Familia: Unifying Interfaces, Type Classes, and Family Polymorphism

YIZHOU ZHANG, Cornell University, USA
ANDREW C. MYERS, Cornell University, USA

Parametric polymorphism and inheritance are both important, extensively explored language mechanisms for providing code reuse and extensibility. But harmoniously integrating these apparently distinct mechanisms—and powerful recent forms of them, including type classes and family polymorphism—in a single language remains an elusive goal. In this paper, we show that a deep unification can be achieved by generalizing the semantics of interfaces and classes. The payoff is a significant increase in expressive power with little increase in programmer-visible complexity. Salient features of the new programming language include retroactive constraint modeling, underpinning both object-oriented programming and generic programming, and module-level inheritance with further-binding, allowing family polymorphism to be deployed at large scale. The resulting mechanism is syntactically light, and the more advanced features are transparent to the novice programmer. We describe the design of a programming language that incorporates this mechanism; using a core calculus, we show that the type system is sound. We demonstrate that this language is highly expressive by illustrating how to use it to implement highly extensible software and by showing that it can not only concisely model state-of-the-art features for code reuse, but also go beyond them.

CCS Concepts: • Theory of computation → Type theory; Abstraction; Object oriented constructs; • Software and its engineering → Polymorphism; Inheritance; Semantics; Classes and objects; Modules / packages; Constraints; Object oriented languages;

Additional Key Words and Phrases: Familia, language design, extensibility, genericity, type-safety, type classes, family polymorphism

ACM Reference format:

1 INTRODUCTION

It is futile to do with more things that which can be done with fewer.

—William of Ockham

Types help programmers write correct code, but they also introduce rigidity that can interfere with reuse. In statically typed languages, mechanisms for polymorphism recover needed flexibility about the types that code operates over.

Authors’ email addresses: yizhou@cs.cornell.edu and andru@cs.cornell.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Association for Computing Machinery.
2475-1421/2017/10-ART70
https://doi.org/10.1145/3133894
Subtype polymorphism [Cardelli 1988] and inheritance [Cook et al. 1990] are polymorphism mechanisms that have contributed to the wide adoption of modern object-oriented (OO) languages like Java. They make types and implementations open to future type-safe extensions, and thus increase code extensibility and reuse.

Parametric polymorphism offers a quite different approach: explicitly parameterizing code over types and modules it mentions [Liskov et al. 1977; Milner 1978; MacQueen 1984]. It has dominated in functional languages but is also present in modern OO languages. Parametric polymorphism becomes even more powerful with the addition of type classes [Wadler and Blott 1989], which allow existing types to be retroactively adapted to the requirements of generic code.

Harmoniously integrating these two kinds of polymorphism has proved challenging. The success of type classes in Haskell, together with the awkwardness of using F-bounded constraints [Canning et al. 1989] for generic programming, has inspired recent efforts to integrate type classes into OO languages [Siek and Lumsdaine 2011; Wehr and Thieman 2011; Zhang et al. 2015b]. However, type classes and instances for those type classes burden already feature-rich languages with entirely new kinds of interfaces and implementations. The difficulty of adopting concepts in C++ [Stroustrup 2009] suggests that the resulting languages may seem too complex.

Meanwhile, work on object-oriented inheritance has increased expressive power by allowing inheritance to operate at the level of families of related classes and types [Madsen and Møller-Pedersen 1989; Thorup 1997; Madsen 1999; Ernst 1999; Nystrom et al. 2004; Aracic et al. 2006; Nystrom et al. 2006; Ernst et al. 2006; Clarke et al. 2007; Qi and Myers 2010]. Such family polymorphism, including virtual types and virtual classes, supports coordinated, type-safe extensions to related types and classes contained within a larger module. These features have also inspired the addition of associated types [Chakravarty et al. 2005b] to type classes. Associated types are adopted by recent languages such as Rust [Rust 2014] and Swift [Swift 2014]. However, the lack of family-level inheritance limits the expressive power of associated types in these languages.

Combining all these desirable features in one programming language has not been done previously, perhaps because it threatens to confront programmers with a high degree of surface complexity. Our contribution is a lightweight unification of these different forms of polymorphism, offering increased expressive power with low apparent complexity. This unified polymorphism mechanism is embodied in a proposed Java-like language that we call Familia.

The key insight is that a lightweight presentation of the increased expressive power can be achieved by using a single interface mechanism to express both data abstraction and type constraints, by using classes as their implementations, and by using classes (and interfaces) as modules. Both interfaces and classes can be extended. The expressive power offered by previous polymorphism mechanisms, including flexible adaptation and family polymorphism, falls out naturally. Specifically, this paper makes the following contributions:

- We show how to unite object-oriented polymorphism and parametric polymorphism by generalizing existing notions of interfaces, classes, and method calls (Sections 3 and 4). The extensibility of objects and the adaptive power of type classes both follow from this reinterpretation.

- We further show how to naturally integrate an expressive form of family polymorphism. The design accommodates features found in previous family-polymorphism mechanisms (including associated types and nested inheritance) in the above setting of generalized classes and interfaces, and goes beyond them by offering new expressive power. We present a case study of using Familia to implement a highly reusable program analysis framework (Section 5).

- We capture the new language mechanisms formally by introducing a core language, Featherweight Familia, and we establish the soundness of its type system (Section 6).
We show the power of the unified polymorphism mechanism by comparing Familia with various prior languages designed for software extensibility (Section 7).

2 BACKGROUND

Our goal is a lightweight, expressive unification of the state of the art in genericity mechanisms. A variety of complementary genericity mechanisms have been developed, with seemingly quite different characteristics.

**Genericity via inheritance.** Object-oriented languages permit a given interface to be implemented in multiple ways, making clients of that interface generic with respect to future implementations. Hence, we call this a form of *implementation genericity*. Inheritance extends the power of implementation genericity by allowing the code of a class to be generic with respect to implementation changes in future subclasses; method definitions are *late-bound*. While the type-theoretic essence of class inheritance is parameterization [Cook and Palsberg 1989], encoding inheritance in this way is more verbose and less intuitive [Black et al. 2016].

Family polymorphism [Ernst 2001] extends the expressive power of inheritance by allowing late binding of the meaning of types and classes declared within a containing class, supporting the design of highly extensible and composable software [Nystrom et al. 2006]. Virtual types [Thorup 1997; Odersky et al. 2003] and associated types [Järvi et al. 2005; Chakravarty et al. 2005b] allow the meaning of a type identifier to be provided by subclasses; with virtual classes as introduced by Beta [Madsen and Møller-Pedersen 1989; Ernst 2001], the code of a nested class is also generic with respect to classes it is nested within. The outer class can then be subclassed to override the behavior and structure of the entire family of related classes and types in a coordinated and type-safe way. Classes and types become members of an object of the family class. The late binding of type names means that all type names implicitly become hooks for later extension, without cluttering the code with a possibly large number of explicit type parameters.

There are two approaches to family polymorphism; in the nomenclature of Clarke et al. [2007], the original *object family* approach of Beta treats nested classes as attributes of objects of the family class [Madsen and Møller-Pedersen 1989; Ernst 2001; Aracic et al. 2006], whereas in the *class family* approach of Concord [Jolly et al. 2004], Jx and J& [Nystrom et al. 2004, 2006], and “FJ” [Igarashi and Viroli 2007] nested classes and types are attributes of the family classes directly. The approaches have even been combined by work on Tribe [Clarke et al. 2007]. Familia follows Jx by providing nested inheritance [Nystrom et al. 2004], a class family mechanism that allows both further binding (specialization of nested classes) at arbitrary depth in the class nesting structure, and also inheritance across families.

To see how support for coordinated changes can be useful, suppose we are building a compiler for a programming language called Saladx, which extends a previous language called Salad. The Salad compiler defines data structures (that is, types) and algorithms that operate on these types. We would like to reuse the Salad compiler code in a modular way, without modification. Figure 1 sketches how this can be done in a modular, type-safe way using nested inheritance.

The original compiler defines abstract syntax tree (AST) nodes such as `Node` and `Stmt`. The extended compiler defines a new module `Saladx` that inherits as a family from the original `Salad` module. The new module adds support for a new type of AST node, `UnaryExpr`, by adding a new class...
definition. Saladx also further binds the class `Node` to add a new method `constFold` that performs constant folding. Importantly, the rest of Salad. `Node` does not need to be restated. Nor does any code need to be written for `Saladx.Stmt`; this class implicitly exists in the `Saladx` module and inherits `constFold` from the new version of `Node`. References in the original Salad code to names like `Node` and `Stmt` now refer in the Saladx compiler to the Saladx versions of these classes. The Salad code is highly extensible without being explicitly parameterized.

By contrast, the conventional OO approach could extend individual classes and types from Salad with new behavior. However, each individually extended class could not access the others’ extended behavior (such as `constFold`) in a type-safe way. Alternatively, extensibility could be implemented for Salad by cluttering the code with many explicit type parameters.

**Genericity via parametric polymorphism.** Parametric polymorphism, often simply called generics, provides a more widely known and complementary form of genericity in which code is explicitly parameterized with respect to the types or modules of data it manipulates. Whereas implementation genericity makes client code and superclass code generic with respect to future implementations, parametric polymorphism makes implementations generic with respect to future clients. Constrained parametric polymorphism [Liskov et al. 1977] ensures that generic code can be instantiated only on types meeting a constraint. These constraints act effectively as a second kind of interface.

Haskell’s type classes [Wadler and Blott 1989] manifest these interfaces as named constraints to which programmers can explicitly adapt existing types. By contrast, most OO languages (e.g., Java and C#) use subtyping to express constraints on type parameters. Subtyping constraints are rigid: they express binary methods in an awkward manner, and more crucially, it is typically impossible to retroactively adapt types to satisfy the subtyping requirement. The rigidity of subtyping constraints has led to new OO languages that support type classes [Siek and Lumsdaine 2011; Wehr and Thiemann 2011; Zhang et al. 2015b].

**Combining genericity mechanisms.** Genericity mechanisms are motivated by a real need for expressive power. Both family polymorphism and type classes can be viewed as reactions to the classic expression problem [Wadler et al. 1998] on the well-known difficulty of extending both data types and the operations on them in a modular, type-safe way [Reynolds 1975]. However, the approaches are complementary: type classes do not also provide the scalable extensibility [Nystrom et al. 2004] offered by family polymorphism, whereas family polymorphism lacks the flexible adaptation offered by type classes. Despite becoming popular among recent languages that incorporate type classes [Rust 2014; Swift 2014], associated types do not provide the degree of extensibility offered by an expressive family-polymorphism mechanism.

On the other hand, data abstraction is concerned with separating public interfaces from how they are implemented so that the implementation can be changed freely without affecting the using code. Implementations are defined in terms of a representation that is hidden from clients of the interface. Abstract data types, object interfaces, and type classes can all provide data abstraction [Cook 2009]. Genericity mechanisms such as inheritance and parametric polymorphism are not essential to data abstraction. However, they add significant expressive power to data abstraction.

Languages that combine multiple forms of polymorphism tend to duplicate data abstraction mechanisms. For example, recent OO languages incorporate the expressive power of type classes by adding new language structures above and beyond the standard OO concepts like interfaces and classes [Siek and Lumsdaine 2011; Wehr and Thiemann 2011; Zhang et al. 2015b]. Unfortunately, a programming language that provides data abstraction in more than one way is likely to introduce feature redundancy and threatens to confront the programmer with added surface complexity. Even
interface Eq(T) {
   boolean T.equals(T);
}

interface Hashable extends Eq {
   int hashCode();
}

interface PartialOrd extends Eq {
   boolean leq(This);
}

interface Ord extends PartialOrd {
   int compare(This);
}

Figure 2. Four interfaces with single representation types. Eq explicitly names its representation type T; the others leave it implicit as This. The receiver types of the interface methods are the representation types.

for Haskell, it has been argued that type classes introduced duplication of functionality [Devriese and Piessens 2011], and that the possibility of approaching the same task in multiple ways created confusion [Palmer 2010].

Our contribution is a clean way to combine data abstraction and these disparate and powerful polymorphism mechanisms in a compact package. As a result, programmers obtain the expressive power they need for a wide range of software design challenges, without confronting the linguistic complexity that would result from a naive combination of all the mechanisms.

3 UNIFYING OBJECT-ORIENTED INTERFACES AND TYPE CLASSES

Both object-oriented interfaces and type classes are important, but having both in a language can lead to confusion and duplication. Fortunately, both can be supported by a single, unified interface mechanism, offering an economy of concepts.

We unify interfaces with type classes by decoupling the representation type of an object-oriented interface from its object type. A representation type is an underlying type used to implement the interface; the implementations of interface methods operate on these representation types. An object type, on the other hand, specifies the externally visible operations on an object of the interface.

For example, an interface Eq describing the ability of a type T to be compared for equality can be written as shown in Figure 2.1 This interface declares a single representation type T (in parentheses after the interface name Eq); the receiver of method equals hence has this representation type. Each implementation of this interface chooses some concrete type as the representation type.

As convenient syntactic sugar, an interface with a single representation type may omit its declaration, implicitly declaring a single representation type named This. In this usage, all non-static methods declared by the interface have implicit receiver type This. In Figure 2, the other three interfaces all exploit this sugar.

An interface may also declare ordinary type parameters for generic programming, grouped in square brackets to distinguish them from representation type parameters. For example, a generic set interface might be declared as shown in Figure 3a, where the interface Set has an explicit type parameter E representing its elements. In the figure, the omitted representation type of Set (i.e., This) is also the implicit representation type of Collection[E], the interface being extended.

Using interfaces to constrain types. Interfaces can be used as type classes: that is, as constraints on types. In Figure 3a, Set has a where-clause where Eq(E), which constrains the choice of types for E to those which satisfy the interface Eq and that therefore support equality. A where-clause may have several such constraints, each constraining a type parameter by instantiating an interface using that type (E in this example) as the representation type. Hence, we also refer to representation types as constraint parameters.

As syntactic sugar, a where-clause may be placed outside the brackets containing type parameters (e.g., line 2 of Figure 3b). If kept inside the brackets, the parameters to the constraint may be omitted,

1Except as noted, Familia follows the syntactic and semantic conventions of Java [Gosling et al. 2005].
interface Set[E where Eq(E)]
extends Collection[E] {
    int size();
    boolean contains(E);
    Self add(E);
    Self remove(E);
    Self addAll(Set[E]);
    ...}

1 interface SortedSet[E]
2 extends Set[E] where Ord(E) {
3     E max() throws SetEmpty;
4     E min() throws SetEmpty;
5     Self subset(E, E);
6     ...
7 }

(a) Interface Set is parameterized by a type parameter and a where-clause constraint.
(b) Interface SortedSet extends Set. Its where-clause constraint Ord(E) entails Eq(E).

Figure 3. Interfaces Set and SortedSet.

defaulting to the preceding type parameter(s), as in line 1 of Figures 6a and 6b. A where-clause constraint can optionally be named (e.g., line 20 of Figure 5).

Using the object type. Each interface also defines an object type that has the same name as the interface. Using an interface as an object type corresponds to the typical use of interfaces in OO languages. In this case, the interface hides its representation type from the client. For example, in the variable declaration “Hashable x;”, the representation type is an unknown type T on which the constraint Hashable(T) holds. The programmer can make the method call x.hashCode() in the standard, object-oriented way. Thus, the interface Hashable serves both as a type class that is a constraint on types and as an ordinary object type.

From a type-theoretic viewpoint, object types are existential types, as in some prior object encodings [Bruce et al. 1999]. A method call on an object (e.g., x.hashCode()) implicitly unpacks the existentially typed receiver. This unpacking is made explicit in the core language of Section 6.

Subtype polymorphism and constraint entailment. Subtype polymorphism is an essential feature of statically typed OO languages. As Figures 2 and 3 show, interfaces can be extended in Familia. The declaration extends Collection[E] in the definition of interface Set introduces a subtype relationship between Set[E] and Collection[E]. An interface definition can extend multiple interfaces.

Such subtype relationships are higher-order [Pierce and Steffen 1997], in the sense that interfaces in Familia can be viewed as type operators that accept a representation type. When the interfaces are used as object types, this higher-order subtyping relation becomes the familiar, first-order subtyping relation between object types. When the interfaces are used to constrain types, this higher-order subtyping relation manifests in constraint entailment, a relation between constraints on types [Wehr and Thiemann 2011; Zhang et al. 2015b]. For example, consider the instantiation of Set on line 2 of SortedSet in Figure 3b. The type Set can only be instantiated on a type T that satisfies the constraint Eq(T); here, because Ord (transitively) extends Eq, constraint Ord(E) being satisfied entails Eq(E) being satisfied.

A second form of constraint entailment concerns varying the representation type, rather than the interface, when the interface is contravariant in its representation type. For example, all interfaces in Figures 2 and 3 are contravariant, because they do not use their representation types in covariant or invariant positions. Because Eq is contravariant, it is safe to use a class that implements Eq(Set[E]) to satisfy the constraint Eq(SortedSet[E]). Figure 4 shows an example of an invariant interface, LowerLattice, which uses this on line 2 both covariantly (as a return type) and contravariantly (as...
interface LowerLattice extends PartialOrd {
  This This.meet(This); // covariant and contravariant uses of This
  static This top(); // a static method
  static This meetAll(Iterable[This] c) {
    This glb = top();
    for (This elem : c) { glb = glb.meet(elem); }
    return glb;
  } // a static method with a default implementation
}

Figure 4. An invariant interface for a lower semilattice.

receiver and argument types). Familia infers interfaces to be either contravariant or invariant in their constraint parameters, with most interfaces being contravariant. A constraint parameter that is inferred contravariant may also be explicitly annotated as invariant. Covariance and bivariance in constraint parameters are not supported because these forms of variance do not seem useful.

Static and default methods in interfaces. Interfaces may also declare static methods that do not expect a receiver. For example, consider the interface LowerLattice in Figure 4. It describes a bounded lower semilattice, with its representation type This as the carrier set of the semilattice. Therefore, LowerLattice extends PartialOrd and additionally declares a binary method meet() and a static method top() that returns the greatest element.

Interfaces can also provide default implementations for methods. Method meetAll in LowerLattice computes the greatest lower bound of a collection of elements. However, an implementation of LowerLattice can override this default with its own code for meetAll.

The current interface Self. In Familia, all interfaces have access to a Self interface that precisely characterizes the current interface in the presence of inheritance. For example, in interface Set (Figure 3a), Self is used as the return type of the addAll method, meaning that the return type varies with the interface the method is enclosed within: in interface Set, the return type is understood as Set[E]; when method addAll is inherited into interface SortedSet, the return type is understood as SortedSet[E]. Hence, adding all elements of a (possibly unsorted) set into a sorted set is known statically to produce a sorted set:

```
SortedSet[E] ss1 = …;
Set[E] s = …;
SortedSet[E] ss2 = ss1.addAll(s);
```

This and Self are implicit parameters of an interface, playing different roles. The parameter This stands for the representation type, which is instantiated by the implementation of an interface with the type of its representation. On the other hand, the parameter Self stands for the current interface, and its meaning is refined by interfaces that inherit from the current interface. Section 4.4 further explores the roles of This and Self.

Interfaces with multiple representation types. An interface can act as a multiparameter type class if it declares multiple representation types—that is, if it has multiple constraint parameters. As an example, the Graph interface (lines 5–11 in Figure 5) constrains both Vertex and Edge types. Note that the implicit constraint parameter of the superinterface Hashable is used explicitly here. As seen on lines 7–10, when there are multiple constrained types, the receiver type of each defined operation must be given explicitly. Unlike interfaces with a single representation type, interfaces with multiple representation types do not define an object type.
graphs
module graphs;
static List[List[V]] findSCCs[V,E](List[V] vertices) where Graph(V,E) {
   ... new postOrdIter[V,E](v) ... new postOrdIter[V,E with transpose[V,E]](v) ...
} // Implements Kosaraju’s algorithm for finding strongly connected components

graphs.Graph
interface Graph(Vertex,Edge)
extends Hashable(Vertex) {
   Vertex Edge.source();
   Vertex Edge.sink();
   Iterable[Edge] Vertex.outgoingEdges();
   Iterable[Edge] Vertex.outgoingEdges();
}

graphs.postOrdIter
class postOrdIter[V,E] for Iterator[V] where Graph(V,E) {
   postOrdIter(V root) { ... }
   V next() throws NoMoreElement { ... }
   ...
}

graphs.transpose
class transpose[Vertex,Edge] for Graph(Vertex,Edge) {
   for Graph(Vertex,Edge) g {
      Vertex Edge.source()
      { return this.(g.sink)(); }
      Vertex Edge.sink()
      { return this.(g.source)(); }
      Iterable[Edge] Vertex.outgoingEdges()
      { ... }
      Iterable[Edge] Vertex.incomingEdges()
      { ... }
      int Vertex.hashCode()
      { return this.(g.hashCode)(); }
   }
}

Figure 5. A generic graph module. Interface Graph has two constraint parameters, so the method receiver types in interface Graph (and also its implementing class transpose) cannot be omitted. The code in this graph (except for lines 5–11) is discussed in Section 4.

4 UNIFYING OO CLASSES AND TYPE-CLASS IMPLEMENTATIONS

All previous languages that integrate type classes into the OO setting draw a distinction between classes and implementations of type classes. Confusingly, different languages use different terminology to describe type-class implementations. Haskell has “instances”, the various C++ proposals have “concept maps” [Stroustrup 2009], JavaGI [Wehr and Thiemann 2011] has “implementations”, and Genus [Zhang et al. 2015b] has “models”.

Familia avoids unnecessary duplication and terminology by unifying classes with type-class implementations. Classes establish the ability of an underlying representation to satisfy the requirements of an interface. The representation may be a collection of fields, in the usual OO style. Alternatively, unlike in OO style, the representation can be any other type that is to be retroactively adapted to the desired interface.

For example, to implement the interface Set (Section 3), we can define the class hashset shown in Figure 6a. Class hashset implicitly instantiates the representation type This of its interface Set[E] as a record type comprising its field types (i.e., {E[] table; int size}). Since this denotes the receiver and the receiver has this representation type, the field access on line 5 type-checks. The receiver type of a class method is usually omitted when the class has a single representation type.

Classes are not types. If classes were types, code using classes would be less extensible because any extension would be forced to choose the representation type in a compatible way. We give classes lowercase names to emphasize that they are more like terms. A class can be used via its constructors; for example, new hashset[E]() produces a new object of type Set[E].
A distinguishing feature of Familia is that a class can also instantiate its representation type explicitly, effectively adapting an existing type or types to an interface. Suppose we already had an interface `Map` and wanted to implement `Set` in terms of `Map`. As shown in Figure 6b, this adaptation can be achieved by defining a class that instantiates the representation type of `Set[E]` as `Map[E, ?]`. Class `mapset` implements the `Set` operations by redirecting them to corresponding methods of `Map`. Note that the value type of the map does not matter, hence the wildcard `?`. Because expression `this` has type `Map[E, ?]` in class `mapset`, the method call `this.containsKey(e)` on line 4 type-checks, assuming `Map` defines such a method.

A class like `mapset` has by default a single-argument constructor that expects an argument of the representation type. So an object `x` of type `Map[K, V]` can be used to construct a set through the expression `new mapset[K](x)`, an expression with type `Set[K]`. It is also possible to define other constructors to initialize the class’ representation.

Classes can be extended via inheritance. A subclass can choose to implement an interface that is a subtype of the superclass interface, but cannot alter the representation type to be a subtype, which would be unsound. The fact that a subclass can add extra fields is justified by treating the representation type of a class with fields as a nested type (Section 5.2).

4.1 Classes as Witnesses to Constraint Satisfaction

The ability of classes to adapt types to interfaces makes generic programming more expressive and improves checking that it is being used correctly. In particular, the Familia type system keeps track of which class is used to satisfy each type constraint. For example, suppose we want sets of strings that are unique up to case-insensitivity; we would like a `Set[String]` where string equality is defined in a case-insensitive way. Because interface `String` has an `equals` method

\[
\text{interface String} \{ \text{ boolean equals(String); } \} \]

it automatically structurally satisfies the constraint `Eq(String)` that is required to instantiate `Set`. However, it satisfies that interface in the wrong, case-sensitive way. We solve this problem in Familia by defining a class that rebinds the necessary methods:

```java
class cihash for Hashable(String) {
    boolean equals(String s) { return equalsIgnoreCase(s); }
    int hashCode() { return toLowerCase().hashCode(); }
}
```

```java
Set[String with cihash] s1 = new hashset[String with cihash]();
Set[String] s2 = s1; // illegal
```

Figure 6. Two implementations of the `Set` interface using different representations.
Notice that the types in this example keep track of the class being used to satisfy the constraint, a feature adopted from Genus [Zhang et al. 2015b]: a Set[String], which uses String to define equality, cannot be confused with a Set[String with cihash]. Such a confusion might be dangerous because the implementation of Set might rely on the property that two sets use the same notion of equality.

The type Set[String] does not explicitly specify a class, so to witness the constraint Eq(String) for it, Familia infers a default class, which in this case is a natural class that Familia automatically generates because String structurally conforms to the constraint Eq. The natural class has an equals method that conforms to that required by Eq(String), and its implementation simply calls through to the underlying method String.equals. We call these “natural classes” by analogy with natural models [Zhang et al. 2015b].

The natural class can also be named explicitly using the name String, so Set[String] is actually a shorthand for Set[String with String]. However, the natural class denoted by String is different from the String type.

We’ve seen that a class can be used both to construct objects and to witness satisfaction of where-clause constraints. In fact, even object construction is really another case of using a class to witness satisfaction of a constraint. For example, creating a Set[E] object needs both a value of some representation type T and a class that satisfies the constraint Set[E](T). The job of a class constructor is to use its arguments to create a value of the representation type.

4.2 Classes as Dispatchers
Like other OO languages, Familia uses classes to obtain dispatch information. Unlike previous OO languages, Familia allows methods to be dispatched using classes that are not the receiver object’s class.

The general form of a method call is

\[ e_0.(d.m)(e_1, \ldots, e_n) \]

where \( e_0 \) is the receiver, class \( d \) is the dispatcher, and \( e_1, \ldots, e_n \) are the ordinary method arguments. Dispatcher \( d \) provides the method \( m \) being invoked; the receiver \( e_0 \) must have the same type as the representation type of \( d \).

This generalized notion of method invocation adds valuable expressive power. For an example, return to class transpose in Figure 5. On line 20, its where-clause constraint, named \( g \), denotes some class for Graph(Vertex,Edge). Class transpose implements the transpose of the graph defined by \( g \) by reversing all edge orientations in \( g \); for example, method call this.(g.sink)() on line 22 uses class \( g \) to find the sink of a vertex and returns it as the source. Note that the transposed graph is implemented without creating a new data structure. On lines 2–4, the method findSCCs() demonstrates one use for the transposed graph. It finds strongly connected components via Kosaraju’s algorithm [Aho et al. 1983], which performs two postorder traversals, one on the original graph and one on the transposed graph.

Dispatcher classes can usually be elided in method calls—Familia infers the dispatcher for an ordinary method call of form \( e_0.m(e_1, \ldots, e_n) \)—offering convenience in the common cases where there is no ambiguity about which dispatcher to use. This inference process handles method invocation for both object-oriented polymorphism and constrained parametric polymorphism.

In the common case corresponding to object-oriented polymorphism, this ordinary method call represents finding a method in the dispatch information from \( e_0 \)’s own class. For example, assuming \( s1 \) and \( s2 \) have type String, the method call \( s1.equals(s2) \) is syntactic sugar for using the natural class implicitly generated for String as the dispatcher: it means \( s1.(String.equals)(s2) \). The natural
class setPO[E where Eq] for PartialOrd(Set[E]) {
    boolean Set[E].leq(Set[E] that) { return this.containsAll(that); } // base implementation
    boolean SortedSet.leq(SortedSet that) { ... } // specialization
    ...
}

Figure 7. The leq methods in class setPO are multimethods. The second leq method offers an asymptotically more efficient implementation for two sets sorted using the same order.

class generated for an interface I actually implements the constraint I(I). So the equals method in the natural class for String must have receiver type String and argument type String. Hence the expanded method call above type-checks.

In another common case corresponding to constrained parametric polymorphism, ordinary method call syntax may be employed by generic code to invoke operations promised by constraints on type parameters. For example, consider the method call e.hashCode() on line 7 in class hashset (Figure 6a). Familia infers the dispatcher to be the class passed in to satisfy the where-clause constraint Hashable(E). So if the programmer named the constraint as in “where Hashable(E) h”, the desugared method call would be e.(h.hashCode)().

The generalized form of method calls provides both static and dynamic dispatch. While the dispatcher class is chosen statically, the actual method code to run is chosen dynamically from the dispatcher class. It is easy to see this when the dispatcher is a natural class; the natural class uses the receiver’s own class to dispatch the method call. An explicit class can also provide specialized behavior for subtypes of the declared representation type; all such methods are dispatched dynamically based on the run-time type of the receiver.

In fact, class methods are actually multimethods that dispatch on all arguments whose corresponding types in the interface signature are This. Since Familia unifies classes with type-class implementations, the semantics of multimethods in Familia is the same as model multimethods in Genus [Zhang et al. 2015b]. For example, class setPO in Figure 7 implements a partial ordering for Set based on set containment. It has two leq methods, one for the base case, and the other for two sorted sets. Notice that in the second leq method, the receiver and the parameter are guaranteed to have the same ordering, because both occurrences of SortedSet indicate a subtype of the same Set[E with eq] type, assuming the where-clause constraint is named eq. Hence, the second leq method can be implemented in an asymptotically more efficient way. When class setPO is used to dispatch a method call to leq, the most specific version is chosen based on the run-time types of the receiver and the argument.

4.3 Inferring Default Classes

As mentioned in Sections 4.1 and 4.2, Familia can infer default classes both for elided with clauses and for elided dispatchers. It does so based on how classes are enabled as potential default choices, similarly to how models are enabled in Genus [Zhang et al. 2015b]. If only one enabled class works in the with clause or as the dispatcher, that class is chosen as the default class. Otherwise, Familia requires the class to be specified explicitly.

Classes can be enabled in four ways: (1) Types automatically generate and enable natural classes. (2) A where-clause constraint enables a class within its scope. (3) A use-statement enables a class in the scope where the use-statement resides; for example, the statement “use mapset;” enables this class as a way to adapt Map[E,?] to Set[E]. (4) A class is enabled within its own definition; this enclosing class can be accessed via the keyword self.
For example, consider the method call on line 6 in class mapset (Figure 6b). If the dispatcher were elided, two classes would be enabled as potential dispatcher choices—one, the natural class generated for the receiver’s type (i.e., \(\text{Map}[\text{E},?]\), which has a \(\text{size}()\) method), and the other, the enclosing class mapset, which defines a \(\text{size}()\) method with a compatible receiver type. Because of this ambiguity, Familia requires the dispatcher to be specified explicitly.

As another example, consider the method call \(\text{glb}\).\text{meet}(\text{elem})\) on line 6 in Figure 4. An enclosing class in this case is the class that implements the \text{LowerLattice} interface (this class inherits the method definition that contains this method call), and only this class is enabled as a potential dispatcher for the call. So Familia desugars the method call as \(\text{glb}.(\text{self}\.\text{meet})(\text{elem})\).

4.4 Self, This, self, and this

As discussed in Section 3, an interface definition has access to both the Self interface and the This type: Self is the current interface, while This is the type of the underlying representation of the current interface. Analogously, a class definition (as well as a non-static default method in an interface) has access to both the self class and the this term: self denotes the current class, while this denotes the underlying representation of the current class (or equivalently, the receiver). A class definition also has access to a Self interface that denotes the interface of the current class self. Hence, class self witnesses the constraint Self(\(T\)) where \(T\) is the type of this.

Although they sound similar, Self (or self) and This (or this) serve different purposes. Both Self and self are late-bound: in the presence of inheritance, their interpretation varies with the current interface or class. In this sense, Self and self provide a typed account of interface and class extensibility. Section 5 shows how the self class is further generalized to support type-safe extensibility at larger scale. On the other hand, the representation type This and the representation this provide encapsulation—objects hide their representations and object types hide their representation types—and adaptation—classes adapt their representations to interfaces.

Ignoring nesting (Section 5), objects in Familia are closest from a type-theoretic viewpoint to the denotational interpretation of objects by Bruce [1994], who gives a denotation of an object type using two special type variables: the first represents the type of an object as viewed from the inside, and the second, the type of the object once it has been packed into an existential. These type variables roughly correspond to This and Self in Familia. This denotational semantics is intended as a formal model for OO languages, but no later language distinguishes between these two types. Familia shows that by embracing this distinction in the surface language, the same underlying model can express both object types and type classes.

4.5 Adaptive Use-Site Genericity

Familia further extends the adaptive power of interfaces to express use-site genericity ala Genus [Zhang et al. 2015b]. The adaptive power arises from the duality between use-site genericity and definition-site genericity (i.e., parametric polymorphism), which respectively correspond to existential and universal type quantification. Because of this adaptive power, we call this “use-site genericity” rather than “use-site variance”, which only concerns subtyping [Torgersen et al. 2004].

Java uses wildcards and subtyping constraints to express use-site variance. For example, the type \(\text{List}\left[{\text{Set}\left[{\text{E}}\right]}\right]\) extends Set\(\left[{\text{E}}\right]\) describes all lists whose element type is a subtype of Set\(\left[{\text{E}}\right]\). The corresponding type in Familia is List\(\left[{\text{out Set}}[{\text{E}}]\right]\), which is sugar for the explicit existential type \(\left[\text{some} \ T \ \text{where} \ \text{Set}[{\text{E}}](\ T)\right]\text{List}\left[{\text{T}}\right]\), where Set\(\left[{\text{E}}\right]\) is used to constrain the unknown type \(T\), and the leading brackets denote constrained existential quantification. Therefore, one can assign a List\(\left[{\text{SortedSet}}[{\text{E}}]\right]\) to a List\(\left[{\text{out Set}}[{\text{E}}]\right]\) because the natural class generated for SortedSet\(\left[{\text{E}}\right]\) satisfies the constraint Set\(\left[{\text{E}}\right](\text{SortedSet}[\text{E}]). More interestingly, one can assign a List\(\left[{\text{Map}}[{\text{E}},?]\right]\) to...
a List[<Any Set[E]>] in a context in which class mapset (Figure 6b) is enabled. Note that this adaptation is asymptotically more efficient than with the Adapter pattern: when assigning a List[Map[<E, Any>]] into a List[<Any Set[E]>], the resulting list is represented by a list of maps and a class that adapts Map to Set, rather than by wrappers that inefficiently wrap each individual map into a set.

5 EVOLVING FAMILIES OF CLASSES AND INTERFACES

Thus far we have seen how to unify OO classes and interfaces with type classes and their implementations. However, the real payoff comes from further unifying these mechanisms with family polymorphism, to support coordinated changes to related classes and types contained within a larger module.

5.1 Class-Based Family Polymorphism in Familia: an Overview

As in many OO languages, Familia’s module mechanism is based on nesting: classes and interfaces are modules that can contain classes, interfaces, and types. Familia has nesting both via classes and via a pure module construct that is analogous to packages in Java or namespaces in C++. Apart from being able to span multiple compilation units, such a module is essentially a degenerate class that has no instances and does not implement an interface. Hence, both classes and modules define families containing their components. Familia interfaces can also contain nested components; a class inherits the nested components from its interfaces. Since nesting is allowed to arbitrary depth, a nested component may be part of multiple families at various levels.

Unlike most OO languages, Familia allows not only classes but all modules to be extended via inheritance. Following C++ convention [Stroustrup 1987], we call the module being extended the base module and the extending module, the derived module. Hence, superclass and base class are synonyms, as are subclass and derived class. We also slightly abuse the terminology “module” to mean not only the module construct but all families that contain nested components.

When a module is inherited, all components of the base module—including nested modules, classes, interfaces, types, and methods—are inherited into the derived module; the inherited code is polymorphic with respect to a family it is nested within. Further, the derived module may override the nested components. In this sense, names of components nested inside a module are implicit parameters declared by their families.

Example: dataflow analysis. As an example where coordinated changes to a module are useful, consider the problem of developing an extensible framework for dataflow analysis. A dataflow analysis can be characterized as a four-tuple \((G, I, L, F_n)\) [Aho et al. 2006]: the direction \(G\) that items flow on the control-flow graph (CFG), the set \(I\) of items being propagated, the operations \(\sqcap\) and \(\sqcup\) of the semilattice \(L\) formed by the items, and the transfer functions \(F_n\) associated with each type of AST node.

We would like to be able to define a generic dataflow analysis framework that leaves the four parameters either unspecified or partially specified, so that a specific analysis task (such as live variable analysis) can be obtained by instantiating or specializing the parameters. This can be achieved using family polymorphism, but does not work with conventional OO languages since they do not support coordinated changes, as discussed in Section 2.

Figure 8 shows the code of the module base, which is nested inside the module dataflow. It provides a base implementation of the extensible dataflow analysis framework discussed earlier. The four parameters \((G, I, L, F_n)\) of a dataflow analysis framework correspond to class cfg, type Item,

---

2 Nested interfaces are similar to nested types except that nested interfaces can be used to constrain types and that nested types need not necessarily be bound to interfaces.
class itemlat, and class transfer, respectively. The rest of the module—especially class worker, which implements the worklist algorithm (lines 5–26)—is generic with respect to the choice of these parameters. Therefore, writing a specific dataflow analysis amounts to instantiating these four parameters in a derived module. Crucially, this instantiation can be done in a lightweight, type-safe way by either binding or further-binding the four nested components.

To illustrate such an extension, Figure 9 shows a module that inherits from the base dataflow module and implements live variable analysis. Recall that live variable analysis is a backward analysis where the items are sets of local variables, the meet operator \( \sqcap \) takes the union of two sets, the greatest element \( \sqcup \) is the empty set, and the transfer functions are defined in terms of the variables each CFG node defines and uses. The module definition (liveness) declares that it extends the base module. It provides the definitions of exactly these four implicit parameters.

The combined power of family inheritance and retroactive adaptation is apparent: with roughly 20 lines of straightforward code, we are able to implement a new program analysis. And this analysis is itself extensible; for example, it can be further extended to report unused variables. Implementing this example with the same extensibility in any previous language would require more boilerplate code or design patterns.

### 5.2 Further Binding

In Familia, all nested names are points of extensibility that can be further-bound in derived modules. In addition, base modules, superclasses, superinterfaces, supertypes, and interfaces of classes can be further-bound in derived modules.

---

**Figure 8.** Excerpt from an extensible dataflow analysis framework.

(A Peer is a vertex in the CFG, and has access to an AST node, Node.)
dataflow.liveness

module dataflow.liveness extends dataflow.base {
    class cfg extends transpose[Peer,Edge with flowGraph]; // Backward analysis
    type Item = Set[Var]; // Liveness analysis propagates sets of variables
}

dataflow.liveness.transfer

// Def-Use
class transfer for Transfer(Node) {
    Item LocalVar.apply(Item item) {
        return item.add(this.var());
    }
    Item LocalAssign.apply(Item item) {
        LocalVar n = this.left();
        return item.remove(n.var());
    }
}

dataflow.liveness.itemlat

class itemlat for LowerLattice(Item) {
    static Item top() fixes Item {
        return new HashSet[Var]();
    }
    Item meet(Item that) {
        return this.addAll(that);
    }
    boolean leq(Item that) {
        return this.(setPO[Var].leq)(that);
    }
    boolean equals(Item that) {
        return this.(setPO[Var].equals)(that);
    }
}

Figure 9. An extension of the base dataflow framework: live variable analysis.

Binding nested, unbound names. A derived module can bind the nested components left unbound in its base module. In the example, derived module liveness binds three components unbound in module base: it binds nested class cfg by using the transpose of flowGraph (we assume flowGraph is a class for constraint Graph(Peer,Edge)) as its superclass, it binds nested type Item to type Set[Var], and it binds nested class itemlat to a nested class definition. A module can nest unbound methods, classes, or types if it is abstract or one of its families is abstract; 3 module base is declared abstract.

Unbound classes are not to be confused with abstract classes. An unbound class is one that a non-abstract, derived module of one of its enclosing families must provide a binding for. So unlike abstract classes, unbound classes can be used to satisfy constraints (including creating objects) and dispatch method calls. For example, consider the worklist() method in the base module, which computes the strongly connected components of the CFG (line 7) to achieve faster convergence [Nielson et al. 1999]. Class cfg, though unbound, is used (as the default class) to satisfy the constraint required by the generic findSCCs() method from Figure 5. As another example, unbound class itemlat is used as the (inferred) dispatcher in the method call on line 14.

It is perfectly okay to give partial definitions to nested classes cfg and itemlat without declaring them abstract, because a non-abstract, derived module of their family is required to complete the definitions. (Class itemlat is indeed partially defined because it inherits a default method implementation from its interface defined in Figure 4.) It follows that the above discussion about unbound classes applies to partially bound classes as well. Previous languages with family polymorphism do not support non-abstract classes that are unbound or partially bound.

Nested type Item is essentially an associated type of its family. Associated types are unbound type members of interfaces and superclasses [Järvi et al. 2005; Chakravarty et al. 2005b]. Associated types were introduced to reduce the number of type parameters, a rationale that applies here as well.

3 “Abstract methods” in previous OO languages actually mean unbound methods in Familia, while “abstract classes” retain their meaning.

4 Associated types take different forms in different languages: typedef in C++, abstract type members in Scala, associatedtype in Swift, to name a few.
interface I { int This.m(); }

class c1 for I(Fields) {

type Fields = { int f }

int Fields.m() { return this.(self.n)(); }

int Fields.n() { return this.f; }
}

class c2 for I(Fields) extends c1 {

type Fields = { int f; int g }

int Fields.n() { return this.f + this.g; }
}

class c2 for I(Fields) extends c1 {

type Fields = { int f; int g }

int Fields.n() { return this.f + this.g; }
}

// Testing code

{ int f } t1 = { f = 1; }

{ int f; int g } t2 = { f = 0; g = 2; }

I i = new c2(t2);

i.(I.m)(); // dispatcher is the natural class

t2.(c2.m)(); // dispatcher is c2

t1.(c2.m)(); // illegal

Figure 10. On the left are two classes with fields as their representations (Section 5.2). Testing code on the right is used to illustrate how late binding ensures type safety (Section 5.3). Receiver types in method signatures and dispatcher classes in method calls are written out in this example.

However, unlike languages like Scala and Haskell that support associated types, it is possible in Familia to further-bind a previously bound nested name in a derived module.

Further-binding nested names. In Familia, a derived module can further-bind nested interface and class definitions, specializing and enriching the behavior of the corresponding definition in the base module. Further binding was introduced by Beta [Madsen et al. 1993]. Languages that support virtual types but not full family polymorphism, such as Scala, can only simulate further binding through design patterns [Odersky and Zenger 2005; Weiel et al. 2014].

In the dataflow example, class transfer in the derived module further-binds its counterpart in the base module. Recall from Section 4.2 that a class can provide specialized behavior for subtypes of the representation type and that all such methods are dynamically dispatched. The implementation of the transfer functions demonstrates how family inheritance interacts with this feature. The transfer class in the base module provides a default implementation of a transfer function through the method apply. The transfer class in the derived module liveness inherits this default implementation, and refines it by specializing the implementation of apply to handle the particular types of AST nodes that play an interesting role in live variable analysis. Dispatching on the receiver object is used to allow choosing the most specific method among all of the three apply methods at run time.

In addition to specialization, a further-bound interface or class definition can also enrich the corresponding definition in the base module. This enrichment was seen in the example from Section 2, in which the class Node and implicitly, all of its derived classes, were extended with a new method constFold. This example works in Familia as well. If Node were an interface rather than a class, Familia would check that every class implementing it in the derived module provides a definition for method constFold.

A nested type can also be further-bound to a subtype. In fact, further-binding is what allows subclasses to add new fields. Recall from Section 4 that the representation type of a class with fields is a record type containing the field types. We take this unification a step further with nested types: field types are essentially a nested record type that can be further-bound to a subtype. For example, consider classes c1 and c2 on the left of Figure 10, whose representation types are a nested type. Class c2 adds a new field g by further-binding the nested type Fields to a subtype. Since this has type Fields in class c2, the nested fields can be accessed via this.f and this.g.
**Further-binding base modules.** In Familia, not only can nested names be further-bound, but base modules, superclasses, superinterfaces, and interfaces that classes implement can also be further-bound. The utility of further-binding base modules can be demonstrated by the new opportunity in Familia to co-evolve related, non-nested families of classes and interfaces.

For example, suppose the dataflow framework in Figures 8 and 9 was developed for the Salad programming language from Section 2. Because the Saladx extension in Figure 1 adds unary expressions to Salad, we cannot expect the dataflow framework to automatically work correctly for Saladx. If the `dataflow` module happened to be nested within the `salad` module, family polymorphism with further binding would allow us to add a transfer function for unary expressions in class `saladx.dataflow.liveness.transfer` (which would further-bind class `salad.dataflow.liveness.transfer`). This approach is illustrated in Figure 11a. Suppose, however, that the dataflow framework were implemented separately by a third party, and thus had to import the module `salad` rather than residing within it. The extensibility strategy just outlined would not work.

This need to co-evolve related families is addressed by further-binding a base module. Figure 11b illustrates how to co-evolve the dataflow framework and the Salad implementation, and Figure 12 shows the code. In Figure 12a, module `dataflow`, the dataflow framework for the base Salad language, declares a nested name `lang` and binds it by using `salad` as its base module. In Figure 12b, derived module `dataflowx`, the dataflow framework for Saladx, further-binds the base module of `lang` to `saladx`, and updates the transfer function to account for unary expressions. (For brevity, we also say that module `dataflow` binds nested name `lang` to `salad` and that module `dataflowx` further-binds it to `saladx`.) As we soon see in Section 5.3, the `dataflow` and `dataflowx` modules interpret names imported from the `salad` and `saladx` modules (e.g., `Node`) relative to their own nested component `lang`. This relativity ensures that the relationships between the related components in modules `salad` and `dataflow` are preserved when inherited into derived modules `saladx` and `dataflowx`; components of module `salad` cannot be mixed with components of module `dataflowx`, which work with `saladx`.

Familia supports this kind of further binding for other kinds of nested components than modules. For example, when a derived module further-binds the superclass of a nested class, Familia checks that the new superclass extends—up to transitivity and reflexivity—the original superclass. This check ensures that inherited code is type-safe with respect to the new binding.
5.3 Late Binding of Nested Names via self

A key to making family polymorphism sound is that relationships between nested components within a larger module are preserved when inherited into a derived module. Familia statically ensures that relationships between nested components are preserved through *late binding* of all nested names. To see what would go wrong without late binding, consider the right side of Figure 10. Without late binding of the nested name *Fields*, the last method call would type-check because the *m()* in *c2* is inherited from *c1* and its receiver type would be `{int f}` instead of the late-bound *Fields*. This would result in a run-time error as *t1* does not have the *g* field.

Recall from Section 4.3 that Familia uses the keyword `self` to represent the current class or module. By adding a qualifier, `self` can also be used to refer to any enclosing class or module. Names that are not fully qualified are interpreted in terms of an enclosing class or module denoted by a `self`-prefix. For example, consider the mentions of the unqualified name *Fields* in class *c1* of Figure 10. Here *Fields* is syntactic sugar for the qualified name `self[c1].Fields`, where the prefix `self[c1]` refers to the enclosing class that is, or inherits from *c1*. When the code is inherited into class *c2*, the inferred `self`-prefix becomes `self[c2]`, a class that is, or inherits from *c2*. The unqualified name *Fields* then has a new meaning in class *c2*: `self[c2].Fields`, where the prefix `self[c2]` refers to the enclosing class that is, or inherits from *c2*. On the right of Figure 10, when class *c2* is used to invoke method *m*, the method signature is obtained by substituting the dispatcher for the `self` parameter in `int self[c2].Fields.m()`. This substitution suggests that the receiver should have type `{int f; int g}`, so the last method call `t1.(c2.m)()` is rejected statically.

The further binding of the base module of nested name *lang* offers another example. The transfer class of the base dataflow module (Figure 8) mentions *Node*, defined by module *salad.ast*. Because the enclosing module *dataflow* binds *lang* to *salad* (Figure 12a), *Node* is expanded to *lang.ast.Node*. Further desugaring the mention of *lang* yields `self[dataflow].lang.ast.Node`. Hence, in the derived module *dataflowx* (Figure 12b), the unqualified name *Node* is reinterpreted as `self[dataflowx].lang.ast.Node`. Similarly, module *dataflowx* interprets the unqualified name *Stmt* as `self[dataflowx].lang.ast.Stmt`, so the subtyping relationship between *Stmt* and *Node* is preserved.

Importantly, late binding in Familia supports separate compilation with modular type checking—existing code need not be rechecked or recompiled when inherited into derived modules. For example, derived module *liveness* inherits method `outflow()` (lines 20–25 of Figure 8), which takes the meet of all incoming flows of a node and computes the outgoing flow by applying the transfer function. The call to `apply()` on line 24 in the base module need not be rechecked in a derived module, say `dataflowx.liveness`, which interprets the receiver type as `self[dataflowx].lang.ast.Node`, the formal parameter type `self[dataflowx.liveness].Item`, and the dispatcher class as `self[dataflowx.liveness].transfer`. This guarantees that the method can only be invoked using arguments and dispatcher that are compatible.
To make the semantics of the unified polymorphism mechanism more precise, we define a core language called Featherweight Familia (abbreviated F²), capturing the key aspects of the unification mechanism behind Familia.

6 A CORE LANGUAGE

To make the semantics of the unified polymorphism mechanism more precise, we define a core language called Featherweight Familia (abbreviated F²), capturing the key aspects of the unification mechanism behind Familia.

6.1 Syntax and Notation

F² follows the style of Featherweight Java [Igarashi et al. 2001], and makes similar assumptions and simplifications. It omits a few convenient features of Familia: in F², uses of nested names are fully expanded, the class used by a method call or a with-clause is always explicit, and natural classes are encoded implicitly rather than generated implicitly. For simplicity, F² does not model certain features of Familia whose formalization would be similar to that in Featherweight Genus [Zhang et al. 2015a]: interfaces with multiple constraint parameters, multimethods, and use-site genericity.

Figure 13 presents the syntax of F². An overline denotes a (possibly empty) sequence or mapping. For example, X denotes a sequence of type variables, m:S denotes an (unordered) mapping that maps method names m to method signatures S, and \( \ominus \) denotes an empty sequence or mapping. The \( i \)-th
A class path represents the use of a class. Class paths have exactness, a notion that is also seen in previous approaches to type-safe extensibility [Bruce et al. 1997; Nystrom et al. 2004; Bruce and Foster 2004; Nystrom et al. 2006]. An exact class path $P$ denotes a particular class, while an inexact class path $P'$ abstracts over all of its subclasses (including itself). Inexact class paths are of the form $P'.c[\omega]$ and can be used in $\text{extends}$-clauses in class headers as superclasses. Similar to Featherweight Java, $F^2$ assumes the well-foundedness condition that there are no cycles in inheritance chains, as well as the existence of a distinguished, universal superclass $\text{empty}$.

An exact class path $P'$ may take one of the following forms:

1. $\diamond$, denoting the program $\mathcal{P}$ that nests everything;
2. $P'.c[\omega]$, denoting class $c[\omega]$—not including a subclass thereof—nested within $P'$; or
3. self[$P$], denoting an enclosing class that must either be $P$, extend $P$, or further-bind $P$.

For example, in a class definition named $c_2$ nested within class definition $c_1$, which is nested within $\mathcal{P}$, the current class $c_2$ is referred to as self[self[self[self[c_1,c_2]]]]. Familia uses the lighter syntax self[c_1,c_2] to denote this path, or even just self if $c_2$ is the immediately enclosing class, but the heavier syntax in $F^2$ makes it straightforward to perform substitution for outer self-parameters like self[c_1,c_2]. Some paths may never appear in programs; for example, the path self[self[c_1,c_2]] has valid syntax but cannot appear in $F^2$ programs. Nevertheless, the static semantics may create such paths to facilitate type checking. Given an inexact class path $P_1 = P_2'.c[\omega]$, we use $P_1'$ to mean the exact class path $P_2'.c'[\omega]$.

Although $F^2$ requires explicit exactness annotations (i.e., !), they are usually not needed in Familia. The exactness of certain uses of classes is obvious and thus inferred: class paths used in $\text{extends}$-clauses are inexact, but class paths used in $\text{with}$-clauses, $\text{pack}$-expressions, and as dispatchers in method calls are all exact.

An interface path $H$ represents the use of an interface. Similar to class paths, interface paths can be exact ($Q'$) or inexact ($P'.I[\omega]$). Inexact interface paths can be used in $\text{extends}$-clauses in interface headers and for-clauses in class headers. The distinguished interface $\text{Any}$ is the universal superinterface.5

An exact interface path $Q'$ may take one of the following forms:

1. $P'.I[\omega]$, denoting interface $I[\omega]$—not including a subinterface thereof—nested within $P'$;

---

5 Class $\text{empty}$ and interface $\text{Any}$ are nested directly within the program $\mathcal{P}$. Class $\text{empty}$ is parameterized by a type variable $X$ and implements the constraint $\diamond \text{Any}(X)$. A class extending $\diamond \text{empty}$ instantiates $X$ as its representation type.
values \( v :\!::= n \mid \text{pack}(v, P') \)

evaluation contexts \( E :\!::= [] \mid E.(P'.m[\omega])(\overline{\tau}) \mid v_0.(P'.m[\omega])(\overline{\tau}_1, E, \overline{\tau}) \mid \text{pack}(E, P') \mid \text{unpack } E \text{ as } (x, p) \text{ in } e_2 \)

\[
\diamond \rightsquigarrow P' \vdash e_1 \longrightarrow e_2
\]

\text{[UNPACK]}

\[
\diamond \rightsquigarrow P' \vdash \text{unpack } (\text{pack } (v, P')) \text{ as } (x, p) \text{ in } e \longrightarrow e\{v/x\}\{p'/p\}
\]

\text{[CALL]}

\[
\diamond \rightsquigarrow P'_{\text{prog}} \vdash P' \rightsquigarrow P'_{\text{disp}} \quad P'_{\text{disp}}(m) = [\varphi]T_1 \rightarrow T_2(\overline{\tau})(e)
\]

\[
\diamond \rightsquigarrow P'_{\text{prog}} \vdash v_0.(P'.m[\omega])(\overline{\tau}_1) \longrightarrow e\{v_0/\text{this}\}\{\omega/\varphi\}\{\overline{\tau}_1/\overline{\tau}\}
\]

\text{Figure 14. Featherweight Familia: operational semantics.}

\text{type environments } \Delta :\!::= \emptyset \mid \Delta, \text{Self}[Q] \mid \Delta, x
\text{class environments } K :\!::= \diamond \rightsquigarrow P' \mid K, \text{Self}[P] \mid K, p \text{ for } H(T)
\text{term environments } \Gamma :\!::= \emptyset \mid \Gamma, x: T

\text{Figure 15. Featherweight Familia: environment syntax.}

(2) \text{Self}[Q], denoting an enclosing interface that must either be } Q, \text{ extend } Q, \text{ or further-bind } Q; \text{ or}
(3) \text{d.itf}, denoting the interface implemented by dispatcher } d.

For example, in the class definition of } c_2 \text{ nested within } c_1 \text{ which is nested within } P, \text{ the interface implemented by the current class is denoted by } \text{self[Self[\cdot].} c_1], c_2].\text{itf}. \text{In Familia, this interface is denoted by the lighter syntax Self[c_1, c_2]. Familia also supports interface paths with inexact prefixes (i.e., } P'.c[\omega_1].1[\omega_2]); \text{ they are not modeled in } F^2 \text{ for simplicity.}

A type } T \text{ is either the integer type, a type variable } X, \text{ or an object type denoted by an interface path, which can be either exact or inexact. Inexactly typed values may have a run-time type that is a subtype, while exactly typed values cannot. A dispatcher } d \text{ is either an exact class path or a class variable. Dispatchers are used in method calls } e_0.(d.m[\omega])(\overline{\tau}_1), \text{ object-creation expressions pack } (e, d), \text{ and with-clauses. Function } FTV(\bullet) \text{ returns the type variables occurring free in } \bullet. \text{ Function } FCV(\bullet) \text{ returns free class variables.}

6.2 Dynamic Semantics

Figure 14 presents the operational semantics of Featherweight Familia, including its values, evaluation contexts, and reduction rules. Object values take the form pack \((v, P')\). Reduction rule \text{[UNPACK]} unpacks an object. Rule \text{[CALL]} reduces a method call. The method body to evaluate is retrieved from \text{P'}_{\text{disp}}, \text{ the linkage of the dispatcher } P'. \text{ A linkage provides a dispatch table indexed by method names, as discussed in more detail in Section 6.3.}

6.3 Static Semantics

The complete static semantics of Featherweight Familia can be found in the accompanying supplemental material. Below we explain the judgment forms and discuss selected judgment rules shown in Figures 16 and 18.

\text{Environments.} The syntax of environments is shown in Figure 15. A type environment } \Delta \text{ contains Self}[Q] \text{ parameters as well as type variables. A class environment } K \text{ always contains the linkage of the entire program } (\diamond \rightsquigarrow P'). \text{ It may also contain Self-parameters as well as class variables.}
For example, checking program $\mathcal{P}$ adds the program linkage into $\Delta$, checking class $c$ (nested within $\mathcal{P}$) adds $\text{Self}[\text{self}[\text{\circ}.c]]$ into $\Delta$, and checking interface $I$ (nested within $c$) adds $\text{Self}[\text{self}[\text{\circ}.c].I]$ into $\Delta$.

**Constrained parametric polymorphism.** As shown in Figure 13, all nested components ($C$, $I$, and $M$) can be parameterized, so their well-formedness rules require the well-formedness of the parameters $[\varphi]$, which is expressed using $\Delta; K \vdash [\varphi] \text{OK}$. Subsequent checks in these well-formedness rules are performed under the environments $\Delta$, $\Delta_\varphi$, and $K$, $K_\varphi$, where $\Delta_\varphi$ and $K_\varphi$ consist of the type parameters and class parameters of $[\varphi]$. The well-formedness rules of class paths, interface paths, and method-call expressions correspondingly check the validity of the substitution of arguments $[\omega]$ for parameters $[\varphi]$. These checks use the judgment form $\Delta; K \vdash [\omega/\varphi] \text{OK}$, and its rule is given by [INST] in Figure 16. In addition to the well-formedness of the arguments, it requires the constraints implemented by the dispatchers to entail the corresponding where-clause constraints. Constraint entailment is expressed using the judgment form $K \vdash H_1(T_i) \equiv H_2(T_i)$.

**Object-oriented polymorphism.** The typing of pack- and unpack-expressions is given by rules [E-PACK] and [E-UNPACK] in Figure 16. The expression $\text{pack}(e,d)$ packs $e$ and the dispatcher $d$ into an object, where $e$ is the underlying representation and $d$ is the class implementing the object type $d.\text{itf}$. The expression $\text{unpack} e_1 \text{as} (x,p)$ in $e_2$ unpacks object $e_1$ into its representation and class (bound to $x$ and $p$, respectively) and evaluates $e_2$, in which $x$ and $p$ may occur free. The standard existential-unpacking rule requires the freshly generated type variable not to occur free in the resulting type; likewise, rule [E-UNPACK] requires the same of the freshly generated class variable.
While Familia automatically generates natural classes for interfaces, $F^2$ gives a concrete encoding of natural classes via unpack-expressions. For example, suppose variable $x_0$ has object type $\diamond . I$, an interface that requires a single method $m: \text{int} \rightarrow \text{int}$. Then invoking $m$ on $x_0$ can be written as $\langle \diamond . \text{natural}_I . m \rangle (8)$, where the natural class natural$_I$ is defined as follows:

```java
class natural_I for $\diamond . I . m$ extends $\diamond . \text{empty} . I$
{
  m : \text{int} \rightarrow \text{int} (x_1) { unpack this as (x_2,p) in x_2 . (p,m)(x_1) }
}
```

The natural class implements the method by unpacking the receiver and subsequently calling the method with the unpacked class as the dispatcher and the unpacked representation as the receiver. Some prior object encodings formalize objects as explicit existentials [Bruce 1994; Abadi et al. 1996]. The unpacking of receiver objects in natural classes is akin to the way these encodings unpack existentially typed objects before sending messages to them.

The way that natural classes are encoded suggests that not all interfaces have natural classes. In fact, an interface such as $\text{Eq}$ that uses its constraint parameter in contravariant or invariant positions other than the receiver type does not have a natural class. The reason is that the encoding of such natural classes would involve unpacking objects of the representation type every time the representation type appears in the method signature, including one for the receiver, and that the unpacked receiver does not necessarily have the same representation type as the other occurrences. The lack of natural classes for interfaces like $\text{Eq}$ means these interfaces cannot be used as object types as other interfaces (e.g., $\text{Set[E]}$) can. This restriction is not a limitation, though; a survey of a large corpus of open-source Java code finds that in practice programmers never actually use interfaces like $\text{Eq}$ as types of objects [Greenman et al. 2014].

**Method calls.** Rule [E-CALL] in Figure 16 type-checks a method call. The method signature that the call is checked against is retrieved from the linkage of the dispatcher’s interface; this linkage contains signatures for the methods the interface requires. When the dispatcher $d$ is a natural class, we get a typical object-oriented method call; when $d$ is a class variable, we have a method call enabled by a type-class constraint; and when $d$ is any other class, we get an “expander call” [Warth et al. 2006] that endows the receiver with new behavior. Rule [E-CALL] unifies all these cases.

**Family polymorphism.** Central to the semantics is the notion of linkages. A class linkage $P$ (or $P'$) collects information about a class path of the form $P$ (or $P'$). As shown in Figure 17, a class linkage is a tuple comprising (1) the path, (2) the constraint being implemented, (3) the superclass, (4) the nested method definitions, (5) the linkages of the nested interfaces, and (6) the linkages of the nested classes. The linkage of an inexact class path $P$ is parameterized by a self$[P]$ parameter;

<table>
<thead>
<tr>
<th>$P'$-linkages</th>
<th>$P$-linkages</th>
<th>$Q'$-linkages</th>
<th>$Q$-linkages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P' ::= \begin{cases} \diamond \ \bullet \ \emptyset \ I: [\varphi_1] Q \ c: [\varphi_2] P \end{cases}$</td>
<td>$P ::= \begin{cases} { self[P] \ Q(T) \ P_{sup}</td>
<td>\bullet \ m: S(x)(e) \ I: [\varphi_1] Q \ c: [\varphi_2] P \end{cases}$</td>
<td>$Q' ::= \begin{cases} Q' \ v \ Q_{sup}</td>
</tr>
</tbody>
</table>

Figure 17. Featherweight Familia: linkage syntax.
Figure 18. Featherweight Familia: selected rules for computing linkages.

Substitution for `self[P]` in that linkage is thus capture-avoiding. We emphasize this fact by putting this `self-path`, instead of the inexact path, as the first element of the tuple. Given the linkage $P$ of an inexact class path $P$, we use $P'$ to mean the linkage of the exact class path $self[P]$. Interface linkages $(Q'_1$ and $Q$) contain fewer components. In the linkage of an inexact interface path $Q$, $Self[Q]$ may occur free. Looking up an (exact) linkage for a nested component named `id` is denoted by $P'[id]$ or $Q'[id]$.
The well-formedness rules for paths can be found in the supplemental material, but here Figure 18 presents the rules that compute linkages for paths. The corresponding judgment forms are $K \vdash P' \leadsto P'$, $K \vdash P \leadsto P$, $K \vdash Q' \leadsto Q'$, and $K \vdash Q \leadsto Q$.

Linkages are computed in an outside-in manner. As shown in rule [PROG] in Figure 16, the linkage of a program is obtained via the helper function $\text{flatten}(\bullet)$ and added to the environment. Rule [P!-PROG] in Figure 18 retrieves this linkage from the environment. The helper function $\text{flatten}(\bullet)$ is defined in the supplemental material; it does not do anything interesting except converting the program text into a tree of linkages. Importantly, linkages nested within an outer linkage do not contain components that are inherited. Thus all nested linkages are incomplete.

Rule [P] in Figure 18 computes the linkage of an inexact class path $P'.c[\omega]$. It first computes the linkage $P'_{\text{un}}$ of the family $P'$, from which the nested linkage $P'_{\text{nest}}$ corresponding to nested class $c$ is obtained. The helper function $\text{parent}(\bullet)$ finds the superclass, whose linkage is $P'_{\text{sup}}$. While $P'_{\text{nest}}$ is incomplete, the superclass linkage $P'_{\text{sup}}$ is complete. The complete linkage of $P'.c[\omega]$ is then obtained by concatenating $P'_{\text{sup}}$ with $P'_{\text{nest}}{\omega}/\phi$, using the linkage concatenation operator $\oplus$ defined by rule [CONCAT-P] in Figure 18. Operator $\oplus$ is defined recursively; a nested linkage is concatenated with the corresponding linkage it further-binds to produce a new nested linkage (see other $\oplus$-rules in the supplemental material). Importantly, $\oplus$ replaces the self-parameter of the first linkage with that of the second linkage; this substitution is key to late binding of nested names. Linkage concatenation is also what enables dynamic dispatch for object-oriented method calls (i.e., calls using a natural class as the dispatcher), because a method in a linkage $P$ overrides less specific methods of the same name in linkages to which $P$ is concatenated.

Rule [P!-NEST] shows that the linkage of an exact path $P'.c'[\omega]$ is obtained by substituting $P'.c'[\omega]$ for $\text{self}[P'.c[\omega]]$ in the linkage of $P'.c[\omega]$.

Some interface paths are equivalent. For example, interface path $p\text{itf}$, where $p$ is declared to witness constraint $Q'(T)$, is equivalent to $Q'$. This equivalence relation is captured by path normalization ($\equiv$). Rules [NORM-ABS] and [NORM-CV] in Figure 18 simplify interface paths of form $d\text{itf}$. A path is simplified to its normal form after finite steps of simplification ($\equiv^\ast$). Other normalization rules are purely structural; they and the normal forms can be found in the supplemental material. The linkage computation rules in Figure 18—except for [Q!-NORM]—are defined for paths of normal forms. Rule [Q!-NORM] suggests that the linkage of a path is the same as that of its normal form. The equivalence relation $\equiv$ (shown in the supplemental material) is then the symmetric closure of $\equiv^\ast$. Because of this path equivalence, substitution for $\text{self}[P]$ (resp. $\text{self}[Q]$) also replaces other self-paths (resp. Self-paths) that are equivalent with $\text{self}[P]$ (resp. $\text{self}[Q]$).

Soundness of family polymorphism hinges on a few conformance checks. For example, if a nested interface definition adds new methods in a derived module, then classes implementing the interface in the base module should also be augmented in the derive module to provide definitions for the new methods. This conformance of classes to interfaces is expressed by the judgment form $K \vdash P' \text{-Conform}$.

Another conformance condition, $K \vdash P' \text{-FB-Conform}$, requires that nested classes and interfaces conform to classes and interfaces they further-bind. In particular, the superclass (or interface) of a nested class in a derived module should be a subclass (or subinterface) of that of the further-bound class (or interface) in the base module. Also, a nested, further-binding interface should not change its variance with respect to the representation type. These checks ensure that inherited code still type-checks in derived modules.

The rules performing the conformance checks above are given in the supplemental material. They work by recursively invoking the checks on nested classes. At the top level, they are invoked from rule [PROG] in Figure 16.
Decidability. Because F\textsuperscript{2} does not infer default classes, the decidability of its static semantics is trivial: the well-formedness rules and the linkage-computation rules are syntax-directed (the subsumption rule can be easily factored into the other expression-typing rules, and the path normalization rules are defined algorithmically), and a subtyping algorithm works by climbing the subtyping hierarchy. The inference of default classes in Familia could potentially introduce nontermination, similar to how default model inference in Genus could lead to nontermination [Zhang et al. 2015a]. We expect that termination can be guaranteed by enforcing syntactic restrictions in the same fashion as in Genus. In particular, the restrictions include Material–Shape separation [Greenman et al. 2014], which, in the context of Familia, prevents the supertype of an interface definition from mentioning the interface being defined; for example, an interface like Double cannot be declared to implement Comparable[Double]. Java allows such supertypes so that types like Double can satisfy F-bounded constraints [Canning et al. 1989] like <T extends Comparable<? super T>>. Familia would use a where-clause instead (i.e., [T where Ord(T)]), eliminating possible nontermination when checking subtyping [Grigore 2017] while adding the flexibility of retroactively adapting types to constraints.

6.4 Soundness

We establish the soundness of Featherweight Familia through the standard approach of progress and preservation [Wright and Felleisen 1992]. The key lemmas and their proofs can be found in the supplemental material.

**Lemma 6.1 (Progress)** If (i) ⊢ \mathcal{P} OK, (ii) flatten(\mathcal{P}) = \mathbb{P}^!; and (iii) ∅; ⊓ → \mathbb{P}^!; ∅ ⊢ e : T, then either e is a value or there exists e’ such that ⊓ → \mathbb{P}^! ⊢ e → e’.

**Lemma 6.2 (Preservation)** If (i) ⊢ \mathcal{P} OK, (ii) flatten(\mathcal{P}) = \mathbb{P}^!; (iii) ∅; ⊓ → \mathbb{P}^!; ∅ ⊢ e : T, and (iv) ⊓ → \mathbb{P}^! ⊢ e → e’, then ∅; ⊓ → \mathbb{P}^!; ∅ ⊢ e’ : T.

**Theorem 6.1 (Soundness)** If (i) ⊢ e [\mathcal{T} C e] OK, (ii) flatten(\diamond [\mathcal{T} C e]) = \mathbb{P}^!, and (iii) ⊓ → \mathbb{P}^! ⊢ e → e’∗, then either e’ is a value or there exists e’ such that ⊓ → \mathbb{P}^! ⊢ e’ → e”.

7 RELATED WORK

Language design remains a core topic of programming-languages research. One way to evaluate language designs is by comparison with prior work, and indeed decades of prior work on language support for extensible and composable software has developed many mechanisms for extensibility and genericity. However, we argue that no prior work integrates the different forms of polymorphism as successfully as Familia does.

**Constrained parametric polymorphism.** Central to a parametric-polymorphism mechanism is a way to specify and satisfy constraints on type parameters. Many prior languages have experimented with either nominal subtyping constraints (e.g., Java and C#) or structural matching constraints (e.g., CLU [Liskov et al. 1984] and Cecil [Chambers 1992]). Both approaches are too inflexible: types must be preplanned to either explicitly declare they implement the constraint or include the required methods with conformant signatures. At the same time, typing is made difficult by the interaction between inheritance and constraints that require binary methods. F-bounded polymorphism [Canning et al. 1989] and match-bounded polymorphism [Bruce et al. 2003, 1997; Abadi and Cardelli 1996] are proposed to address this typing problem. However, they do not address the more urgent need to allow types to retroactively satisfy constraints they are not prepared to satisfy; this inflexibility is an inherent limitation of subtyping-based approaches.

To allow retroactive adaptation, more recent work on language support for generic programming follows Haskell type classes [Wadler and Blott 1989]. JavaGI [Wehr and Thiemann 2011] supports generalized interfaces that can act as type classes. A special implementation construct is used as
a type-class instance. Genus [Zhang et al. 2015b] introduces constraints and models on top of interfaces and classes. It avoids the global uniqueness limitation of Haskell and JavaGI—that type-class instances are globally scoped and that a given constraint can only be satisfied in one way. To avoid complicating the easy case, Genus allows constraints to be satisfied structurally via natural models. Genus also supports model-dependent types that strengthen static checking and model multimethods that offer convenience and extensibility. Familia incorporates all these Genus features without requiring extra constructs for constraints or models. Unlike Familia, neither JavaGI nor Genus supports associated types.

Support for generic programming in Rust [Rust 2014] and Swift [Swift 2014] is inspired by type classes as well. In Rust, objects and generics are expressed using the same constructs (trait and impl), but Rust lacks support for implementation inheritance. These languages also have the limitation of global uniqueness. Dreyer et al. [2007] and Devriese and Piessens [2011] integrate type classes into ML and Agda, respectively, with a goal of not complicating the host language with duplicate functionality. Although not intended for generic programming, expanders [Warth et al. 2006] and CaesarJ wrappers [Aracic et al. 2006] support statically scoped adaptation of classes to interfaces.

In Scala, generics are supported by using the Concept design pattern, made more convenient by implicits [Oliveira et al. 2010]: traits act as constraints, and trait objects are implicitly resolved arguments to generic abstractions. This approach does not distinguish types instantiated with different trait objects (cf. Familia types that keep track of the classes used to satisfy constraints), and does not allow specializing behavior for subtypes of the constrained type (cf. class multimethods in Familia). Scala also supports higher-order polymorphism by allowing higher-kindred type parameters and virtual types [Moors et al. 2008]. Familia supports higher-order polymorphism via nested, parameterized types and interfaces. Because nested components can be further-bound, higher-order polymorphism in Familia goes beyond Scala’s higher-kindred virtual types.

**Family polymorphism.** Prior work on family polymorphism has been largely disjoint from work on parametric polymorphism. Virtual types [Thorup 1997; Torgersen 1998; Igarashi and Pierce 1999] are unbound type members of an interface or class. They support family polymorphism [Ernst 2001], with families identified by an instance of the enclosing class. Virtual types inspired Haskell to add associated types to type classes [Chakravarty et al. 2005b,a]. Virtual types are not really “virtual”: once they are bound in a class, their bindings cannot be refined as can those of virtual methods and virtual classes. In this sense, they act more like type parameters; in fact, virtual types are considered an alternative approach to parametric polymorphism [Thorup 1997]. It is understood that virtual types are good at expressing mutually recursive bounds [Bruce et al. 1998]; this use of virtual types in generic programming is largely subsumed by the more flexible approach of multi-parameter type classes [Jones et al. 1997] available in Haskell, JavaGI, Genus, and Familia.

Virtual classes, both based on object families [Madsen et al. 1993; Madsen and Møller-Pedersen 1989; Ernst 1999; Aracic et al. 2006; Bracha et al. 2010], and class families [Nystrom et al. 2004, 2006; Clarke et al. 2007; Qi and Myers 2010], offer a more powerful form of family polymorphism than virtual types do: a subclass can specialize and enrich nested classes via further binding. Path-dependent types are used to ensure type safety for virtual types and virtual classes (e.g., Nystrom et al. [2004]; Ernst et al. [2006]). A variety of other mechanisms support further binding, including virtual classes, mixin layers [Smaragdakis and Batory 2002], delegation layers [Ostermann 2002], and variant path types [Igarashi and Viroli 2007]. The family-polymorphism mechanism in Familia is closest to that in Jx [Nystrom et al. 2004]. Our use of prefix types is adapted from Jx; the fact that self-prefixes can be inferred makes family polymorphism lightweight in Familia.
Unlike the class-family approach taken in Familia, the object-family approach (virtual classes) does not readily support cross-family inheritance. For example, with virtual classes, class a.b.c cannot extend class a.d.e because class a.b.c has no enclosing instance of a.d. Tribe [Clarke et al. 2007] and Scala support cross-family inheritance for virtual classes and virtual types, respectively, but by adding extra complexity to virtual classes or by resorting to verbose design patterns. Few prior languages support coordinated evolution of related, non-nested families. Cross-family inheritance and cross-family coevolution are crucial to deploying family polymorphism at large scale, where we expect components from different modules to be frequently reused and composed.

Scala supports virtual types but not virtual classes, simulating the latter with a design pattern [Odersky and Zenger 2005]. While this pattern seems effective at a small scale for tasks like the Observer pattern, it does not scale to a larger setting where cross-family inheritance is needed, where entire frameworks are extended, and where further binding is therefore needed at arbitrary depth. The effort required to encode virtual classes in Scala appears to be significant [Weiel et al. 2014]. Scala also supports mixin composition. A mixin has an unbound superclass that is bound in classes that incorporate the mixin. Familia is expressive enough to encode mixin composition via late binding of superclasses, rather than requiring a separate language mechanism for mixins.

Familia extends the expressivity and practicality of family polymorphism. It allows classes to be unbound yet non-abstract. It also allows externally imported names to coevolve with the current module by further-binding base modules. Prior languages that support family polymorphism beyond virtual types have omitted support for parametric polymorphism. We believe support for parametric polymorphism is important even when virtual types are available, because it allows applicative instantiation of generic abstractions, often a more convenient and interoperable approach than virtual types [Bruce et al. 1998].

8 CONCLUSION

Familia achieves a high degree of expressive power by unifying multiple powerful mechanisms for type-safe polymorphism. The resulting language has low surface complexity—it can be used as an ordinary Java-like object-oriented language that supports inheritance, encapsulation, and subtyping. With little added syntax, several powerful features become available: parametric polymorphism with flexible type classes, wrapper-free adaptation, and deep family polymorphism with cross-family inheritance and cross-family coevolution. We have described the language intuitively and presented examples that illustrate its added expressive power. We have also given both operational and static semantics via $F^2$, a core language that we have proved type-safe. Comparisons with previous mechanisms for generic programming show that Familia improves expressive power in a lightweight way. Implementation of Familia is left to future work but should be guided by the formal semantics of $F^2$.

ACKNOWLEDGMENTS

We thank Sophia Drossopoulou for thoughtful syntax suggestions and the anonymous reviewers for their many useful comments. This work was supported by NSF grant 1513797.

REFERENCES


R. Grigore. Java generics are Turing complete. In 41st ACM Symp. on Principles of Programming Languages (POPL), 2017.


A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. on Programming Languages and Systems, 23(3):396–450, 2001.


S. Peyton Jones. Classes, Jim, but not as we know them—type classes in Haskell: What, why, and whither. In 23rd European Conf. on Object-Oriented Programming, 2009.


B. Stroustrup. The C4+0x “remove concepts” decision. Dr. Dobbs, July 2009.


