution (step 2). From the `L-Sub` rule, the `L-Nest` rule is called (step 3), which will compute the wrapper linkage (for path `prog.A`, steps 4-6), compute the parent linkage (for path `prog.A.B1`, steps 7-10), and perform linkage concatenation. For each concatenation operation in the figure, we show the parent linkage on the left hand side, and the incomplete child linkage (retrieved from the wrapper linkage) on the right hand side. When there is no parent path, we use { } to denote an empty parent linkage. The label `L-Prog` generalizes over the two rules that perform program parsing, `L-Prog-S` and `L-Prog-D`. In our implementation, we cache the computed linkages for efficiency, which means that the exact linkage for path `prog.A` would be computed and cached in steps 4-6, and later retrieved in step 7.

Linkage Concatenation. Concatenation of linkages has the shape $L_1 + L_2 = L_3$, where L_1 is the complete linkage for the parent family, L_2 is the incomplete linkage for the derived family, and L_3 is the resulting complete linkage for the derived family. L_3 includes all constructs inherited from the parent family, newly defined or extended constructs in the derived family, and constructs in the derived family that overwrite inherited constructs. We recursively propagate the concatenation operation to all nested components of the linkages: sets of nested families, types, ADTs, etc. All

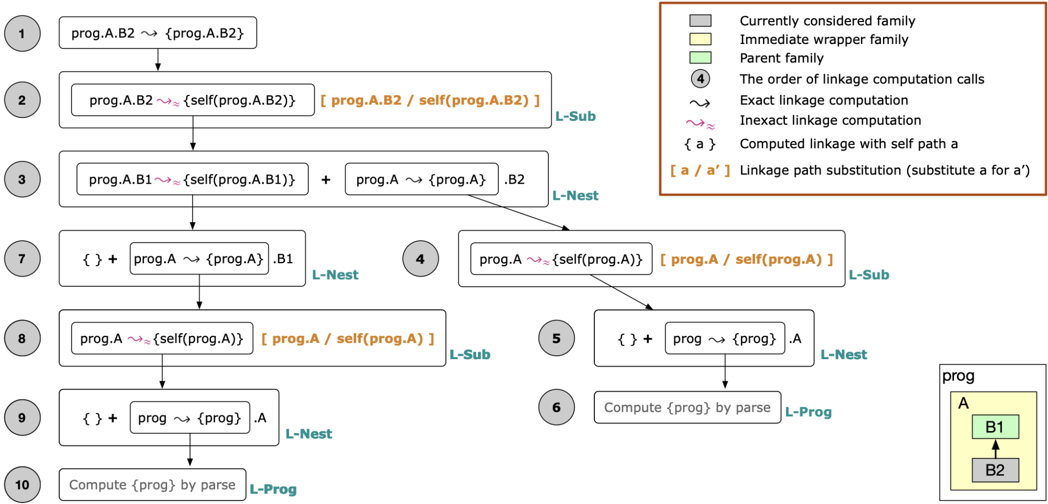

Figure 12. A step-by-step example of linkage computation. Here, we start at step 1 to compute the complete linkage for family B2 nested within family A. Family B2 extends family B1 which is also nested in A. Each subsequent step shows the linkage computation rule applied, as well as any recursive computation calls triggered by the rule.

paths in the parent linkage that refer to the parent family are updated to refer to the derived family via *path substitution*, before concatenation. This ensures that inherited code is safe for use with the extended types in the derived family. Our rules for path substitution are available in the appendix.

We show selected linkage concatenation rules in Figure 13. Linkages for nested families are recursively concatenated (rule `Cat-Nest`). Within each linkage, nested family names are mapped to their corresponding linkages (such as $A \mapsto L$). For each nested family name unique to the parent or the derived family, the mapping to its linkage is copied unchanged to the resulting linkage. These mappings are the symmetric difference, $\triangle$, of the two collections. However, when the same family A has a mapping in both linkages for the parent and derived families (represented by property $\mathcal{P}$ in the rule), it means that A is further bound in the derived family. We handle further binding in the same way as inheritance, via linkage concatenation. We concatenate the linkage L that nested family A maps to in the parent linkage with the linkage L' that A maps to in the derived linkage.

The concatenation rules for record types (`Cat-Types`) and ADTs (`Cat-ADTs`) in Figure 13 follow a similar pattern. Types cannot be overwritten in PERSIMMON since inherited code would no longer be safe for use in derived families. Thus, types and ADT definitions that have the same name in the base family and derived family will be treated as extensions in the derived family. After concatenation, the resulting sets of record types and ADTs consist of all definitions inherited from the parent, newly defined in the extension, or extended in the derived family. For record types, we define the concatenation operation + in the usual way, with no duplicate fields allowed.

The concatenation operation + as defined for records simply combines the contents of the two records, as long as there are no duplicate fields. We do not allow overwriting of existing fields to avoid unsafe interactions with inherited functionality. Concatenation for ADT definitions works similarly, with the additional constraint that constructor names cannot be duplicated.

Concatenation for function and `cases` signatures is shown by `Cat-Funs-S` and `Cat-Cases-S`. The resulting set of signatures contains the symmetric difference of the signatures. Any functions with the same signature are considered overwritten in the extension. For `cases`, we allow overwriting

$$\frac{NEST'' = \{A \mapsto L \in NEST \vartriangle NEST'\} \cup \{A \mapsto L'' : \mathcal{P}(A, L'')\} \quad \mathcal{P}(A, L'') = A \mapsto L \in NEST \wedge A \mapsto L' \in NEST' \wedge L + L' = L''}{NEST + NEST' = NEST''} \text{ (Cat-Nest)}$$

$$\frac{FUNS''_S = \{m \mapsto T \to T' \in FUNS_S \vartriangle FUNS'_S\} \cup \{m \mapsto T \to T' \in FUNS_S \bigcap FUNS'_S\}}{FUNS_S + FUNS'_S = FUNS''_S} \text{ (Cat-Funs-S)}$$

$$\frac{\begin{array}{c} TYPES'' = \{R \mapsto \{(f_k : T_k)*\} \in TYPES \vartriangle TYPES'\} \cup \{R \mapsto \{(f_k : T_k)*\} : \mathcal{P}(R, \{(f_k : T_k)*\})\} \\ \mathcal{P}(R, \{(f_k : T_k)*\}) = R \mapsto \{(f_i : T_i)*\} \in TYPES \wedge R \mapsto \{(f_j : T_j)*\} \in TYPES' \wedge \\ \{(f_i : T_i)*\} + \{(f_j : T_j)*\} = \{(f_k : T_k)*\} \end{array}}{TYPES + TYPES' = TYPES''} \text{ (Cat-Types)}$$

$$\frac{\begin{array}{c} ADTS'' = \{R \mapsto \overline{C_k \ \{(f_n : T_n)*\}} \in ADTS \vartriangle ADTS'\} \cup \{R \mapsto \overline{C_k \ \{(f_n : T_n)*\}} : \mathcal{P}(R, \overline{C_k \ \{(f_n : T_n)*\}})\} \\ \mathcal{P}(R, \overline{C_k \ \{(f_n : T_n)*\}}) = R \mapsto \overline{C_i \ \{(f_j : T_j)*\}} \in ADTS \wedge R \mapsto \overline{C'_i \ \{(f'_j : T'_j)*\}} \in ADTS' \wedge \\ \overline{C_i \ \{(f_j : T_j)*\}} + \overline{C'_i \ \{(f'_j : T'_j)*\}} = \overline{C_k \ \{(f_n : T_n)*\}} \end{array}}{ADTS + ADTS' = ADTS''} \text{ (Cat-ADTs)}$$

$$\frac{\begin{array}{c} CASES''_S = \{c \mapsto (\langle a.R \rangle, T \to T') \in CASES_S \vartriangle CASES'_S\} \cup \{c \mapsto (\langle a.R \rangle, T \to T') \in CASES_S \bigcap CASES'_S\} \cup \\ \{c \mapsto (\langle a.R \rangle, T \to T''') : \mathcal{P}(c, \langle a.R \rangle, T \to T''')\} \\ \mathcal{P}(c, \langle a.R \rangle, T \to T''') = c \mapsto (\langle a.R \rangle, T \to T') \in CASES_S \wedge c \mapsto (\langle a.R \rangle, T \to T'') \in CASES'_S \wedge T' + T'' = T''' \end{array}}{CASES_S + CASES'_S = CASES''_S} \text{ (Cat-Cases-S)}$$

Figure 13. Selected linkage concatenation rules.

when the definition in the derived family has the same name, scrutinee type, and arrow type. When a `cases` definition is extended, its resulting output type is a concatenation of the output types from the base and derived families. Our rules for extending definitions are included in the appendix. We concatenate `cases` constructs by concatenating their records of case handlers, after replacing the bound variables for the match context inside each definition with a fresh variable. We also ensure that after extension the `cases` construct does not have any duplicate case handlers.

Precedence of further binding. In our system, further binding takes precedence over inheritance, mirroring other related systems with nested inheritance, such as Jx [Nystrom et al. 2004]. Consider family `A2.B2` in Figure 14, which extends family `A2.B1`, and further binds family `A1.B2`. The function f is defined in both `A2.B1` and `A1.B2`, but the definition in `A1.B2` (further bound) takes precedence, since `A1.B2` is considered structurally more similar to `A2.B2`. Rule `Cat-Nest`, along with the linkage computation rule `L-Nest` (Figure 11), ensures this. When we compute the complete linkage for `A2.B2` via `L-Nest`, we concatenate the complete parent linkage L' (for `A2.B1`) with the incomplete child linkage L'' (for `A2.B2`). The latter is retrieved from the complete linkage L for the wrapper family, `A2`. Further binding of any nested families will be performed by rule `Cat-Nest` when L – the linkage for the wrapper family – is computed. Thus, any further bound nested components will be on the right hand side of concatenation in `Cat-Nest`, taking precedence over the inherited components on the left hand side.

5 FORMAL RESULTS

We prove that Persimmon is sound by proving progress and preservation for our calculus.

Theorem 1 (Progress). *For any main expression e in program p, if* $[]; [] \vdash p : T$ *and* $[\texttt{prog}]; [] \vdash e : T'$*, then either e is a value or there exists some* e' *such that* $[\texttt{prog}] \vdash e \longrightarrow e'$.

Theorem 2 (Preservation). *For any main expression e in program p, if* $[]; [] \vdash p : T$, $[\texttt{prog}]; [] \vdash e : T'$*, and forall* e' *such that* $[\texttt{prog}] \vdash e \longrightarrow e'$*, then* $[\texttt{prog}]; [] \vdash e' : T'$.

```
1 Family A1 {
2   Family B1 {
3     type Exp = ENat {n : N}
4     val f: N -> N = λ(n: N). n
5     val ev: Exp -> N = λ(e: Exp). match e with evc {}
6     cases evc <Exp> : {} -> {ENat: {n: N} -> N} =
7       λ(unit: {}). {ENat = λ(x: {n: N}). x.n}
8   }
9   Family B2 extends .B1 {
10    val f: N -> N = λ(n: N). n+1
11  }}
```

```
12 Family A2 extends A1 {
13   Family B1 {
14     val f: N -> N = λ(n: N). n+2
15   }
16   Family B2 extends .B1 {
17     type X = {x: B}
18     type Exp += EPlus {e1: Exp, e2: Exp}
19     cases evc <Exp> : {} -> {EPlus: {e1: Exp, e2: Exp} -> N} +=
20       λ(unit: {}).
21         {EPlus = λ(x: {e1: Exp, e2: Exp}). (ev(x.e1) + ev(x.e2))}
22   }}
```

Figure 14. Persimmon code snippet exhibiting both inheritance and further binding.

We prove these properties by induction on the typing derivation $[\texttt{prog}]; [] \vdash e : T'$. For progress, most cases follow directly from our operational semantics. For preservation, most cases are handled in a straightforward way using induction hypotheses for sub-derivations. Proof cases for rules `T-FamFun` and `T-Cases` rely on the fact that function and `cases` definitions retrieved from linkages are well-typed. We show this by proving that linkages parsed from well-typed programs are well-formed, and well-formedness is preserved by linkage concatenation. The full proofs are available in the supplemental material.

6 COMPILATION TO SCALA

We have implemented a prototype compiler for Persimmon. The compiler consists of about 2,300 lines of Scala code. Code generation works by translating Persimmon code into Scala code. Scala is already a powerful language with advanced, statically typed code reuse and extensibility mechanisms. However, it is not powerful enough to support Persimmon's nested family polymorphism and extensible variant types out of the box. Therefore, to enable code sharing, our compiler has to parameterize code with explicit extensibility hooks, use wrapper types and trampoline procedures to make dispatching explicit, and insert run-time type casts.

An excerpt of the translated code from Persimmon to Scala is available in Figure 15. A family, however nested, is compiled into a top-level Scala "trait." Each extensible variant type is compiled into a "sealed trait," with each constructor a "case class" and with case classes for inherited constructors. The translation functions enable converting from an inherited instance through a chain of inherited constructors.

The trait `Interface` generated for each family provides a layer of abstraction for each family's constructs, so that they can be safely reused in future extensions. The singleton object `Family`, which implements the interface, then provides definitions for all of the constructs. In the singleton, helper functions ending with `$Impl` are generated for the actual right-hand side implementations. These helper functions are parameterized by a list of selfs, breaking down the path of a family from the outermost `self$1` to the innermost `self$` families. As needed, these helper functions perform explicit dispatching to the relevant extending or further binding family.

7 Evaluation

In this section, we revisit our design goals and show how our solution meets these goals. We also compare Persimmon to existing extensibility solutions, namely object-oriented decomposition [Odersky and Zenger 2005a] and compositional programming [Zhang et al. 2021]. Finally, we include a case study of mixin compilers, showcasing the expressive power of Persimmon that is not easily replicated by other solutions.

```
1 import reflect.Selectable.reflectiveSelectable
2 object A2$B2 {
3   // Types
4   type X = {val x: Boolean}
5   // ADTs
6   sealed trait Exp
7   // Defined constructors
8   case class EPlus[self$$$Exp](e2: self$$$Exp, e1: self$$$Exp)
9     extends Exp
10  // Inherited constructors
11  case class A2$B1$$Exp(inherited: A2$B1.Exp) extends Exp {
12    override def toString(): String = inherited.toString()
13  }
14  case class A1$B2$$Exp(inherited: A1$B2.Exp) extends Exp {
15    override def toString(): String = inherited.toString()
16  }
17  // Path interface
18  trait Interface extends A2$B1.Interface
19    with A1$B2.Interface {
20      self$ =>
21      // Self Named types
22      type X
23      // Self ADTs
24      type Exp
25      // Functions
26      val ev: self$.Exp => Int
27      val f: Int => Int
28      // Cases
29      def evc(matched: self$.Exp): Unit => Int
30      // Translations
31      def A2$B2$$Exp(from: A2$B2.Exp): Exp
32  }
33  // Path implementation
34  object Family extends A2$B2.Interface { self$ =>
35    // Self named types instantiation
36    override type X = A2$B2.X
37    // Self ADTs instantiation
38    override type Exp = A2$B2.Exp
39    // Function implementations
40    override val ev: self$.Exp => Int = ev$Impl(A2.Family,self$)
41    def ev$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
42      self$.Exp => Int = A1$B1.Family.ev$Impl(self$1, self$)
43    override val f: Int => Int = f$Impl(A2.Family, self$)
44    def f$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
45      Int => Int = A1$B2.Family.f$Impl(self$1, self$)
46    // Cases implementations
47    def evc(matched: self$.Exp): Unit => Int =
48      evc$Impl(A2.Family,self$)(matched.asInstanceOf[A2$B2.Exp])
49    def evc$Impl(self$1: A2.Interface, self$: A2$B2.Interface)
50      (matched: A2$B2.Exp): Unit => Int =
51      (unit: Unit) => matched match {
52      case matched@A2$B2.EPlus(_, _) =>
53        val x: A2$B2.EPlus[self$.Exp] =
54          matched.asInstanceOf[A2$B2.EPlus[self$.Exp]]
55        (self$.ev.asInstanceOf[self$.Exp => Int](x.e1) +
56         self$.ev.asInstanceOf[self$.Exp => Int](x.e2))
57      case A2$B2.A2$B1$$Exp(inherited) =>
58        A2$B1.Family.evc$Impl(self$1, self$)(inherited)(unit)
59      case A2$B2.A1$B2$$Exp(inherited) =>
60        A1$B2.Family.evc$Impl(self$1, self$)(inherited)(unit)
61    }
62    // Translation function implementations
63    override def A2$B2$$Exp(from: A2$B2.Exp): Exp = from
64    override def A2$B1$$Exp(from: A2$B1.Exp): Exp =
65      A2$B2.A2$B1$$Exp(from)
66    override def A1$B1$$Exp(from: A1$B1.Exp): Exp =
67      A2$B2.A2$B1$$Exp(A2$B1.Family.A1$B1$$Exp(from))
68    override def A1$B2$$Exp(from: A1$B2.Exp): Exp =
69      A2$B2.A1$B2$$Exp(from)
70  }
71 }
```

Figure 15. The translation of Persimmon code in Figure 14 to Scala code.

7.1 Design Goals

We aimed to achieve the following design goals with our solution, in addition to the classic goal of **type safety**:

- **Extensibility at scale**. We believe that extensibility should exist at the large scale of reusable, nested components. Persimmon achieves this through nested family polymorphism. All structural and hierarchical relationships between families are preserved during inheritance. We showcase the inheritance and extension of nested components in Persimmon with our extensible compilers example in Figure 4.
- **Scalable extensibility**. We believe that support for extensibility should not come at the cost of parameter clutter and painstaking advance preparation by the user. Code should look similar in the presence and absence of extensions, and dependencies between components should be minimized. Persimmon achieves this by treating most constructs as built-in extensibility hooks. Relative path types and path substitution keep inherited code type-safe, while keeping the names of base and derived constructs consistent. To highlight the user-friendly aspects of our approach as well as the expressive power of Persimmon, we encode a case study from the work on independently extensible solutions [Odersky and Zenger 2005a] in Figure 16, and discuss the comparison below.

```
1 (* Base code *)
2 Family Base {
3   type Exp = Num {v: N}
4   def eval: Exp -> N =
5       case Num(v: N) = v
6 }
7 (* Adding variants: plus and negation *)
8 Mixin BasePlus extends Base {
9   type Exp += Plus {l: Exp,r: Exp}
10  def eval: Exp -> N +=
11      case Plus(l: Exp,r: Exp) = (eval l) + (eval r)
12 }
13 Mixin BaseNeg extends Base {
14   type Exp += Neg {t: Exp}
15   def eval: Exp -> N +=
16       case Neg(t: Exp) = - (eval t)
17 }
18 Family BasePlusNeg extends Base with BasePlus, BaseNeg {}
19 (* Adding functionality: printing *)
20 (* Must define all cases at once due to exhaustivity of pattern matching *)
21 Mixin ShowPlusNeg extends BasePlusNeg {
22   def show: Exp -> Str =
23       case Num(v: N) = "" + v
24       case Plus(l: Exp,r: Exp) = (show l) + "+" + (show r)
25       case Neg(t: Exp) = "-(" + (show t) + ")"
26 }
27 (* Adding functionality: doubling *)
28 Mixin DblePlusNeg extends BasePlusNeg {
29   def dble: Exp -> Exp =
30       case Num(v: N) = Exp(Num{v = v * v})
31       case Plus(l: Exp,r: Exp) = Exp(Plus{l=dble(l),r=dble(r)})
32       case Neg(t: Exp) = Exp(Neg{t = dble(t)})
33 }
34 (* Mixing in both sets of added functionality *)
35 Family ShowDblePlusNeg extends BasePlusNeg
36   with ShowPlusNeg, DblePlusNeg {}
```

Figure 16. Code in PERSIMMON for the object-oriented decomposition case study by Odersky and Zenger [2005a]. The original code can be found in our appendix.

- **Mutual recursion**. We believe that the nested components of a program should support mutually recursive references to constructs in other components. PERSIMMON achieves this via our algorithmic linkage mechanism. Linkages provide a way to look up names and signatures of all constructs for type checking, including inherited and extended constructs, which may not be easily available in other representations.
- **Composable extensions**. We believe that parallel extensions should be composable, promoting code reuse and minimizing linear dependencies between families. Since nested family polymorphism has the expressive power to encode mixins, PERSIMMON supports composable extensions via an encoding (as shown in the case study in Figure 16). Since mixins in PERSIMMON are encoded as families, they can themselves nest families to arbitrary depth. Inheritance of nested components makes mixins in PERSIMMON especially powerful, allowing us to express examples such as the mixin compilers in Section 7.4.
- **Idiomatic functional style**. We believe that extensible programming should feel natural to a functional programmer and be user-friendly to novices. Programmers can enjoy the familiar functional style in PERSIMMON, while novices can enjoy the convenience of built-in extensible constructs.

7.2 Comparison to the Independently Extensible Solutions

Odersky and Zenger [2005a] propose two independently extensible solutions expressed in Scala: object-oriented decomposition and functional decomposition. With the first approach, new variants can be added easily using shallow mixin composition, but adding functionality requires deep mixin composition. On the other hand, the functional approach easily accommodates adding functionality, but variants must be added via deep mixin composition. Functional decomposition also requires the use of the Visitor pattern, adding extra code that is not relevant to the semantics of the extensions.

Our solution offers multiple advantages. The PERSIMMON code corresponding to the object-oriented decomposition example is shown in Figure 16. We also include the original example in the appendix for reader convenience. In our language, we can add both variants and functionality via shallow mixin composition (lines 18 and 35 in Figure 16), eliminating the need for deep mixin composition. Since PERSIMMON uses the same technique for extensibility in both dimensions, the user need not choose which dimension – variants or functionality – to prioritize. Another

advantage of PERSIMMON is that ADT constructors do not need to be manually re-parameterized by the extended type when functionality is added. For example, in object-oriented decomposition, adding functionality requires all variants to be restated to resolve the abstract expression type to the appropriate (extended) concrete type. In PERSIMMON, this resolution is accomplished automatically by path substitution and does not require any effort from the user. Finally, extensibility in PERSIMMON does not rely on the use of programming patterns, further reducing user burden and improving code readability.

7.3 Comparison to Compositional Programming

Zhang et al. [2021] propose compositional programming (CP): a new, highly modular programming style. This solution, while not presented as "functional", does support extensible variant types as well as nested family polymorphism via a unifying notion of first-class traits. An instance of an object that supports extended variants and extended functionality can be created using nested composition of traits. We include an example from this work in the appendix for reader convenience. PERSIMMON differs from CP in some important ways. PERSIMMON treats types as members of the family, while CP allows only top-level type definitions. In PERSIMMON, users can define types at the exact nesting level where they are needed, while avoiding excessive type parameterization and explicit type applications. Type instances in PERSIMMON can be constructed by simply using the name of the type; there is no need for manual composition of traits on the part of the user. Type members in PERSIMMON are also quite expressive, as they can refer to themselves and other type members recursively. The nested compilers example in Figure 4 is more difficult to model in CP, as the type members of the nested families are recursive within the family. CP requires explicit parameterization to express mutually defined data types. Finally, while CP strongly enforces the separation of interfaces and implementations, PERSIMMON takes the more familiar functional programming approach: both the interface and the implementation are specified within the family.

7.4 Case Study: Mixin Compilers

Finally, we highlight the expressive power of mixins in PERSIMMON with a case study of mixin compilers below. Mixin compilers are extensible compilers that are also composable in parallel. We build on the example in Figure 4, while making each compiler itself a mixin. PERSIMMON mixins are themselves families and thus contain nested families, which can be inherited and extended upon mixin composition. In comparison, other closely related works cannot support this example quite as elegantly. FPOP [Jin et al. 2023] does not support nested family polymorphism or unrestricted mutually recursive references between families, due to its application in a proof assistant. Compositional programming [Zhang et al. 2021] supports nested family polymorphism, but does not support the use of types as nested family members, making it difficult to represent types that are recursive via the family. We include a partial implementation of the figure below in the appendix.

```
1 Family BaseComp { (* contents of the base compiler *) }
2 Mixin IfExt extends BaseComp { (* mixin compiler that supports if-statements *) }
3 Mixin ArithExt extends BaseComp { (* mixin compiler that supports arithmetic *) }
4 Family IfArithComp extends BaseComp with IfExt, ArithExt {}
```

8 RELATED WORK

Family Polymorphism and Nested Inheritance. Families can be represented in OO systems using an *object-based* or a *class-based* approach. The seminal object-based solution by Ernst [2001] represents families as enclosing *family objects*, while the class attributes of the object comprise

family members. With this approach, any number of object instances (and thus, families) can exist at runtime. In the class-based approach, proposed in .FJ by Igarashi et al. [2005], a family is associated with the class itself. This approach restricts the number of families at run time, but has a more straightforward implementation as a nested class system: families are top-level classes, and family members are nested classes. Persimmon is inspired by the class-based approach, with top-level families nesting other families, types, functions, and `cases` as family members. Families in Persimmon are not types, and path types cannot be subtypes of each other due to potential unsafe uses [Ernst 2001; Igarashi et al. 2005]. This differs from a follow up work by Igarashi and Viroli [2007], where variant path types reconcile class-based family polymorphism with subtyping.

Jx, .FJ, *vc*, Tribe, and Familia are all class-based systems that support type-safe, nested family polymorphism [Clarke et al. 2007; Ernst et al. 2006; Igarashi et al. 2005; Nystrom et al. 2004; Zhang and Myers 2017]. Jx utilizes the notion of containers and their inheritable components (including nested containers), *vc* and Tribe are based around virtual classes, while Familia unifies the genericity mechanisms of inheritance and parametric polymorphism. These systems differ from Persimmon in that they do not support extensible variant types or pattern matching. Persimmon guarantees that pattern matching is type safe in the presence of extended variants and nested family polymorphism by introducing `cases`, which are direct family members and are thus polymorphic to the family.

Recently, Jin et al. [2023] presented a family polymorphism design (FPOP) for extensible metatheory mechanization, including type-safe, extensible pattern matching. Unlike Persimmon, FPOP does not support nested family polymorphism, limiting its capability for modular reuse. Furthermore, Persimmon supports unrestricted mutually recursive references, while FPOP cannot support this due to its application in a proof assistant.

Solutions to the Expression Problem. Persimmon meets some of the goals of the Expression Problem [Wadler et al. 1998]; namely, our calculus supports type-safe extension of data types *and* functionality over those data types. Other goals of the Expression Problem, such as modular type checking, are not met by the current calculus. Multiple works have since proposed an additional requirement that extensions should also be composable [Nystrom et al. 2006; Odersky and Zenger 2005a], which Persimmon also meets. Unlike J& [Nystrom et al. 2006], which introduces intersection types to support composable extensions, Persimmon encodes composition via nested families instead of introducing a new type. Odersky and Zenger [2005a,b] support composable extensions through the use of Scala traits, the Visitor pattern, and deep mixin composition. In comparison, Persimmon reduces user effort, as it does not require any setup of patterns or manual composition of mixins. Persimmon also cuts the parameter clutter by treating most constructs as built-in extensibility hooks. A recent object-oriented solution, SuperOOP, supports mixin composition and open recursion via late binding of the keywords `this` and `super` [Fan and Parreaux 2023]. Persimmon supports open recursion via relative path types. Unlike SuperOOP, Persimmon also supports the composition of arbitrarily nested families.

Oliveira and Cook [2012] use object algebras (an abstraction related to Church encodings) and rely on simple generics, which makes the solution applicable to mainstream languages. Object algebras are powerful abstractions that can express family polymorphism. One downside is that modular composition of object algebras requires manual setup by defining a combinator. In Persimmon, extensions can be composed in parallel using our convenient mixin syntax, as in Section 7.4.

Related to object algebras is also the "tagless final" approach, which relies on interpreters and a skillful embedding of DSLs in the host language [Kiselyov 2012]. Extensibility is achieved by adding syntactic forms or interpreters, and it is possible to abstract over families of interpreters [Carette et al. 2009]. The tagless final approach requires explicit parameterization of interpreter instances by the representation type, while Persimmon uses relative path types that adapt upon inheritance.

Continuing this line of work, Zhang et al. [2021] have recently proposed "compositional programming," a style for statically typed modular programming in a language design called CP, which solves the Expression Problem as well as more generally the problem of expressing dependencies in a modular way. Compared to Persimmon, CP still requires parameterization (in particular, self-type annotations to inject dependencies). Unlike CP, types in Persimmon are family members – they can be defined at any level of nesting, and can be recursive via the family.

Other EP solutions propose more flexible definitions of data types. For example, open data types and open functions can be scattered throughout modules, allowing the definitions to be provided at any point in the program [Löh and Hinze 2006]. Polymorphic variants [Garrigue 2000] allow constructors to exist independently of types, and support open pattern matching. In contrast, Persimmon keeps code safe for reuse in derived families by ensuring that code is polymorphic to the family. Families in Persimmon retain the organizational advantages of modules and support code reuse at a large scale via nested inheritance.

Extensible Variant Types and Pattern Matching. Some record-based solutions rely on row polymorphism to support extensible variants [Gaster and Jones 1996]. Gaster and Jones [1996] also propose an extension to their system, which makes pattern match cases first-class, extensible values. In a related work, [Blume et al. 2006] support extensible pattern matching and composable extensions via extensible first-class cases, capitalizing on the dual relationship between polymorphic records and sums. While these solutions support extensible variants and pattern matching, they do not support family polymorphism. In Persimmon, `cases` are not first-class, as their usage is restricted to application within a match expression; however, they are family members and are polymorphic to the family.

Zenger and Odersky [2001] implement extensible ADTs by providing default variants that subsume any future extensions. Their solution uses a new design pattern for extensible visitors. Pattern matching becomes extensible by delegating computation in the default case to the methods overridden in the extension. In Persimmon, the delegation is implicit thanks to relative path types and path substitution. Unlike Persimmon, this solution does not support composable extensions. Recently, [Zhang and Oliveira 2020] introduced a Scala-based solution using extensible generative visitors, which supports exhaustive and composable pattern matching. However, some exhaustivity checking for pattern matching must be delayed to the visitor instantiation site, whereas Persimmon ensures exhaustivity at definition.

OCaml has introduced extensible variant types as well as polymorphic variants [Garrigue 1998]. Extensible functions can be implemented by keeping a reference to the evolving function in a polymorphic record field [Balestrieri and Mauny 2018]. In Persimmon, functions are not extensible in the general case, but `cases` are directly extensible constructs within the family.

Extensible ML (EML), supports hierarchical, extensible data types and extensible functionality over those data types, while preserving modular type checking [Millstein et al. 2004]. Both Persimmon and EML support exhaustivity checking for pattern match expressions at definition. Syme et al. [2007] implement extensible pattern matching through the use of active patterns in *F#*, handling both partial and total decompositions. Persimmon does not support partial patterns due to the conflict with exhaustivity checking. `match` is an extensible language which implements extensible pattern matching for Racket using macros [Tobin-Hochstadt 2011]. JMatch [Isradisaikul and Myers 2013] is an extension of Java that provides modal abstraction (integration of pattern matching and iteration abstractions), where patterns are not tied to constructors. Both Persimmmon and JMatch ensure static exhaustivity checking, while `match` does not. Among the pattern match techniques evaluated by Emir et al. [2007], our solution is most similar to case classes in Scala. However, the

shortcoming of case classes – inability to define new patterns for new variants – is addressed in Persimmon with extensible cases.

9 FUTURE WORK

Our current design has some limitations that could be addressed in future work. As mentioned earlier, there is a conflict between on-demand linkage computation in Persimmon theory and modular type checking. Currently, Persimmon does not support separate type checking and compilation of programs that contain multiple fragments (for example, multi-file programs, where each file contains some of the dependencies). Type checking each file separately would require a dependency analysis that goes beyond Persimmon's current on-demand approach. While linkage concatenation can remain an on-demand operation for linking files together, the incomplete static linkages for each family would need to be pre-computed, along with the path context K of valid family paths for each file (currently, this context is global in the theory). Similarly, our code generation tool could be modified to support separate compilation by generating the necessary typing information for each file.

Unlike other languages that do not have separate `cases` constructs, all pattern match expressions in Persimmon must call a top-level `cases` construct. While this constraint simplifies the extension of `cases` constructs, it also precludes in-line nested pattern matches. In future work, we could support in-line nested pattern matching with syntax sugar, akin to the in-line match cases we already provide. However, this would require additional syntax support for extensibility, since the user must specify which pattern match in the nested structure they would like to extend.

Finally, unlike other functional systems such as ML and Haskell, Persimmon does not support global type inference. However, bidirectional type checking could be supported in the future.

10 CONCLUSION

We present Persimmon, the first functional system with nested family polymorphism and extensible variant types. Nested, extensible families in Persimmon combine the benefits of modules (code modularity and reuse), the benefits of family polymorphism (type safety of inherited and extended constructs), and the benefits of composable extensions. Linkages are the engine behind extensibility in Persimmon, eliminating the need for complex type checking and operational semantics. Our explicit `cases` constructs separate match case definitions from their uses, and provide a natural mechanism for extensibility of pattern matching. Exhaustivity of pattern matching is maintained by the well-formedness checking of definitions. Since types and `cases` in Persimmon serve as built-in extensibility hooks, parameter clutter is not an issue in our language.

DATA-AVAILABILITY STATEMENT

Our implementation, consisting of the Persimmon type checker and our prototype compiler to Scala, is available on Zenodo [Kravchuk-Kirilyuk et al. 2024].

ACKNOWLEDGMENTS

We would like to thank William Byrd, Stephen Chong, Samuel Grütter, John Li, and Yao Li for insightful discussions throughout the course of this work. We would also like to thank Aaron Bembenek, David Holland, George Klevorn, Joomy Korkut, Cameron Wong, and Kevin Zhang for their thoughtful suggestions on drafts. Finally, we thank our anonymous reviewers for their valuable feedback and suggestions to improve the paper.

This material is based upon work supported by the National Science Foundation under Award No. 2303983, and by the Amazon Research Awards program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Florent Balestrieri and Michel Mauny. 2018. Generic programming in OCaml. *arXiv preprint arXiv:1812.11665* (2018).

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2006. Extensible programming with first-class cases. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming - ICFP '06*. ACM Press, New York, New York, USA, 239. https://doi.org/10.1145/1159803.1159836

Luca Cardelli. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 266–277.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated Types with Class. In *ACM Symp. on Principles of Programming Languages (POPL)*.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th international conference on Aspect-oriented software development*. 121–134.

Burak Emir, Martin Odersky, and John Williams. 2007. Matching objects with patterns. In *European Conference on Object-Oriented Programming*. Springer, 273–298.

Erik Ernst. 2001. Family Polymorphism. In *ECOOP 2001 —Object-Oriented Programming*, Jørgen Lindskov Knudsen, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen (Eds.). Lecture notes in computer science, Vol. 2072. Springer Berlin Heidelberg, Berlin, Heidelberg, 303–326. https://doi.org/10.1007/3-540-45337-7_17

Erik Ernst. 2003. Higher-order hierarchies. In *European Conference on Object-Oriented Programming*. Springer, 303–328.

Erik Ernst, Klaus Ostermann, and William R Cook. 2006. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 270–282.

Andong Fan and Lionel Parreaux. 2023. super-Charging Object-Oriented Programming Through Precise Typing of Open Recursion. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML workshop*, Vol. 13. Baltimore.

Jacques Garrigue. 2000. Code reuse through polymorphic variants. Sasaguri, Japan.

Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants*. Technical Report. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham.

Atsushi Igarashi, Chieri Saito, and Mirko Viroli. 2005. Lightweight Family Polymorphism. In *Programming languages and systems*, Kwangkeun Yi, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). Lecture notes in computer science, Vol. 3780. Springer Berlin Heidelberg, Berlin, Heidelberg, 161–177. https://doi.org/10.1007/11575467_12

Atsushi Igarashi and Mirko Viroli. 2007. Variant path types for scalable extensibility. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 113–132.

Chinawat Isradisaikul and Andrew C. Myers. 2013. Reconciling Exhaustive Pattern Matching with Objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 343–354. https://doi.org/10.1145/2491956.2462194

Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023). https://doi.org/10.1145/3591286

Oleg Kiselyov. 2012. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 130–174.

Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested Family Polymorphism with Extensible Variant Types (Artifact). https://doi.org/10.5281/zenodo.10798266

Andres Löh and Ralf Hinze. 2006. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06*. ACM Press, New York, New York, USA, 133. https://doi.org/10.1145/1140335.1140352

O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. 1993. *Object Oriented Programming in the BETA Programming Language.* Addison-Wesley.

Todd Millstein, Colin Bleckner, and Craig Chambers. 2004. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 26, 5 (2004), 836–889.

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications - OOPSLA '04*. ACM Press, New York, New York, USA, 99. https://doi.org/10.1145/1028976.1028986

Nathaniel Nystrom, Xin Qi, and Andrew C Myers. 2006. J& nested intersection for scalable software composition. *ACM SIGPLAN Notices* 41, 10 (2006), 21–36.

Martin Odersky and Matthias Zenger. 2005a. Independently Extensible Solutions to the Expression Problem. ACM.

Martin Odersky and Matthias Zenger. 2005b. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*. ACM Press, New York, New York, USA, 41. https://doi.org/10.1145/1094811.1094815

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the masses. In *ECOOP 2012 – Object-Oriented Programming*, James Noble, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). Lecture notes in computer science, Vol. 7313. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

Simon Peyton Jones. 2009. Classes, Jim, But Not as We Know Them—Type Classes in Haskell: What, Why, and Whither. In *European Conf. on Object-Oriented Programming.*

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. 29–40.

Kresten Krab Thorup. 1997. Genericity in Java with virtual types. In *European Conf. on Object-Oriented Programming.*

Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).

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

Matthias Zenger and Martin Odersky. 2001. Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 241–252.

Weixin Zhang and Bruno C. d. S. Oliveira. 2020. Castor: Programming with extensible generative visitors. *Science of Computer Programming* 193 (2020), 102449.

Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems* 43, 3 (30 sep 2021), 1–61. https://doi.org/10.1145/3460228

Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (12 oct 2017), 1–31. https://doi.org/10.1145/3133894

Received 21-OCT-2023; accepted 2024-02-24