Persimmon: Nested Family Polymorphism with Extensible Variant Types

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 modularity often gives way to code duplication. We present our solution: Persimmon[1], the first functional system with nested family polymorphism, extensible variant types, and extensible pattern matching. Since most constructs in our language are built-in "extensibility hooks," we cut down on parameter clutter and user burden associated with extensible code. Relationships between nested families in Persimmon are maintained upon inheritance, enabling extensibility at a large scale. Persimmon also supports composable extensions by encoding, and separate compilation by translation to Scala. Finally, we show that our system is sound by proving progress and preservation.

1 Introduction

Writing modular, extensible code is hard. The Expression Problem epitomizes the difficulty [27]. 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 affordances in this task. The object-oriented programming style makes it easy to add new constructors (as classes), while adding new functions requires sweeping changes to all constructors. By contrast, the functional programming style makes it easy to add new functions, but adding new constructors to variant types requires sweeping changes to all functions. When changes to existing code are infeasible, the programmer has to 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, but has also been picked up by functional-language designers [6, 11, 30]. It allows extension to happen at the level of *families* of related types. Code is *polymorphic* to a family it is nested within, so code defined in a base family can be safely reused by derived families. Virtual classes [16], virtual types [25], and nested inheritance [18] are all forms of family polymorphism.

The power of family polymorphism has not been fully realized in the design of functional languages, however. Although associated types [3] in Haskell are inspired [23] by virtual types, they do not provide the same level of extensibility that family polymorphism can offer in the OO setting. In particular, no existing family polymorphism design supports extensible variant types.

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 [8]. Hence, we consider it critical that a deeper integration of 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 and pattern matching is the tension between type safety and modular type checking. To ensure that code is type safe, we must check *exhaustivity* of pattern matching – there must exist a match case for each variant. This requirement is inimical to modular type checking: to ensure exhaustivity of pattern matching in the derived family despite extended variants, we would need access to definitions in the base family. Re-checking inherited code in the derived family is not modular, as we expect to only check the base family once.

[1] The name is a pun on the Law of Parsimony.

Fundamentally, the tension results from the duality between variants and records. It is always safe for a record to have more fields – or for a variant to have fewer constructors – than specified in its type. Any inherited functionality defined for a base variant type should be type safe for use with new variants, but we must ensure exhaustivity of pattern matching. Our design reconciles this tension. We introduce `cases` constructs, nominal pattern matching expressions that are automatically polymorphic to their enclosing families. `cases` exploit the duality that caused the tension in the first place: it is always safe for a `cases` definition to handle more constructors than specified in its type.

1.1 Design Considerations

Solutions to the Expression Problem (EP) already exist for functional programming. This paper is not just another solution; it is concerned with additional design goals beyond those required by the Expression Problem. Specifically, we aim for a language design that meets the following goals *in addition* to the classic EP goals (such as **modular typechecking**, **separate compilation**, and **type safety**).

- ➢ **Extensibility at scale**. It is a pity that many solutions to EP 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 can nest modules) will enable the programmer to create *extensible software frameworks* with ease.
- ➢ **Scalable extensibility**. Little bookkeeping should be required of the programmer before any extension is introduced, and programmer effort required to implement extensions should be proportional to the delta in program functionality. It is not uncommon to address extensibility by explicitly parameterizing a unit of code with extensibility hooks. EP 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.
- ➢ **Composable extensions**. Extensions should be not just possible, but also 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 language design that supports extensible variant types via family polymorphism while maintaining type safety and modular type checking. The design is based on a simple functional core and is thus applicable to other functional languages.
- We showcase the expressive power of our design, and its applicability to real programming challenges such as extensible compilers. Our examples show that the language design not only solves the Expression Problem but also meets the additional 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. Importantly, our compiler supports separate compilation: code compiled for a base family is shared by its derived families.

STLC

types	τ	::= unit | $\tau_1 \to \tau_2$
values	v	::= () | x | $\lambda x.\, e$
expressions	e	::= v | $e_1\ e_2$

STLC extended with if

types	τ	::= ... | bool
values	v	::= ... | true | false
expressions	e	::= ... | if e_0 then e_1 else e_2

IL

types	τ	::= unit | $(\vec{\tau}) \to$ void
values	v	::= () | x
expressions	e	::= let $x = v$ in e | $v(\vec{v})$ | halt v
global functions	F	::= $f = \text{fun}(\vec{x}).\, e$

IL extended with if

types	τ	::= ... | int
values	v	::= ... | i
expressions	e	::= ... | if e_0 then e_1 else e_2

IL_K, extending IL — *CPS conversion compiles* STLC *into* IL_K

values	v	::= ... | $\lambda \vec{x}.\, e$	IL_K *allows nested lambdas*

IL_C, extending IL — *Closure conversion compiles* IL_K *into* IL_C

types	τ	::= ... | α | $\exists \alpha.\, \tau$	*Existentials for abstracting closure environment*
values	v	::= ... | pack $[\tau, v]$ | f	
expressions	e	::= ... | let $[\alpha, x]$ = unpack v in e	
programs	P	::= $\vec{F}; e$	*In* IL_C*, all lambdas are hoisted to the top*

Figure 1. Motivating example: STLC, intermediate languages for compiling STLC, and language extensions.

2 Motivation

We use an example of extensible compilers for simply-typed lambda calculus (STLC) to show how powerful (and practical!) the combination of nested family polymorphism and extensible variant types can be. Consider a compiler that can (1) transform 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 by having both target languages extend some shared, intermediate language. In Figure 1, the left column shows the language components of the compiler: the source language (STLC), the shared language IL, the target language of continuation-passing style (IL_K, which extends IL), and the target language of closure conversion (IL_C, which extends IL).

Already, we see the need for extensible variant types. Types, values, and expressions in the target languages IL_K and IL_C are represented using algebraic data types (ADTs). Since both target languages extend an intermediate, shared language IL, the ADTs we use must be extensible. Otherwise, extended types will not be able to share names with their inherited counterparts, creating name confusion and unnecessary duplication of concepts. Moreover, without extensible variant types, functions that operate on values v in IL cannot automatically adapt to operate on extended values v in IL_K or IL_C, limiting reuse of inherited functionality.

One way to implement extensible variant types is by use of family polymorphism. We can use relative path types to ensure that constructs defined within each family are polymorphic to the family [11]. For example, the ADT e defining expressions in IL relies on the definition of values v in IL. However, when v is extended in IL_K, e does not have to be redefined within IL_K as it refers implicitly to the extended type v 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 would happen to our compiler if the source language STLC was extended with if-expressions (the left-hand side of Figure 1)? Normally, we would be forced to build a new compiler for each extension of STLC, while only modifying the components in small ways. We would much rather have an extensible compiler instead. Figure 2 shows the setup for two compilers: the `BaseComp` compiler for base

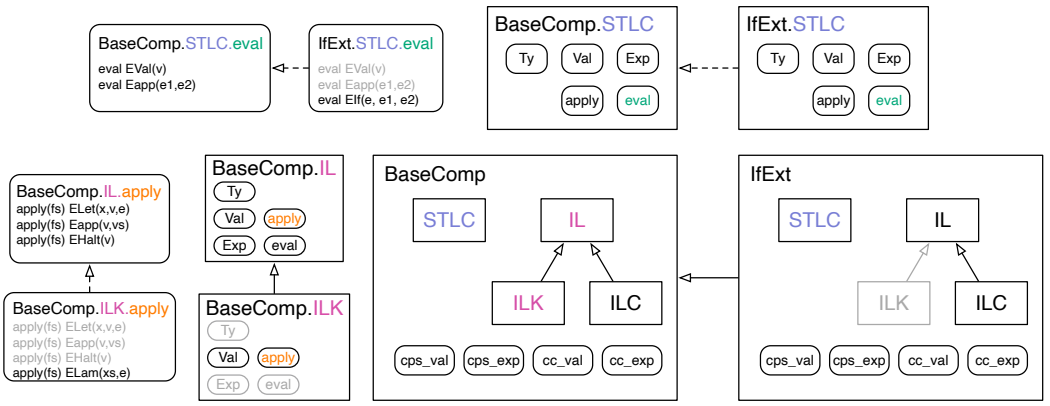


Figure 2. Motivating example: extensible compilers with nested inheritance.

STLC, and the IfExt compiler for STLC with if-expressions. The compiler IfExt is an extension of BaseComp, and its component languages STLC, IL, and ILC are extensions of their counterparts in BaseComp. There is a clear need for *nested family polymorphism*: we should be able to inherit any nested components from the BaseComp compiler, and use the code safely within IfExt. 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 via mixin families, further detailed in Section 3.3.

3 Nested Family Polymorphism, Functionally

In this section, we present the key features of our language, Persimmon, via case studies. Our case studies highlight extensible variant types and pattern matching, nested families and inheritance of nested components, and mixin capabilities of our language. Here, we ease the reader into the syntax and demonstrate the expressivity of the language before introducing our formal calculus in Section 4.

3.1 Extensible Variant Types and Extensible Pattern Matching

First, we introduce the syntax of our language, focusing on extensible variant types and extensible pattern matching. Consider the Persimmon code snippet in Figure 3. Here, we show evaluation for a base lambda calculus (STLC) with natural numbers and unit, implemented in extended Persimmon syntax.[2] The base calculus is contained in family STLCBase. Within this family, we declare algebraic data types (ADTs) such as Exp and Ty, representing expressions and types in the base calculus. Each ADT is defined using the keyword type, as a set of constructors with corresponding fields in parentheses (omitted if empty). For example, the constructor Var of type Val has a single field x of type str, while the constructor Unit has no fields. Functions are declared with the keyword val, such as the function eval on line 5 and apply on line 9. Each function must specify a name, an arrow type, and a definition. For functions involving pattern matching, such as eval and apply, each pattern match case is specified separately.[3] At this point, we have not yet extended any

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

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

```
Family STLCBase {
    type Ty = TUnit | TNat | TArr(t1: Ty, t2: Ty)
    type Val = Unit | Var(x: str) | Lam(x: str, e: Exp)
    type Exp = EVal(v: Val) | EApp(e1: Exp, e2: Exp)

    def eval : Exp -> Option Val =
        case EVal(v) = Some v;
        case EApp(e1, e2) =
            (eval e1) >>= (λ v -> apply(e2) v)

    def apply(e2: Exp) : Val -> Option Val =
        case Lam(x, e) = eval (subst x e2 e);
        case _ = None

    ... (* subst, tc, print, etc. *)
}
```

```
Family STLCIf extends STLCBase {
    type Ty += TBool
    type Val += True | False
    type Exp += EIf(e: Exp, e1: Exp, e2: Exp)

    def eval : Exp -> Option Val +=
        case EIf(e, e1, e2) =
            (eval e) >>= (λ v -> branch(e1, e2) v)

    def branch(e1: Exp, e2: Exp) : Val -> Option Val =
        case True = eval e1;
        case False = eval e2;
        case _ = None

    ... (* extensions to subst, tc, print, etc. *)
}
```

Figure 3. A base lambda calculus (left) and an extension (right) in extended PERSIMMON syntax.

constructs, and the base code is quite ordinary. This is one advantage of PERSIMMON: our code follows the familiar, functional programming style that users are already comfortable with, and no prior set up is required in base families to enjoy extensibility in the derived families.

Family `STLCIf` in Figure 3 shows the elegance of extensible variant types and pattern matching in PERSIMMON. `STLCIf` is an extension to the base calculus that adds booleans and if-expressions to the constructs from `STLCBase` using our extensibility marker `+=`. For example, the type `Val` is extended with `True` and `False`, and a new case for the if-expression is added to function `eval`. PERSIMMON ensures exhaustivity of pattern matching: 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 definitions to derived families, such as function `branch` in `STLCIf`.

Note that only the new variants and pattern match cases need to be specified in `STLCIf`; the rest are inherited as-is from `STLCBase`. Inherited code is automatically polymorphic with respect to the enclosing family and thus automatically adapted to the extended types. This is accomplished by the use of relative path types: each type in Figure 3 has an implicit path prefix, which is inferred. When code is inherited, any path prefix referring to the base family is replaced by the path prefix referring to the derived family. We further discuss path types in 4.1.

Our approach has multiple advantages. We achieve type safe code reuse with inherited code that is polymorphic to the enclosing family. We achieve code modularity and readability due to a minimal code overlap between base and derived families. We achieve built-in extensibility of constructs by inherently treating them as "extensibility hooks," eliminating parameter clutter associated with extensibility. We also reduce programmer effort associated with the set up of extensible frameworks, as compared to design patterns.

3.2 Nested Families and Inheritance

In this section, we explore nested families and inheritance of nested components in PERSIMMON. To do this, we take a closer look at using nested families to implement the extensible compilers example in Figure 1. We show partial PERSIMMON code for this example in Figure 4, and an inheritance diagram of all nested components in Figure 2. This implementation assumes built-in option types, `let` expressions, and pairs for convenience.

First, we show how nested families are defined in PERSIMMON and how inheritance relationships between nested families are created. PERSIMMON supports an arbitrary level of nesting within families. In Figure 4, the base compiler is represented by family `BaseComp`. Within this family, we have four nested families representing languages that are necessary for compilation: (1) the source

```
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) =
13         (eval(fs) v) >>= (λ v -> apply(fs, vs) v);
14       case EHalt(v) = Some v
15     def apply(fs: List Fun, vs: List Val): Val -> Option Val =
16       case _ = None
17     ... (* subst, tc, print, etc. *)
18   }
19   (* target language of CPS *)
20   Family ILK extends IL {
21     type Val += Lam(xs: List str, e: Exp)
22     def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
23       case Lam(xs,e) = eval(fs) (subst xs vs e)
24     ... (* subst, tc, print, etc. *)
25   }
26   (* target language of closure conversion *)
27   Family ILC extends IL {
28     type Ty += TVar(α: str) | TExist(α: str, t: Ty)
29     type Val += Pack(t: Ty, v: Val) | Name(n: str)
30     type Exp += EUnpack(α: str, x: str, v: Val, e: Exp)
31     def eval(fs: List Fun): Exp -> Option Val +=
32       case EUnpack(α,x,v,e) = unpack(fs,α,x,e) v
33     def unpack(fs: List Fun, α: str, x: str, e: Exp):
34         Val -> Option Val =
35       case Pack(t, v) = eval(fs) (subst x v (subst α t e));
36       case _ = None
37
38     def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
39       case Name(n) =
40         let (xs, e) = lookup(fs, n) in eval(fs) (subst xs vs e)
41       case Pack(t, v) = None
42     ... (* subst, tc, print, etc. *)
43   }
44   (* CPS translation *)
45   def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp = ... (* cps_val cases *)
46   def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp = ... (* cps_exp cases *)
47   (* closure conversion *)
48   def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) = ... (* cc_val cases *)
49   def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) = ... (* cc_exp cases *)
50 } (* end of BaseComp family *)
51
52 (* Compiler extension: add if-then-else to STLC and ILs *)
53 Family IfExt extends BaseComp {
54   Family STLC extends STLCIf {}
55   Family IL {
56     type Ty += TInt
57     type Val += Int(i: int)
58     type Exp += EIf(v: Val, e1: Exp, e2: Exp)
59     def eval(fs: List Fun): Exp -> Option Val += ... (* new Elf case *)
60     def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
61       ... (* new Int case *)
62     ... (* subst, tc, print, etc. *)
63   }
64   Family ILC extends IL {
65     def unpack(fs: List Fun, α: str, x: str, e: Exp):
66         Val -> Option Val +=
67       case Int(i) = None
68     ... (* subst, tc, print, etc. *)
69   }
70   def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp += ... (* new cases *)
71   def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp += ... (* new cases *)
72   def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) += ... (* new cases *)
73   def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) += ... (* new cases *)
74 }
```

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

language `STLC`, (2) the base intermediate language `IL`, (3) the target language of CPS, `ILK`, and (4) the target language of closure conversion, `ILC`. Both target languages `ILK` and `ILC` extend the intermediate language family `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 45–46. Since the types of these functions are family polymorphic, the functions can be safely reused for translation 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). Furthermore, we 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 50, family `IfExt` represents the compiler that has been extended with if-expressions in the source language `STLC`. Only the new constructs are added in the extension, such as boolean types, if-expressions, and the new match cases for `eval` and `apply` (omitted from figure). All unchanged constructs from `BaseComp.STLC` are inherited and do not need to be repeated in the extension. The nested family `IfExt.IL` is extended with necessary constructs and further binds `BaseComp.IL`.

```
1  (* Mixin example to encode *)
2  Mixin IfExt extends STLCBase {
3    (* ... contents of mixin IfExt *)
4  }
5  Mixin ArithExt extends STLCBase {
6    (* ... contents of mixin ArithExt *)
7  }
8  Family STLCIfArith extends STLCBase
9    with IfExt, ArithExt {}
```

```
9  (* Mixin encoding in Persimmon *)
10 Family IfExt {
11   Family Base extends STLCBase {}
12   Family Derived extends Base { (* ... contents of mixin IfExt *) }
13 }
14 Family ArithExt {
15   Family Base extends STLCBase {}
16   Family Derived extends Base { (* ... contents of mixin ArithExt *)  }
17 }
18 Family IfExtBuild extends IfExt {
19   Family Base extends STLCBase {}
20 }
21 Family ArithExtBuild extends ArithExt {
22   Family Base extends IfExtBuild.Derived {}
23 }
24 Family STLCIfArith extends ArithExtBuild.Derived {}
```

Figure 5. An encoding of mixins in PERSIMMON.

Family `IfExt.ILC` must be 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, the translation functions must be extended with new pattern match cases (omitted from figure). Figure 2 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, preserving the structural and hierarchical relationships between them in the derived family. We provide family polymorphic guarantees: inherited code is type safe for use in a derived family, and prevent interaction between members of incompatible families (for example, a function defined in the derived family cannot be called on a type from the base family, if the type has been extended). The extensible compilers example in PERSIMMON shows how these benefits can be enjoyed together.

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 imposing new inheritance relationships. For example, consider different versions of `STLC` with varying combinations of features: `STLCIfArith` with if-expressions and arithmetic, `STLCLetRef` with let-expressions and references, etc. Ideally, we would like to define an extension for each feature only once, and then create composite extensions with arbitrary combinations of features. Lines 1–9 in Figure 5 show how this can be accomplished using mixins. `STLCIfArith` on lines 8–9 is an extension of `STLCBase` with mixed in functionality for handling if-expressions and arithmetic. We use this code snippet as a roadmap for our PERSIMMON encoding.

We encode mixins in PERSIMMON by combining linear extension with a flexible base for extension (the code on the right in Figure 5). Each family representing a mixin has 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 actual extension code. This nested family structure ensures that the dependencies between extensions are flexible, and that the extensions are composable.

Lines 18–24 in Figure 5 show how we perform composition of extensions `IfExt` and `ArithExt` in PERSIMMON. This code is a direct translation from the roadmap syntax on lines 8–9.[4] We

[4]Although PERSIMMON does not support the exact mixin syntax on the left-hand side of Figure 5, it could be easily implemented as syntax sugar.

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	K	::=	$[] \mid [sp] \mid sp :: \text{K}$			

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 (+)?= $\{(f_i : T_i(= v_i)?)*\}$

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

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

casesdef ::= **cases** c $\langle a.R \rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}$ (+)?= $\lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\}$

Figure 6. The Persimmon syntax.

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. We could have also chosen to omit line 27 as it is redundant to say that `STLCBase` further binds itself, but we include it for a more accurate translation.

4 The Persimmon Calculus

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

- First, we discuss the calculus syntax and in particular representations for relative paths which enable family polymorphism, nested families, and top-level `cases` constructs (Section 4.1).
- Next, we detail the map-like data structures – linkages – that are used to store all information about each family in our system (Section 4.2). Inheritance, further binding of families, and extensibility of data types and pattern matching are handled during nested linkage concatenation, a recursive operation which combines linkages for a base family and a derived family.
- Finally, we show that our type system relies heavily on the computed linkages and their contents, and as a result is fairly straightforward. Similarly, our operating semantics relies on linkages as well (Sections 4.3 and 4.4).

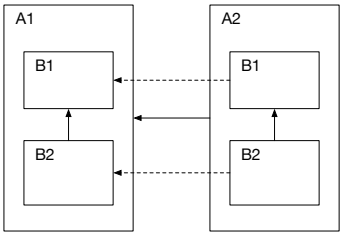
Underlying the linkage engine, type system, and operating semantics is the unifying notion of well-formedness: well-formed family definitions parse into well-formed linkages, and well-formedness of linkages is preserved by concatenation. Exhaustivity of pattern matching is checked at program definition as part of the program's well-formedness (Section 4.3).

4.1 Syntax

The full syntax of Persimmon is shown in Figure 6.

Family Names and Paths. Families are identified by their names, e.g. `Family` A. In the larger context of a program, each family has a unique path that specifies the nesting depth of the family with respect to the program path, `prog`. For example, family A' nested within family A has path $\texttt{prog}.A.A'$. We also distinguish *relative paths*, which reference the current family using the keyword `self`, such as $\texttt{self}(\texttt{prog}.A)$. When code is inherited, relative paths referring to the base family are

```
1  Family A1 {
2    Family B1 {
3      type Exp = ENat {n : N}
4      val f: N -> N = lam (n: N). n
5      val ev: Exp -> N = lam (e: Exp). match e with evc {}
6      cases evc <Exp> : {} -> {ENat: {n: N} -> N} =
7        lam (unit: {}). {ENat = lam (x: {n: N}). x.n}
8    }
9    Family B2 extends self(A1).B1 {
10     val f: N -> N = lam (n: N). n+1
11   }
12 }
13 Family A2 extends A1 {
14   Family B1 {
15     val f: N -> N = lam (n: N). n+2
16   }
17   Family B2 extends self(A2).B1 {
18     type X = {x: B}
19     type Exp += EPlus {e1: Exp, e2: Exp}
20     cases evc <Exp> : {} -> {EPlus: {e1: Exp, e2: Exp} -> N} +=
21       lam (unit: {}). {EPlus = lam (x: {e1: Exp, e2: Exp}).
22         (ev(x.e1) + ev(x.e2))
23     }
24   }
25 }
```



`A2.B2.f(3)` returns 4 because `f` is further bound in `A1.B2` which takes precedence over the extension `A2.B1`. See Figure 9.

Figure 7. PERSIMMON code snippet (left) exhibiting both inheritance and further binding, and its diagram (right) with the *extends* (solid) and *further binds* (dotted) links.

updated to reference the derived family. This keeps the inherited code up to date and compatible with latest extensions. `prog` is also a valid relative path, and is the prefix to all family paths.

Types. Our types include natural numbers `N`, booleans `B`, arrow types, record types, and path types $a.R$, where a is the path to the family in which type R is defined. Note that there is no raw ADT type – all ADTs in our system are path types with ADT definitions (for example, ADT `Exp` on line 3 in Figure 7).

Expressions. Our expressions include natural numbers n, booleans b, variables x, applications, lambda abstractions, records, projections $e.f$, and if-expressions in their usual forms. Our function calls, $a.m$, specify the family path a in which the function m appears. We also have an expression $a.c$ to select a `cases` definition c from family path a, akin to a function call.

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. The latter must specify a valid constructor name C to create an instance of ADT $a.R$. In any instance expression, all fields f_i of a record type or an ADT constructor must receive a corresponding input e_i, or be filled with a default value for that field (discussed later).

Our `match` expressions have special shape due to the `cases` constructs involved (for an example, see the function `ev` and the corresponding `cases` construct `evc` on lines 5–6 in Figure 7). Within the match expression, `match` e `with` $a.c\ \{(f_{arg} = e_{arg})*\}$, we select the appropriate `cases` definition $a.c$ (which contains the corresponding match cases) and apply it to a record of arguments $\{(f_{arg} = e_{arg})*\}$. These arguments represent any additional information needed for the pattern match, such as variables referenced within the match cases. We also refer to this additional information as the *match context*. The shape of our `match` expressions forces the use of top-level `cases` constructs for pattern matching (applied within the body). This also makes translation of our match expressions to other languages more straightforward.

$\boxed{\mathrm{K} \vdash a \rightsquigarrow L}$ $\boxed{\mathrm{K} \vdash a \rightsquigarrow_{\approx} L}$

$$\frac{\texttt{parse(p)} = L}{\mathrm{K} \vdash \texttt{prog} \rightsquigarrow L}\ \text{(L-Prog)} \qquad \frac{\mathrm{K} \vdash a.A \rightsquigarrow_{\approx} L \quad \texttt{self}(a.A) \in K}{\mathrm{K} \vdash \texttt{self}(a.A) \rightsquigarrow L}\ \text{(L-Self)} \qquad \frac{\mathrm{K} \vdash a.A \rightsquigarrow_{\approx} L \quad L\,[a.A \,/\, \texttt{self}(a.A)] = L'}{\mathrm{K} \vdash a.A \rightsquigarrow L'}\ \text{(L-Sub)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad L'' = L.A \qquad \mathrm{K} \vdash L''.\texttt{super} \rightsquigarrow_{\approx} L' \qquad L' + L'' = L'''}{\mathrm{K} \vdash a.A \rightsquigarrow_{\approx} L'''}\ \text{(L-Nest)}$$

Figure 8. Rules for computing linkages.

Programs and Definitions. A Persimmon program contains an arbitrary number of family definitions and a main expression. Each family definition can contain nested family definitions, record type and ADT definitions, function definitions, and top-level `cases` construct definitions. Record types, ADTs, and `cases` construct can be marked as extensions for existing definitions using symbol (+)=. A `cases` definition (for example, `evc` in Figure 7) specifies a lambda "handler" for each constructor, and its output type explicitly mentions the constructors and the types of their "handlers". This detailed syntax is used in our system for ease of typechecking, but users can enjoy the convenient in-line syntax as shown earlier.

4.2 Linkage Computation

Next, we discuss the extensibility engine of our system – linkages. A linkage is a data structure (essentially, a map of maps) containing all information about a single family (such as types, functions, and `cases` definitions). The linkage for a given family also stores the self-path to that family and a path to the parent family, if one exists. The self-path is used for updating relative path types within inherited components, and the parent path is stored for linkage concatenation. When the program is parsed, an "incomplete" linkage for each family is created, containing only the code that appears directly in that family. Incomplete linkages do not include any inherited code just yet. We then compute a "complete" linkage (which include inherited and extended components) for each family by recursive linkage concatenation. A linkage may be complete even if it has incomplete nested linkages, since we compute complete linkages from the "outside" in (we address this shortly). During concatenation, an incomplete linkage for a derived family is combined with a complete linkage for its parent, resulting in a a complete linkage for the derived family. Since complete linkages account for all inherited, extended, and overwritten components in a derived family, we consider them to be the engine behind extensibility in our system. Both our type system and our operational semantics (Section 4.3 and Section 4.4) rely on complete linkages, making those formalisms more straightforward. Below, we detail the three main parts of the linkage engine: parsing of families into incomplete linkages, linkage computation for a given family path, and linkage concatenation of parent families and derived families.

Parsing. We assume the existence of a single program (`p`) at path `prog` in our system. The program contains a collection of families and a main expression. When the program is parsed, a linkage is created for the program itself and for each family within the program. The linkage for the program simply contains the self-path (`prog`), the null parent path, and nested linkages for all families within the program. The linkage for each family definition contains the self-path to the family, the parent path, linkages for all nested families, and all other definitions such as types, functions, and `cases`. For convenience, we separate the type itself from its defaults when parsing record type definitions. The full rules are available in the appendix (Figure 14).

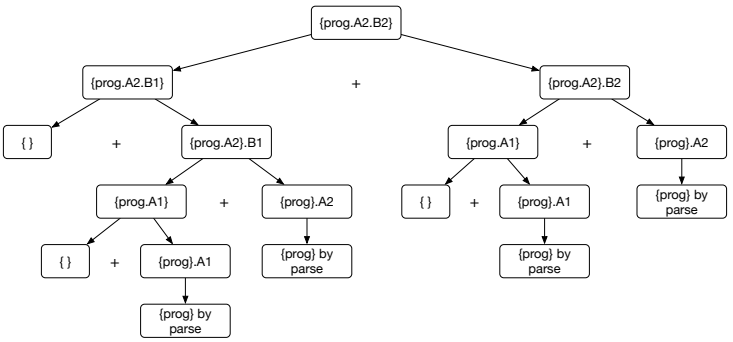


Figure 9. Linkage computation tree for family `A2.B2` in Figure 7. The shorthand notation $\{a\}$ stands for "the linkage for path a," while $\{a\}.A$ retrieves the nested linkage for family A from $\{a\}$. The linkage `{prog.A2}.B2` accounts for the further binding of family `A1.B2`, and takes precedence over the parent linkage `{prog.A2.B1}` during concatenation (+).

Computation of Linkages. We compute a complete linkage L for family path a as shown in Figure 8. Before diving into the details, let us consider a more intuitive example in Figure 9. Here, we see a computation tree for a single linkage. The + operation represents linkage concatenation. At each leaf, there is an empty parent linkage (the end of an inheritance chain) or the linkage for path `prog`. This indicates that complete linkages are computed from the "outside" in – we compute linkages for wrapper families before linkages for nested families. To compute the linkage for family path `prog.A2.B2` we must first compute the complete linkage for its parent path (`prog.A2.B1`), and concatenate it with the incomplete linkage for `prog.A2.B2`. We retrieve the latter by first computing the complete linkage for the *wrapper* family path, `prog.A2`, and projecting the corresponding nested linkage for `B2`. The arrows in the tree represent recursive linkage computation triggered by rule `L-Nest`, the + represents the concatenation in rule `L-Nest`, and {`prog`} leaves represent rule `L-Prog`. We omit the path substitution rules (`L-Self` and `L-Sub`) for readability.

Now, the details. We make a distinction between *exact* linkage computation (marked $\rightsquigarrow$) and *inexact* linkage computation (marked $\rightsquigarrow_{\approx}$). When the self-path to the current family in L is a, the computation $\mathrm{K} \vdash a \rightsquigarrow L$ is exact. However, we may need to compute a linkage for a path that is not relative – for example, some nested family $a.A$. Since a self-path of a linkage must be relative to account for future extensions, we introduce an inexact linkage computation where the non-relative path is wrapped in the keyword `self(...)`. Simply put, in an inexact computation $\mathrm{K} \vdash a \rightsquigarrow_{\approx} L$, the computed linkage L refers to family path a as `self`(a) throughout.

Having clarified this distinction, we explain the purpose of each rule in Figure 8. `L-PROG` computes the linkage for path `prog` by parsing the program. `L-SELF` and `L-SUB` convert between exact and inexact computations of linkages by path substitution. `L-SELF` also checks whether we are at the proper level of nesting to compute the linkage for a given self-path by ensuring it is in the path context K. Finally, `L-NEST` performs the concatenation of linkages for base and derived families. We first compute the complete linkage L for family path a inside which the family $a.A$ is nested. Then, we retrieve the nested incomplete linkage L'' for family A from L. Once we compute the complete linkage L' for the parent path, we are ready to perform the concatenation.

Linkage Concatenation. When we concatenate linkages for a base family and a derived family, we must recursively perform concatenation on all nested linkages and definitions. This process ensures that the complete linkage for the derived family contains all inherited, extended, and overwritten

$$\frac{\begin{array}{c} NEST_1\ [sp_2\ /\ sp_1]\ +\ NEST_2\ =\ NEST_3 \\ TYPES_1\ [sp_2\ /\ sp_1]\ +\ TYPES_2\ =\ TYPES_3 \\ DEFS_1\ [sp_2\ /\ sp_1]\ +\ DEFS_2\ =\ DEFS_3 \\ ADTS_1\ [sp_2\ /\ sp_1]\ +\ ADTS_2\ =\ ADTS_3 \\ FUNS_1\ [sp_2\ /\ sp_1]\ +\ FUNS_2\ =\ FUNS_3 \\ CASES_1\ [sp_2\ /\ sp_1]\ +\ CASES_2\ =\ CASES_3 \end{array}}{\left\{\begin{array}{l}\texttt{self} = sp_1 \\ \texttt{super} = a_1 \\ NEST_1 \\ TYPES_1 \\ DEFS_1 \\ ADTS_1 \\ FUNS_1 \\ CASES_1\end{array}\right\} + \left\{\begin{array}{l}\texttt{self} = sp_2 \\ \texttt{super} = a_2 \\ NEST_2 \\ TYPES_2 \\ DEFS_2 \\ ADTS_2 \\ FUNS_2 \\ CASES_2\end{array}\right\} = \left\{\begin{array}{l}\texttt{self} = sp_2 \\ \texttt{super} = a_2 \\ NEST_3 \\ TYPES_3 \\ DEFS_3 \\ ADTS_3 \\ FUNS_3 \\ CASES_3\end{array}\right\}} \quad \text{(CAT-TOP)}$$

$$\frac{\begin{array}{c} NEST_3 = \{A \mapsto L \in NEST_1 \bigsqcup NEST_2\} \cup \{A \mapsto L : \mathcal{P}(A, L)\} \\ \mathcal{P}(A, L) = A \mapsto L_1 \in NEST_1 \wedge A \mapsto L_2 \in NEST_2 \wedge L_1 + L_2 = L \end{array}}{NEST_1 + NEST_2 = NEST_3} \quad \text{(CAT-NEST)}$$

$$\frac{\begin{array}{c} TYPES_3 = \{R_{(=)} \mapsto \{(f_k : T_k)*\} \in TYPES_1 \bigsqcup TYPES_2\} \cup \{R_{(=)} \mapsto \{(f_k : T_k)*\} : \mathcal{P}(R, \{(f_k : T_k)*\})\} \\ \mathcal{P}(R, \{(f_k : T_k)*\}) = R_{(=)} \mapsto \{(f_i : T_i)*\} \in TYPES_1 \wedge R_{(+=)} \mapsto \{(f_j : T_j)*\} \in TYPES_2 \wedge \\ \{(f_i : T_i)*\} + \{(f_j : T_j)*\} = \{(f_k : T_k)*\} \end{array}}{TYPES_1 + TYPES_2 = TYPES_3} \quad \text{(CAT-TYPES)}$$

$$\frac{\begin{array}{c} DEFS_3 = \{R_{(=)} \mapsto \{(f_k = v_k)*\} \in DEFS_1 \bigsqcup DEFS_2\} \cup \{R_{(=)} \mapsto \{(f_k = v_k)*\} : \mathcal{P}(R, \{(f_k = v_k)*\})\} \\ \mathcal{P}(R, \{(f_k = v_k)*\}) = R_{(=)} \mapsto \{(f_i = v_i)*\} \in DEFS_1 \wedge R_{(+=)} \mapsto \{(f_j = v_j)*\} \in DEFS_2 \wedge \\ \{(f_i = v_i)*\} + \{(f_j = v_j)*\} = \{(f_k = v_k)*\} \end{array}}{DEFS_1 + DEFS_2 = DEFS_3} \quad \text{(CAT-DEFS)}$$

$$\frac{\begin{array}{c} ADTS_3 = \{R_{(=)} \mapsto \overline{C_k\ \{(f_n : T_n)*\}} \in ADTS_1 \bigsqcup ADTS_2\} \cup \{R_{(=)} \mapsto \overline{C_k\ \{(f_n : T_n)*\}} : \mathcal{P}(R, \overline{C_k\ \{(f_n : T_n)*\}})\} \\ \mathcal{P}(R, \overline{C_k\ \{(f_n : T_n)*\}}) = R_{(=)} \mapsto \overline{C_i\ \{(f_j : T_j)*\}} \in ADTS_1 \wedge R_{(+=)} \mapsto \overline{C'_i\ \{(f'_j : T'_j)*\}} \in ADTS_2 \wedge \\ \overline{C_i\ \{(f_j : T_j)*\}} + \overline{C'_i\ \{(f'_j : T'_j)*\}} = \overline{C_k\ \{(f_n : T_n)*\}} \end{array}}{ADTS_1 + ADTS_2 = ADTS_3} \quad \text{(CAT-ADTS)}$$

$$\frac{\begin{array}{c} FUNS_3 = \{m \mapsto (T \to T', \lambda(x : T).e) \in FUNS_1 \bigsqcup FUNS_2\} \cup \{m \mapsto (T \to T', \lambda(x : T).e) : \mathcal{P}(m, (T \to T', \lambda(x : T).e))\} \\ \mathcal{P}(m, (T \to T', \lambda(x : T).e)) = m \mapsto (T \to T', \lambda(x' : T).e') \in FUNS_1 \wedge m \mapsto (T \to T', \lambda(x : T).e) \in FUNS_2 \end{array}}{FUNS_1 + FUNS_2 = FUNS_3} \quad \text{(CAT-FUNS)}$$

$$\frac{\begin{array}{c} CASES_3 = \{c_{(=)} \mapsto (\langle a.R\rangle, T \to T', \lambda(x : T).e) \in CASES_1 \bigsqcup CASES_2\} \cup \\ \{c_{(=)} \mapsto (\langle a.R\rangle, T \to T', \lambda(x : T).e) : \mathcal{P}(c, (\langle a.R\rangle, T \to T', \lambda(x : T).e)) \vee \mathcal{P}'(c, (\langle a.R\rangle, T \to T', \lambda(x : T).e))\} \\ \mathcal{P}(c, (\langle a.R\rangle, T \to T', \lambda(x : T).e)) = \\ c_{(=)} \mapsto (\langle a.R\rangle, T \to T', \lambda(x' : T).e') \in CASES_1 \wedge c_{(=)} \mapsto (\langle a.R\rangle, T \to T', \lambda(x : T).e) \in CASES_2 \\ \mathcal{P}'(c, (\langle a.R\rangle, T \to T', \lambda(x : T).e)) = \\ c_{(=)} \mapsto (\langle a'.R\rangle, T \to T'_1, \lambda(x_1 : T).e_1) \in CASES_1 \wedge c_{(+=)} \mapsto (\langle a.R\rangle, T \to T'_2, \lambda(x_2 : T).e_2) \in CASES_2 \wedge \\ T'_1 + T'_2 = T' \wedge \texttt{fresh}\ x \wedge e_1[x/x_1] + e_2[x/x_2] = e \end{array}}{CASES_1 + CASES_2 = CASES_3} \quad \text{(CAT-CASES)}$$

Figure 10. Linkage concatenation.

constructs. Below, we discuss the concatenation process, as well as pre- and post-processing of linkages.

Pre-processing. Before concatenation, we unfold all wildcards in `cases` constructs within linkages. Every wildcard case is replaced by the explicit set of cases it implicitly covers within the given family. 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 extra cases for the match to be exhaustive.

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, constructs that have been newly defined or extended in the derived family, and constructs in the derived family that overwrite inherited constructs. At the top level, concatenation is recursively propagated to all nested components of the linkages: collections of nested families, types, ADTs, etc. We illustrate this propagation in Figure 10 (rule `CAT-TOP`) where linkages are represented using brackets {...}. All paths referring to the parent family within inherited code are updated to refer to the derived family post-concatenation, adapting the inherited code for use with extended types.

Linkages for any nested families are recursively concatenated (`CAT-NEST`). Within a linkage, names of the nested families are mapped to their corresponding linkages (such as $\overline{A \mapsto L}$). For each nested family whose name appears only in the parent family or the extension, the mapping to its linkage is copied to the resulting linkage unchanged. This is represented as a disjoint union, $\sqcup$, 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 $\mathcal{P}(A, L)$ in the rule), it means that A is further bound in the derived family. We handle further binding in the same way as inheritance, by triggering linkage concatenation of linkages L_1 and L_2.

The concatenation rules for record types (`CAT-TYPES`), defaults (`CAT-DEFS`), and ADTs (`CAT-ADTS`) in Figure 10 follow a similar pattern. We use subscript markers (+=) and (=) to differentiate between extended and new definitions. The resulting sets of types and defaults after concatenation consist of any definitions $R_{(=)}$ inherited unchanged from the parent or newly defined in the extension (represented by the disjoint union), plus any definitions $R_{(+=)}$ that have been extended in the derived family. The concatenation operation + as defined for records simply combines the fields 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 functions (`CAT-FUNS`) and `cases` (`CAT-CASES`) is shown in Figure 10. After concatenation, the resulting set of functions $FUNS_3$ contains the disjoint union of functions (discerned by the function header, its name and type), as well as any functions with the same name and type overwritten in the extension. For `cases`, we allow overwriting when the definition in the derived family has the same name, match type, and arrow type. We also allow extension of `cases` by concatenating their bodies e_1 and e_2 and replacing the bound variables for the match context inside each with a fresh variable. We also ensure that after extension the `cases` construct does not have duplicate handlers for the same ADT constructor (omitted from figure).

Post-processing. After concatenation, we examine the resulting complete linkage and fill in any existing "holes" with default values, if they were provided by the user. Every instance of a record type must provide inputs for each record field in the type. If that is not the case, we check whether there exist defaults for the missing record fields. If the defaults exist, we fill them in. If the defaults for missing fields do not exist, the expression will not typecheck. Default handling is done

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

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

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

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

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

$$\frac{\begin{array}{c}\mathtt{K} \vdash a \rightsquigarrow L \qquad R\ \ (+)?{=}\ \overline{C_j\ T_j}\ \in L.\mathtt{ADTS} \\ \mathtt{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\} \to \{(C_j : T_j \to T'_j){*}\}) \\ \mathtt{K}; [\,] \vdash \lambda(x : \{(f_i : T_i){*}\}).\{(C_j = \lambda(y_j : T_j).e_j){*}\} : \{(f_i : T_i){*}\} \to \{(C_j : T_j \to T'_j){*}\}\end{array}}{\mathtt{K} \vdash \mathtt{WF}(\textbf{cases}\ c\ \langle a.R\rangle\ :\ \{(f_i : T_i){*}\} \to \{(C_j : T_j \to T'_j){*}\}\ \ (+)?{=}\ \lambda(x : \{(f_i : T_i){*}\}).\{(C_j = \lambda(y_j : T_j).e_j){*}\})}\ (\text{WF-CasesDef})$$

Figure 11. Well-formedness of definitions.

after concatenation, when we have access to all fields (inherited and extended) of a record type simultaneously.

Precedence of further binding. In our system, further binding takes precedence over inheritance. Other related systems with nested inheritance, such as Jx, make the same choice [18]. For example, in Figure 7, family `A2.B2` extends family `A2.B1`, and further binds family `A1.B2`. If some function f is defined in both `A2.B1` and `A1.B2`, the definition in `A1.B2` (further bound) should take precedence, since `A1.B2` is structurally more similar to `A2.B2` than `A2.B1`. Rule `CAT-NEST`, along with linkage computation rule `L-Nest` from Figure 8, ensures that is the case. When the complete linkage for `A2.B2` is computed using `L-Nest`, we concatenate the complete parent linkage L' (for `A2.B1`) with the incomplete child linkage L'' (for `A2.B2`) 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. This means that 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. The full concatenation tree for this example is included in Figure 9.

4.3 Type System

Our type system relies heavily on linkages. By delegating the heavy-duty handling of our extensible features (such as combining inherited ADT constructors with newly added constructors, combining inherited pattern match cases with new cases, and overriding of cases and functions) to linkage concatenation, we maintain a fairly straightforward type system.

Well-Formedness of Types. Our rules for well-formedness of types are available in the appendix. Well-formedness of primitives, arrow types, and record types is defined in the usual way. A path type $a.R$ is well-formed if there exists a record type definition or an ADT definition for R in the complete linkage for family path a.

Typing and Well-Formedness of Programs. Typing of programs p is shown in Figure 12 (`T-Prog`). A program p consists of family definitions and a main expression e. A program p is well-typed if every family definition is well-formed (discussed below), and the main expression e is well-typed. At this level of nesting, the linkage context K contains only one path, `prog`, which is the path to p.

Well-formedness of definitions is checked according to Figure 11, and is driven by the Persimmon program syntax. A well-formed family definition (`WF-FamDef`) has no circular inheritance – the family A may not have itself as an ancestor, which is recursively checked by computing linkages for each parent up the chain. Within a family definition, all nested family definitions, type and ADT definitions, function definitions, and cases definitions should also be well-formed. The contents of family A are checked recursively with respect to an expanded linkage context K′ which contains the path to A (`self`$(sp.A)$). We maintain the convention that the most recent path in the linkage context points to the immediate container of the definition that is being checked.

For record type definitions, we check that the type itself is well-formed, and that any default values v_i have the expected types (`WF-TypDef`). Similarly, for ADT definitions, the record type accompanying each constructor C_j must be well-formed (`WF-AdtDef`). Function definitions are well-formed (`WF-FunDef`) when the type of the function is well-formed and the function body is well-typed.

`cases` definitions are checked similarly to functions (`WF-CasesDef`), with an additional check that each `cases` construct is exhaustive with respect to the ADT definition of R in the linkage for family path a. Therefore, exhaustivity is checked for each newly appearing `cases` construct and for all extensions to existing constructs at the time of definition. The advantage of checking exhaustivity at definition time is twofold: (1) we can prevent typing errors before an incomplete definition is used, and (2) we can avoid typechecking the same constructs multiple times. However, one downside is that we cannot check exhaustivity of the *combined* components (after extension) at the time of definition. To get this, we would have to run the well-formedness check on complete linkages post-concatenation, which would result in multiple checking of components (before and after inheritance). To preserve modular typechecking, we choose to check exhaustivity at the time of definition, and then double-check at use-site. The duplicate check in our type system will not allow an application of a non-exhaustive `cases` definition to typecheck (rule `T-Match`, detailed below).

We also ensure that all types have unique names, cases and functions have unique headers, and that there are no duplicate constructor names in an ADT. These repetitive checks are omitted in Figure 11.

Typing of Expressions. We typecheck expressions as shown in Figure 12. Within typechecking, all types, functions, and `cases` definitions are retrieved from complete linkages – meaning that inherited and extended components have already been combined via linkage concatenation, and any overwritten components updated. Expressions are typed with respect to two contexts: Γ (the typing context) and K (the family path context). Below, we highlight the typing rules that best reflect our use of linkages as the extensibility engine of our system.

The rule for function selection, `T-FamFun`, retrieves the type of function $a.m$ directly from the complete linkage L for path a. Since typing of programs relies on well-formedness of family definitions (see `T-Prog`), the type of $a.m$ stored in L must be well-formed and the correct type for the definition of $a.m$. Hence, we do not need to check this again by invoking `T-Lam` from `T-FamFun`. Selection of `cases` constructs is typed similarly (`T-Cases`), as `cases` can be viewed as special, extensible functions.

An instance of a record type, $a.R(\{(f_i = e_i)*\})$, is well-typed by rule `T-Constr` if the linkage L for path a contains a definition for type R, and the inputs e_i are well-typed with respect to this

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

$$\frac{\mathrm{K} \vdash \texttt{WF}(T) \qquad \mathrm{K};(x:T,\Gamma) \vdash e : T'}{\mathrm{K};\Gamma \vdash \lambda(x:T).e : T \to T'}\ \text{(T-Lam)} \qquad \frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad m : T \to T' = \lambda(x:T).e \in L.\text{FUNS}}{\mathrm{K};\Gamma \vdash a.m : T \to T'}\ \text{(T-FamFun)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad R = \{(f_i : T_i)*\} \in L.\text{TYPES} \qquad \forall i,\ \mathrm{K};\Gamma \vdash e_i : T_i}{\mathrm{K};\Gamma \vdash a.R(\{(f_i = e_i)*\}) : a.R}\ \text{(T-Constr)}$$

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

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad c\ \langle a'.R\rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\} = \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\} \in L.\text{CASES}}{\mathrm{K};\Gamma \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}}\ \text{(T-Cases)}$$

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

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

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

Figure 12. Select typing rules for expressions and programs.

definition. The instance expression must specify an input e_i for every field f_i that appears in the definition of R; this is implied by using the same set of fields f_i in the goal and premises of the rule. If the user does not assign inputs to some fields, those fields are filled with default values (if they exist) prior to typing. This is part of post-processing of linkages after concatenation. The instance will not typecheck unless each field is assigned a user input or a default.

We type ADT instances similarly. The instance of an ADT $a.R$ specifies the constructor C used to create the instance, and a record of inputs $\{(f_k = e_k)*\}$ to instantiate the fields of C. Rule `T-ADT` ensures that constructor C exists in the definition of R (retrieved from the complete linkage L), and that all inputs are well-typed.

Finally, pattern match expressions are typed according to rule `T-Match`. The matched expression e must have some path type $a'.R$, and the complete linkage L for path a' must contain an ADT definition for R. The body of the match expression has a special shape – a `cases` selection $a.c$ applied to the match context, as introduced in Section 4.1. We ensure that both expressions within the body are well-typed. Finally, since both the ADT definition R and the type of $a.c$ refer to the same set of constructors C_j, pattern match exhaustivity is double checked at use-site.

Subtyping. 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 corresponding record type). The full rules are included in the appendix. We do not have a subtyping relation for families, due to undesired interactions between inherited and extended components [11]. For example, if family A' extends family A and is also a subtype of A, then a type instance from family A' could pass as an instance of the same type from A. This interaction may

not be type safe because the derived family might make assumptions that are not true of the super family. It also defies the family polymorphic guarantee (members of different families should not interact).

4.4 Operational Semantics

Like our type system, our operational semantics delegates the heavy lifting to linkages. Most rules follow the convention of reducing subexpressions from left to right. Function calls $a.m$ and `cases` selections $a.c$ reduce directly to their definitions retrieved from the linkage for family path a. The special shape of our match expressions means that we perform the application of the `cases` selection to the match context (within the body of the match) before we project the required case handler. Our full reduction and substitution relations are included in the appendix.

5 Formal Results

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

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

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 resulting from well-typed programs are well-formed, and well-formedness is preserved by linkage concatenation. The full proofs are available in the supplementary material.

6 Separate Compilation

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. Importantly, our compiler supports separate compilation: code compiled for a base family can be shared with any of its derived families, without needing recompilation.

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 13. 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.

```scala
import reflect.Selectable.reflectiveSelectable

object A2$B2 {
  // Types
  type X = {val x: Boolean}
  // ADTs
  sealed trait Exp
  // Defined constructors
  case class EPlus[self$$$Exp](e2: self$$$Exp, e1: self$$$Exp)
    extends Exp
  // Inherited constructors
  case class A2$B1$$Exp(inherited: A2$B1.Exp) extends Exp {
    override def toString(): String = inherited.toString()
  }
  case class A1$B2$$Exp(inherited: A1$B2.Exp) extends Exp {
    override def toString(): String = inherited.toString()
  }

  // Path interface
  trait Interface extends A2$B1.Interface with A1$B2.Interface {
    self$ =>
    // Self Named types
    type X
    // Self ADTs
    type Exp
    // Functions
    val ev: self$.Exp => Int
    val f: Int => Int
    // Cases
    def evc(matched: self$.Exp): Unit => Int
    // Translations
    def A2$B2$$Exp(from: A2$B2.Exp): Exp
  }

  // Path implementation
  object Family extends A2$B2.Interface { self$ =>
    // Self named types instantiation
    override type X = A2$B2.X
    // Self ADTs instantiation
    override type Exp = A2$B2.Exp
    // Function implementations
    override val ev: self$.Exp => Int = ev$Impl(A2.Family, self$)
    def ev$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
      self$.Exp => Int = A1$B1.Family.ev$Impl(self$1, self$)
    override val f: Int => Int = f$Impl(A2.Family, self$)
    def f$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
      Int => Int = A1$B2.Family.f$Impl(self$1, self$)
    // Cases implementations
    def evc(matched: self$.Exp): Unit => Int =
      evc$Impl(A2.Family, self$)(matched.asInstanceOf[A2$B2.Exp])
    def evc$Impl(self$1: A2.Interface, self$: A2$B2.Interface)
      (matched: A2$B2.Exp): Unit => Int =
      (unit: Unit) => matched match {
      case matched@A2$B2.EPlus(_, _) =>
        val x: A2$B2.EPlus[self$.Exp] =
          matched.asInstanceOf[A2$B2.EPlus[self$.Exp]]
        (self$.ev.asInstanceOf[self$.Exp => Int](x.e1) +
         self$.ev.asInstanceOf[self$.Exp => Int](x.e2))
      case A2$B2.A2$B1$$Exp(inherited) =>
        A2$B1.Family.evc$Impl(self$1, self$)(inherited)(unit)
      case A2$B2.A1$B2$$Exp(inherited) =>
        A1$B2.Family.evc$Impl(self$1, self$)(inherited)(unit)
    }

    // Translation function implementations
    override def A2$B2$$Exp(from: A2$B2.Exp): Exp = from
    override def A2$B1$$Exp(from: A2$B1.Exp): Exp =
      A2$B2.A2$B1$$Exp(from)
    override def A1$B1$$Exp(from: A1$B1.Exp): Exp =
      A2$B2.A2$B1$$Exp(A2$B1.Family.A1$B1$$Exp(from))
    override def A1$B2$$Exp(from: A1$B2.Exp): Exp =
      A2$B2.A1$B2$$Exp(from)
  }
}
```

Figure 13. Separate compilation translation of Persimmon code in Figure 7 to Scala code.

7 Related Work

Solutions to the Expression Problem. Our work is a solution to the expression problem [27]. The expression problem aims to extend data types *and* functionality over these data types, while maintaining separate compilation and type safety. Persimmon achieves type safe extensibility in both dimensions, and has a separate compilation translation to Scala. We discuss other solutions below.

One solution by Nystrom et al., Jx, is an object-oriented language with nested inheritance [18]. Inheritance in Jx is centered around the notion of "containers." All components of a container, including nested containers, can be inherited, preserving the relationships between components [18]. Persimmon supports nested family polymorphism, which subsumes nested inheritance. While Jx supports extensible encodings of ADTs and extensible functionality in the OO programming style, Persimmon supports extensible ADTs and pattern matching in the functional style.

J&, which builds on Jx, supports modular composition of extensions by adding nested intersection via intersection types [19]. Persimmon supports composable extensions by encoding and does not include a special type for this purpose.

Odersky and Zenger [20] rely on Scala traits and modular mixin composition to support composable extensions [20, 21]. The use of a Visitor pattern and deep mixin composition is required to

achieve extensibility of both types and functionality. In Persimmon, extensibility is built-in and does not require the use of patterns. Most constructs in Persimmon are themselves extensibility hooks, eliminating parameter clutter.

A solution by Oliveira and Cook uses object algebras (an abstraction related to Church encodings) and relies on simple generics [22]. Object algebras are powerful abstractions that can express family polymorphism. One downside is that modular composition of object algebras requires some manual set-up by defining a combinator. In Persimmon, extensions can be composed more straightforwardly via late binding of the base family.

Related to object algebras is also the "tagless final" approach, which relies on interpreters and a skillful embedding of DSLs in the host language [14]. The flexibility comes from interpreting the language; extensibility can be achieved by adding more syntactic forms or interpreters [14]. One solution using this approach exploits abstraction over families of interpreters [2]. In Persimmon, extensibility is built-in and no encoding is necessary.

Continuing the line of work started with object algebras, Zhang et al. [29] 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 parametrization, in particular self-type annotations to inject dependencies.

Another solution uses open data types and open functions scattered throughout modules, allowing the definitions to be provided at any point in the program [15]. Polymorphic variants [9] allow constructors to exist independently of types, and support open pattern matching. Persimmon achieves a similar result by making code polymorphic to the family and thus safe for reuse in derived families.

Extensible Variant Types and Pattern Matching. Gaster and Jones support extensible records and variants, emphasizing the use of rows for their construction [10]. An extension to their system allows pattern match cases to be treated as first-class, extensible values. Another related solution supports extensibility of pattern matching and composable extensions via extensible first-class cases, capitalizing on the dual relationship between polymorphic records and sums [1]. These record-centered solutions support row polymorphism for records, but not family polymorphism. Persimmon supports both family polymorphism and extensible pattern matching. In Persimmon, `cases` are not first-class – their usage is restricted to application within a match expression. This allows for translation of Persimmon to other languages with a more traditional match expression shape.

Zenger and Odersky propose an object-oriented solution for extensible algebraic data types by providing default variants for all ADTs that subsume any future extensions [28]. They encode their solution using a new design pattern for extensible visitors. Pattern matching becomes extensible by delegating computation in the default case to methods overridden in the extension. Although both solutions proactively adapt code for use with extensions, in Persimmon this happens automatically via built-in relative path types. Unlike Persimmon, this solution does not support composable extensions.

Extensible ML (EML), supports hierarchical, extensible data types and extensible functionality over those datatypes, while preserving modular typechecking [17]. While EML focuses on unifying the OO and functional programming styles to support extensibility, Persimmon brings extensibility to the functional programming style via family polymorphism.

Among the pattern match techniques evaluated by Emir et al. [5], 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`.

Syme et al. implement extensible pattern matching through the use of active patterns in *F#*, handling both partial and total decompositions [24]. Persimmon does not support partial patterns due to the conflict with pattern match exhaustivity.

`match` is an extensible language which implements extensible pattern matching for Racket using macros [26]. JMatch [13] is an extension of Java that provides modal abstraction (integration of pattern matching and iteration abstractions), where patterns are not tied to constructors. Unlike Persimmmon, these solutions do not check for exhaustivity of pattern matching.

Family Polymorphism. Family polymorphic systems can be *object-based* or *class-based*, depending on the explicit representation of families in the system. The seminal paper by Ernst [6] uses an object-based approach, in which families are expressed as enclosing *family objects*. The class attributes of the family object comprise members of the family. Because any number of object instances can exist at runtime, the type system can recognize an unbounded number of families. Other object-oriented systems, such as *vc* and Tribe, support family polymorphism through virtual classes [4, 7].

In the *class-based* approach, introduced by Igarashi et al. as a more "lightweight" version, a family is not associated with a single instance, but with the class itself [11]. Although the number of families is bounded by the number of defined classes, it allows for a more straightforward implementation as a nested class system. Families are top-level classes, and members of the family are nested classes. We have also chosen this class-based approach for our system. A follow-up work by Igarashi and Viroli introduces variant path types, which allow class-based family polymorphism to coexist with subtyping [12]. Persimmon supports subtyping for types, but not subtyping for families.

Family polymorphism can now be found in functional systems, such as Familia [30]. Familia unifies the genericity mechanisms of inheritance and parametric polymorphism, supports class-based family polymorphism, and allows for arbitrary-level nesting of classes, interfaces and types [30]. Our approach is also inspired by the use of linkages in Familia. Although both Persimmon and Familia are functional, family polymorphic systems, Persimmon supports extensible variant types and pattern matching, whereas Familia does not.

8 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 a complex type system and operational semantics. Our explicit `cases` constructs separate match cases from pattern match expressions and provide a natural mechanism for extensibility of pattern matching. Exhaustivity of pattern matching is maintained by 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. Finally, we preserve modular type checking in the presence of extended variants, and achieve separate compilation by translation to Scala.

In future work, we plan to apply our extensible design in the context of proof engineering. Consider the verification of extensible compilers, such as our example in Figure 2. To mechanize the proofs about this system, we would need a verification framework that supports nested inheritance of verified components. By extending Persimmon to support extensible proof terms, we can create a powerful framework for modular verification of nested developments.

References

[1] 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

[2] 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.

[3] 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)*.

[4] 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.

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

[6] 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

[7] 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.

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

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

[10] 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 . . ..

[11] 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

[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.

[13] 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

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

[15] 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

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

[17] 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.

[18] 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

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

[20] Martin Odersky and Matthias Zenger. 2005. Independently extensible solutions to the expression problem. ACM.

[21] Martin Odersky and Matthias Zenger. 2005. 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

[22] 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

[23] 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*.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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

[30] 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