Parameterized Modules for Classes and Extensible Functions

Keunwoo Lee and Craig Chambers

University of Washington
Department of Computer Science and Engineering
Box 352350, Seattle WA 98195-2350, USA
{klee, chambers}@cs.washington.edu

Abstract. We present F(EML), a language that combines classes, extensible functions, symmetric multiple dispatching, and a practical system for *parameterized modules.* Parameterized modules permit subclasses and function extensions to be defined and typechecked once, and then reused to extend multiple argument modules. F(EML)'s predecessor, EML, supported classes and extensible functions with multiple dispatch, but its support for parameterized modules was weak. F(EML)'s key novel features are *alias declarations*, *generalized type relations* in module signatures, and a nontrivial definition of *signature subsumption*.

1 Introduction

Programmers should be able to write code so it can later be extended—with new cases of existing data types, and new cases of existing functions. Programmers should also be able to write these extensions so they can be reused to extend a wide range of base modules. Finally, these extensions should support modular reasoning, including modular typechecking. Unfortunately, it is hard to support all of these desiderata at once.

Consider the core of an interpreter in a language like Java:

```
package Lang;
abstract class Expr {
  Expr() {}
  abstract Int eval(); }
```

In the classic "expression problem" [35, 41], one wishes to add both new types of `Expr` and new functions over `Expr` types. In object-oriented languages, one can straightforwardly do the former without changing the original code:

```
package ConstPackage;
class Const extends Expr {
  Int value;
  Const(Int v_in) {value=v_in;}
  Int eval() {return value;} }
```

However, to add a new dispatching function for `Expr` — say, `print` — we must invasively alter the original code:

```
abstract class Expr {        ... // as before
  abstract String print(); }
class Const extends Expr {   ... // as before
  String print() { return value.toString(); } }
```

Traditional functional languages have the converse problem: adding new functions is easy, but adding new cases to data types requires invasive changes, either to the original source code, or to existing clients.

Previous work on EML [28] and related languages [27, 10] integrates both object-oriented and functional extensibility in a single unified framework. These languages include extensible class hierarchies and method overriding (as in traditional object-oriented languages), while also allowing functions to be added externally to classes, and to dynamically dispatch on any subset of their arguments (as in traditional functional languages). In EML, we would write:

```
module Lang = { abstract class Expr() of {}
                abstract fun eval:Expr -> Int }
```

It is straightforward to add new data types:

```
module ConstMod uses Lang = {
  class Const(v_in:Int) extends Lang.Expr() of {value:Int = v_in}
  extend fun Lang.eval(Const {value=v}) = v }
```

Note that `extends` adds a new subclass to an existing class, and `extend fun` adds a new (pattern-matching) case to an existing function.

It is also straightforward to add new functions:

```
module PrintMod uses Lang, ConstMod = {
  fun print:Lang.Expr -> String
  extend fun print(Lang.Expr) = ""
  extend fun print(ConstMod.Const {value=v})= Std.intToString(v) }
```

EML therefore supports both data type and function extensibility (with some restrictions, which is why `print` has a default case for `Expr` — see Section 3.2).

Now, we would like it to support code reuse as well. Suppose the interpreter code base had many *features* — i.e., expression types, and functions over those types — and we wished to combine various subsets to produce a *product line* [24] of interpreters. In this case, we would like to define a feature once, typecheck it once, and reuse it to extend several interpreter instances.

Like ML [23, 30, 19, 11], EML supports *functors*, or parameterized modules:

```
signature LangSig = sig { abstract class Expr() of {}
                          abstract fun eval:Expr -> Int }
```

```
module MakePlus = (L:LangSig) -> {
  class Plus(l_in:Int, r_in:Int) extends L.Expr()
      of {left:Int = l_in, right:Int = r_in}
  extend fun L.eval(Plus {left=l, right=r}) = L.eval(l)+L.eval(r)}

module PlusMod = MakePlus(Lang)
```

`MakePlus` defines a function over modules; it can be applied to any module that implements `LangSig`, to produce a module containing a (freshly minted) class `Plus` and its `eval` implementation.

Note that `Plus` inherits from `L.Expr`, a class provided by the module parameter. In principle, such parameterization supports and subsumes many useful idioms, including mixins [5, 17] (`Plus` is a mixin), mixin layers [37] (which apply mixins to multiple classes at once), and certain aspect-oriented extensions [21] that extend members of multiple base modules.

However, limitations in EML prevent it from realizing this potential:

– EML functors are sensitive to the names of classes and functions in their arguments. In our example, `MakePlus` could only be applied to modules with a class named `Expr`. However, a truly reusable functor should be insensitive to inessential details like class names — other mixin systems, for example, do not constrain the names of classes with which a mixin may be composed.
– EML's signature language could only specify *direct* subclassing relations in functor arguments. Therefore, for example, it would be impossible to write an EML functor that extended a transitive subclass of `Expr`.
– EML included no useful form of *signature subsumption.* Therefore, for example, a module that provided all the features of `Lang`, plus some extra declarations, would be incompatible with `LangSig`.

In combination, these limitations meant that EML functors were not truly reusable. The contributions of the present work are as follows:

– We have designed F(EML), a language that combines EML's data type and function extensibility with practical, reusable parameterized modules. F(EML) enriches EML with three key features that lift the above limitations: (1) constructs for renaming declarations, and controlling the aliasing that results; (2) generalized type relations, including freshness information; and (3) useful signature subsumption.
– We have formalized the essence of F(EML) in a core language, Mini-F(EML). Section 3 summarizes the semantics and soundness properties; details will appear in a companion report [22].
– We have implemented a prototype F(EML) interpreter, and verified that it can typecheck interesting idioms. Our interpreter also supports some extensions (such as information hiding via *signature ascription*) which we do not discuss in this paper.

Finally, Sections 4 and 5 discuss related work and conclude.

2 Motivation and Design Overview

Fig. 1 gives the grammar of a F(EML) subset which we call F(EML)$^-$; except for shallow syntactic differences, this sublanguage corresponds roughly to EML. In the remainder of this section, we informally explain the semantics of this language using examples (Sections 2.1 and Section 2.2), show its limitations (Section 2.3), and then present our solution (Section 2.4). We conclude by highlighting and motivating a few of F(EML)'s *unusual* technical features informally (Section 2.5) prior to the more formal treatment in Section 3.

Module declarations, expressions, bodies

$$
\begin{array}{rl}
\mathit{Md} ::= & \texttt{module}\ M\ \texttt{uses}\ \overline{M}\ \texttt{=}\ \mathit{Me} \\
\mathit{Me} ::= & \{\ \overline{\mathit{Mb}}\ \}\ |\ \texttt{(}M : \mathit{Se}\texttt{)}\ \texttt{->}\ \mathit{Me}\ |\ M\texttt{(}M'\texttt{)} \\
\mathit{Mb} ::= & [\texttt{abstract}]\ \texttt{class}\ c\texttt{(}\overline{x:\tau}\texttt{)}\ [\texttt{extends}\ C\texttt{(}\overline{e}\texttt{)}]\ \texttt{of}\ \{\overline{l:\tau'=e'}\} \\
 & |\ \texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \tau' \\
 & |\ \texttt{extend fun}\ F\ P\ \texttt{=}\ e \\
 & |\ \texttt{val}\ x\ \texttt{=}\ e
\end{array}
$$

Core expressions, patterns, types

$$
\begin{array}{rl}
e ::= & \texttt{(}\overline{e}\texttt{)}\ |\ C\ \texttt{(}\overline{e}\texttt{)}\ |\ F\ |\ e\ e'\ |\ x\ |\ \hat{M}.x \\
P ::= & \texttt{(}\overline{P}\texttt{)}\ |\ C\ \{\overline{L\texttt{=}P}\}\ |\ x\ [\texttt{as}\ P]\ |\ \_ \\
\tau ::= & \texttt{(}\overline{\tau}\texttt{)}\ |\ C\ \{\overline{L:\tau}\}\ |\ \tau\ \texttt{->}\ \tau'\ |\ \texttt{bottom} \\
\tau^{\#} ::= & \texttt{(}\overline{\tau}, \tau_i^{\#}, \overline{\tau'}\texttt{)}\ |\ \texttt{\#}C\ \{\overline{L:\tau}\}\ |\ C\ \{\overline{L:\tau}, L:\tau^{\#}, \overline{L':\tau'}\}
\end{array}
$$

Signatures

$$
\begin{array}{rl}
\mathit{Sd} ::= & \texttt{signature}\ S\ \texttt{uses}\ \overline{M}\ \texttt{=}\ \mathit{Se} \\
\mathit{Se} ::= & \texttt{sig}\ \{\ \overline{\mathit{Sb}}\ \}\ |\ \texttt{(}M : \mathit{Se}\texttt{)}\ \texttt{->}\ \mathit{Se}'\ |\ S \\
\mathit{Sb} ::= & [\texttt{abstract}]\ \texttt{class}\ c\texttt{(}\overline{\tau}\texttt{)}\ [\texttt{extends}\ C]\ \texttt{of}\ \{\overline{L:\tau'}\} \\
 & |\ \texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \tau' \\
 & |\ \texttt{extend fun}\ F\ \tau \\
 & |\ \texttt{val}\ x : \tau
\end{array}
$$

Qualified names, identifiers

$$
\hat{M} ::= M\ |\ \texttt{ThisMod} \qquad C ::= \hat{M}.c \qquad F ::= \hat{M}.f \qquad L ::= \hat{M}.l
$$

$$
S, M, f, c, l, x ::= \mathit{identifier}
$$

Fig. 1. Syntax of F(EML)$^-$

2.1 Ground Modules and Declarations

We have already seen examples of ground (non-functor) modules; here, we give a more systematic description of each construct in Fig. 1. Returning to `Lang`:

```
module Lang = { abstract class Expr() of {}
                abstract fun eval:Expr -> Int }
```

This module declaration (*Md*) declares a new ground module (or *structure*) named `Lang`, having two members (*Mb*). The first member is a *fresh class* declaration for an abstract class named `Expr`, which has the trivial constructor argument `()` and the trivial representation {}. Since `Expr` specifies no superclass, it is assumed to inherit from the distinguished root class `Object`.

The second member is a *fresh function* declaration, having the type `Expr -> Int`. Note that in Fig. 1, a function's argument type must be a *marked type* $\tau^{\#}$, wherein exactly one class type is prefixed by a hash mark `#`. If no mark is present, we mark the topmost, leftmost class by default — in this case, `Expr`. We explain marked types further Section 3, but intuitively, they statically constrain future extensions so that they will not be ambiguous with each other.

Next, consider our `ConstMod` example, slightly extended:

```
module ConstMod uses Lang = {
  class Const(v_in:Int) extends Lang.Expr() of {value:Int = v_in}
  extend fun Lang.eval(Const {value=v}) = v
  val zero = Const(0) }
```

This module declares another fresh class `Const`, a *fresh method* that extends `eval`, and a *value binding* named `zero`. `Const` has a non-trivial constructor specification with one argument `v_in` of type `Int`.

`Const` extends `Lang.Expr`; the name reference must be qualified with the *module path* `Lang` because it is not a local class.[1] All module paths used in a module body must appear in the `uses` clause of the enclosing module declaration, or one of the (transitive) `uses` clauses of used modules. `Const` also invokes `Lang.Expr`'s constructor, passing an argument tuple of appropriate type (in this case, the empty tuple, but in general any tuple of expressions may appear here). Finally, `Const` defines a representation containing one *field* (in addition to any inherited fields, although here the superclass has no fields), having *label* `value`.[2] and type `Int`. This field is initialized to the value of `v_in`, which is bound in the constructor argument. As with superclass constructor arguments, field initializers may be arbitrary expressions.

The fresh method `extend fun Lang.eval` adds a case to the function `eval` in `Lang`.[3] Methods define an argument pattern P, similar in spirit to pattern matching constructs in functional languages. This method's pattern is `Const {value=v}`, which specifies that this method overrides `eval` on arguments of class `Const` (or any subclass), matching on the `value` field, and assigning that field's value to the variable `v`, which is bound in the method body expression (patterns may also be tuples `(`$\overline{P}$`)`, binders x [`as` P], or wildcards `_`). This method's body is `v`, so it returns the `v` bound during pattern matching.

Finally, a *value binding* evaluates a core language expression and binds it to a name. In the case of the `zero` binding, the expression is `Const(0)`, which applies the `Const` constructor to the single-element argument tuple `(0)`.

The syntax of core language expressions e, from left to right in Fig. 1, is as follows: tuples `(`$\overline{e}$`)`, which construct tuple values; object constructors C `(`$\overline{e}$`)`,

[1] Technically, references to local declarations and standard classes like `Object` are automatically qualified with the paths `ThisMod` and `Std` respectively.

[2] Internally, field labels are qualified by a module name; this is a technical point which, for presentation purposes, we will ignore in the rest of this paper.

[3] Note that EML, unlike many other object-oriented languages, distinguishes explicitly between *introduction* of functions (`fun` declarations) and *overriding* of a function by a method (`extend fun` declarations).

which construct a fresh value of class C by invoking its constructor with the argument tuple $(\overline{e})$; named function references F; message sends $e\ e'$, which apply e to e'; local pattern-bound variables x; or `val`-bound variables $\hat{M}.x$.

At runtime, a message send dispatches to the *globally most-specific case* among all method cases that have been defined for the invoked function. The specificity relation between method cases is defined by the subtyping relation between the patterns in their arguments (Section 3.1 gives a formal description of the dispatch semantics). The dynamic semantics of dispatch give no priority to any particular position in (the abstract syntax tree of) a method's argument pattern — i.e., dispatching is *symmetric*.

2.2 Basic Signatures and Functors

Following ML, we call a module interface a *signature*. A module definition implicitly defines a *principal signature* (which is generated automatically from the module by the type system), but F(EML) also supports explicit interfaces.

Signature body declarations *Sb* have four cases, paralleling the four basic kinds of declarations that can appear in a module. Recall our `LangSig` example:

```
signature LangSig = sig {
  abstract class Expr() of {}
  abstract fun eval:Expr -> Int }
```

This signature has a *class signature* and a *function signature*. Class signatures indicate whether the class is abstract, give the class name and constructor argument types, the class's superclass, and a list of field names and types. Function signatures simply give the function name and type.

The following signature is equivalent to the principal signature generated for the `ConstMod` from the previous section:

```
sig { class Const(Int) extends Lang.Expr of {value:Int}
      extend fun Lang.eval(Const {value:Int})
      val zero:Const }
```

`Const`'s class signature shows that it is a concrete class with a constructor of type `Int` and a representation with a single field. A *method signature* `extend fun` $F\ \tau$ names the extended function F (here, `Lang.eval`) and the argument type τ at which the method overrides the function (here, `Const {value:Int}`). A *value signature* `val` $x : \tau$ gives the name and type of the bound name.

For this paper's purposes, the most important use of explicit signatures is to describe the arguments of parameterized modules. Recall our `MakePlus` example:

```
module MakePlus = (L:LangSig) -> {
  class Plus(l_in:Int, r_in:Int) extends L.Expr()
      of {left:Int = l_in, right:Int = r_in}
  extend fun L.eval(Plus {left=l, right=r}) = L.eval(l)+L.eval(r)}
```

A parameterized module expression begins with a parameter definition $(M : Se)$, where M is the formal parameter name and Se is a signature expression.

In `MakePlus`, the parameter definition is `(L:LangSig)`; `L` is the formal parameter name, and `LangSig` is the formal parameter's signature. The parameter declaration is followed by an arrow `->` and a module expression. As one might expect, in the module body, declarations specified by the argument signature are available as names qualified by the formal parameter name.

A functor application $M(M')$ applies the module named by M to the argument M'. For presentation, we follow Leroy [23] and Harper et al. [18], and limit functor application expressions to named modules; a practical implementation would perform "lambda lifting" to allow applications of arbitrary functors to arbitrary argument modules. Informally, the application $M(M')$ copies the body of M to a new module expression Me' and substitutes M' for the formal name in Me'. For example, `MakePlus(Lang)` generates the following module expression:

```
{ class Plus(l_in:Int, r_in:Int) extends Lang.Expr()
      of {left:Int = l_in, right:Int = r_in}
  extend fun Lang.eval(Plus {left=l, right=r})
      = Lang.eval(l) + Lang.eval(r) }
```

2.3 Problem: Limited Reuse

To explore the limitations of this language, we now examine a more complex example. Consider Fig. 2. The signature `Algebra` defines an abstract class `Expr` with two concrete direct subclasses, `Plus` and `Times`. The `MakeDist` functor provides `dist`, which distributes occurrences of `Times` over `Plus`. Notice that this operation defines four cases. The first case is a default, which leaves other `Expr` forms unchanged. One case each is defined for a root expression of `Times` with `Plus` on the left subtree, the right subtree, and both subtrees.

```
signature Algebra = sig {
  abstract class Expr() of {}
  class Plus(Expr, Expr) extends Expr of {left:Expr, right:Expr}
  class Times(Expr, Expr) extends Expr of {left:Expr, right:Expr} }

module MakeDist = (A:Algebra) -> {
  fun dist:A.Expr -> A.Expr
  extend fun dist(e as A.Expr) = e
  extend fun dist(A.Times { left=(A.Plus {left=l,right=r}), right=r_outer }) =
      A.Plus(A.Times(l, r_outer), A.Times(r, r_outer))
  extend fun dist(A.Times { left=l_outer, right=(A.Plus {left=l,right=r}) }) =
      A.Plus(A.Times(l, l_outer), A.Times(r, l_outer))
  extend fun dist(A.Times { left=(A.Plus {left=l,right=r}),
                            right=(r_outer as A.Plus {left=_, right=_}) }) =
      A.Plus(dist(A.Times(l, r_outer)), dist(A.Times(r, r_outer))) }
```

Fig. 2. The `Algebra` signature and `MakeDist` functor

Now, recall that we would like to reuse this extension in many contexts. However, consider the following reasonable definition of an "algebra". First, use `Lang` and `PlusMod` as defined in Section 1; finally, define a third module:

```
module TimesMod uses Lang = {
  abstract class DistOp extends Lang.Expr() of {}
  class OpTimes(l_in:Lang.Expr, r_in:Lang.Expr)
    extends DistOp() of {left:Lang.Expr=l_in, right:Lang.Expr=r_in}
  extend fun eval(OpTimes {...}) = ... }
```

Considered together, `Lang`, `PlusMod`, and `TimesMod` contain all the pieces needed for an "algebra", yet they do not constitute an `Algebra`, for several reasons:

– First, and most obviously, this functor assumes a particular prior modularization strategy. `Algebra` is the signature of a single module, but in this case the client chose to factor the declarations into separate modules.
– Second, `Algebra` requires a declaration named `Times`, not `OpTimes`.
– Third, `Algebra` requires classes that *directly* extend `Expr`. `OpTimes` *transitively* extends `Expr`, so again it would be incompatible with `Algebra`. More generally, one might wish to specify direct subclassing, strict subclassing, inequality, and other relations; for example, inequality constraints might help prove the non-ambiguity of two methods. However, the language presented so far cannot express these constraints.

Finally, we note briefly one further problem that is not obvious from the examples' syntax, but arises in typechecking. EML did not permit *signature subsumption*; an EML module could be incompatible with a signature having fewer declarations, or less-precise information. Hence, even if we bundled all the declarations in one module and allowed `Algebra` to accept a transitive subclass of `Expr` for `Times`, the presence of the `DistOp` class or the `eval` function would render the module incompatible with `Algebra`. Clearly, this greatly reduces the utility of `MakeDist`. This was not merely an oversight in the EML design; as we shall see in Section 3, signature subsumption turns out to be rather tricky.

2.4 Solution: An Enriched Language

The limitations described in the previous section share a common theme: the argument signature makes the functor depend on *inessential details* of the extended code. Our solution is to enrich the language so as to remove these dependencies — either by generalizing the signature language, or by letting the programmer "adapt" a potential argument to the required signature.

The enriched grammar is shown in Fig. 3. Note that we extend the syntax of module bodies, but replace the syntax of signatures; the signatures in Fig. 1 are legal, but F(EML) rewrites them internally into the form shown.

There are three general kinds of changes. First, we add *alias declarations*; second, we add *relation constraints* to signatures; third, we enable *selective sealing* of class and function declarations. In the rest of this subsection, we discuss these changes in turn, and then revisit our `MakeDist` example.

Alias Declarations. Alias declarations define a new module declaration that aliases an existing declaration rather than creating a new one. An *alias class* `alias class` c `=` C defines a module member named c that refers to the existing

Module expressions and bodies

$Mb ::= \ldots$
$\quad\mid$ `alias class` c `=` C
$\quad\mid$ `alias fun` f `=` F
$\quad\mid$ `alias extend fun` $F\ \tau$ `in` $\hat{M}$

Signatures

$Se ::=$ `sig` $\{\ \overline{Sb}$ `fresh` ϕ `where` $\rho\ \}\mid$ `(`$M : Se$`) ->` $Se' \mid S$
$Sb ::=$ [`closed`] `class` c [`(`$\overline{\tau}$`)`] `of` $\{\overline{L : \tau}\}$ [`abstract on` $\overline{F}$]
$\quad\mid$ `fun` $f : \tau^{\#}$ `->` τ' `open below` τ''
$\quad\mid$ `extend fun` $F\ \tau$
$\quad\mid$ `val` $x : \tau$

$\phi ::= \overline{y} \qquad \rho ::= \overline{r} \qquad y ::= c \mid f \mid q$
$q ::= F.\tau \qquad Q ::= \hat{M}.q$
$r ::= C\ \mathcal{R}_C\ C' \mid F\ \mathcal{R}_F\ F' \mid Q\ \mathcal{R}_Q\ Q'$

Class, function, method, and type relations

$\mathcal{R}_C ::= \top \mid \bot \mid\ \leq\ \mid\ \neq\ \mid\ \not\Delta\ \mid\ <\ \mid\ <^0\ \mid\ <^1\ \mid\ <^2\ \mid \ldots$
$\mathcal{R}_F ::= \ =\ \mid\ \neq \qquad \mathcal{R}_Q ::= \ =\ \mid\ \neq$
$\mathcal{R}_\tau ::= \top \mid \bot \mid\ \leq\ \mid\ \neq\ \mid\ \not\Delta\ \mid\ <\ \mid\ =$

Fig. 3. Syntax of F(Eml) (diff from Fig. 1)

class C. An *alias function* `alias fun` f `=` F defines a member named f that refers to F. An *alias method* `alias extend fun` $F\ \tau$ `in` $\hat{M}$ defines an alias for the method found in module $\hat{M}$ that extends the function F on type τ. The need for function and class aliases is relatively straightforward, as we shall see shortly in Section 2.4; however, the need for method aliases is somewhat technical, and we postpone further discussion of them to Section 3.

Relation Constraints. There are two kinds of relation information: *binary relations* ρ, and *freshness* information ϕ.

Binary relations describe the relationships between two declarations. Classes have the richest language of relations, including general subclassing $\leq$, inequality $\neq$, disjointness $\not\Delta$ (sharing no common subclasses; in ASCII we write `disjoint`), strict subclassing $<$, and k-level subclassing $<^k$ (for $k \in \{0, 1, 2, \ldots\}$). $<^0$ is reflexive subclassing, i.e. equality, and can be written $=$; $<^1$ is direct subclassing, and can be written `extends`. $\top$ and $\bot$ denote "unknown" and "impossible" relations respectively; these are a technical convenience permitting certain rules to be stated more concisely, and we will not discuss them further in this paper.

Class relations serve two purposes. First, they enrich the language of constraints that a programmer can describe in a signature. Second, they permit the programmer to track the aliasing that results from the use of alias classes. It turns out that typechecking often requires knowledge that two classes, for example, are *not* aliases for each other. This second rationale also applies to functions and methods, so we require relations for these as well; function and method relations include only aliasing ($=$) or non-aliasing ($\neq$).

It is impossible for a signature to anticipate all the must-not-alias relationships that future clients might need. Therefore, F(EML) also tracks freshness information: when a name appears in the `fresh` ϕ portion of a signature, it indicates that the name (which must be bound by the enclosing signature) describes a fresh declaration and not an alias declaration. When a name appears in a `fresh` clause, its referent therefore is known not to alias any other `fresh` name, without requiring an explicit $\neq$ relation between the two names.

Selective Sealing. Class and function signatures in F(EML) have additional clauses, which restrict how they may be used. These restrictions play a key role in signature subsumption; for the moment we explain only their informal meaning, postponing the details of *how* they make subsumption safe to Section 3.

Class signatures change in several ways. First, they may be marked `closed`, indicating that clients may not extend them *through this signature* (although other aliases of the underlying class may not be marked `closed`, so `closed` is not equivalent to Java's `final`). Second, class constructors are optional in signatures; when the constructor argument type is absent, the constructor is hidden, and the class may not be instantiated. Third, class signatures may have an `abstract on` clause, naming a list of functions that need an implementing case for this class. Note that functions no longer carry an optional `abstract` flag; `abstract on` replaces `abstract` on the functions.

Function signatures gain one piece: an `open below` clause, which names the *extension type* of that function. If a function has the signature `fun` $f : \tau^{\#}$ `->` τ' `open below` τ'', then methods outside of f's module can only extend f on τ if τ is a strict subtype of τ'' (again, other aliases of f may have a more permissive extension type).

A revised `Algebra`. Fig. 4 gives an alternative definition of the `Algebra` signature, and a module that remodularizes the declarations we defined previously to fit this signature.

```
signature Algebra = sig {
  closed class Expr of {}
  closed class Plus(Expr, Expr) of {left:Expr, right:Expr}
  closed class Times(Expr, Expr) of {left:Expr, right:Expr}
  fresh .
  where Plus < Expr, Times < Expr, Plus != Times }

module LangAlgebra uses Lang = {
  alias class Expr = Lang.Expr
  alias class Plus = PlusMod.Plus
  alias class Times = TimesMod.OpTimes }
```

Fig. 4. Revision of `Algebra` from Fig. 2, and a module satisfying this signature

If we use this revised `Algebra`, then both the functor definition `MakeDist` and the functor application `MakeDist(LangAlgebra)` will typecheck. Our fix uses all three of the extensions described previously. First, we use alias declarations to "repackage" existing declarations so they can be extended by the functor. Second, we use generalized class relations to specify exactly the relations needed for `MakeDist` to conclude that no two cases of `dist` are ambiguous with each other. Third, we seal all the classes in the signature, marking them `closed`, which constitutes a "promise" that `MakeDist`'s body will not subclass any of these classes. This promise is necessary to make `Algebra` compatible with `LangAlgebra`, for a somewhat subtle reason. Consider the signature of `LangAlgebra.Expr`:

```
class Expr() of {} abstract on Lang.eval  // (1)
```

because its source class (`Lang.Expr`) is abstract on `eval`. But `(1)` is not compatible with (not a valid subsignature of)

```
class Expr() of {}                        // (2)
```

A hypothetical concrete subclass of `(2)` would not need to implement a case for `eval`, whereas any valid concrete subclass of `(1)` *must* implement a case for `eval`. Hence, valid clients of `(2)` are not necessarily valid clients of `(1)`.

However, a `closed` class with a hidden constructor cannot be subclassed or instantiated, so its signature may freely "forget" about its abstract functions. Therefore, the signature `(1)` *is* a valid subsignature of the following signature (the absence of the tuple of constructor argument types signifies that the constructor cannot be called):

```
closed class Expr of {}                   // (3)
```

because no client can use `(3)` inconsistently with legal uses of `(1)`.

2.5 Discussion: Unusual Features, and Their Motivation

Before diving into our semantics, we highlight a few forces, arising from certain design choices, which motivate specific unusual supporting technical features.

First, as previously noted, combining extensibility and symmetric multiple dispatch raises the problem of ambiguous function implementation. As a result, our type relations include *inequality*, *disjointness*, and *strict subtyping*, which can be used to deduce non-ambiguity of methods. For example, methods that override a function on disjoint argument types can never apply to the same argument, and hence cannot be ambiguous. This is unusual because most type systems either do not care about distinctness (ML signatures, for example, transmit type equalities, but not inequalities) or treat inequality only implicitly.

Second, because F(EML) aims to support modular programming, we cannot require programmers to list all useful inequality constraints for every class — for any class C, it may be useful to know that C is distinct from classes that are not visible or not yet defined at the point of C's declaration. Therefore, F(EML) explicitly tracks the *freshness* of classes and other declarations, and deduces, for example, that two fresh class declarations always name distinct classes. This is unusual because, again, most type systems treat freshness only implicitly.

Third, because F(EML) permits class and function extension from outside the original declaration, F(EML) requires fine-grained *selective sealing* to restrict the extensibility of declarations. As we've shown in the previous section, sealing is crucial to signature subsumption. This is unusual because most languages either lack extensibility, or conflate a construct's visibility with permission to extend it, or permit coarse-grained limits on extensibility to express programmer intent (e.g., Java's `final`) but never require it for soundness.

3 Semantics and Typechecking

We have formalized the essence of F(EML) in a reduced language called Mini-F(EML). Actually, the language presented thus far *is* Mini-F(EML), except for the differences in Fig. 5. This grammar also specifies which subsets of module and core expressions are module language and core language values.

Module values, bodies

$$\begin{aligned}
Mv &::= \{\ \overline{Mb}\ \} \mid \texttt{(}M : Sv\texttt{)}\ \texttt{->}\ Me \\
Mb &::= \ldots \text{((alias) classes and (alias) functions as before)} \\
&\mid \texttt{extend fun}\ F\ \texttt{with}\ q\ P\ \texttt{=}\ e \\
&\mid \texttt{alias extend fun}\ F\ \texttt{with}\ q\ \texttt{=}\ Q \\
&\mid \cancel{\texttt{val}\ x\ \texttt{=}\ e}
\end{aligned}$$

Signature values, bodies

$$\begin{aligned}
Sv &::= \texttt{sig}\ \{\ \overline{Sb}\ \texttt{fresh}\ \phi\ \texttt{where}\ \rho\ \} \mid \texttt{(}M : Sv\texttt{)}\ \texttt{->}\ Sv' \\
Sb &::= [\texttt{closed}]\ \texttt{class}\ c\ [\texttt{(}\overline{\tau}\texttt{)}]\ \texttt{of}\ \{\overline{L : \tau}\}\ [\texttt{abstract on}\ \overline{F}] \\
&\mid \texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \tau\ \texttt{open below}\ \tau^{P} \\
&\mid \texttt{extend fun}\ F\ \texttt{with}\ q\ \tau^{P}
\end{aligned}$$

Core patterns, types, expressions, values

$$\begin{aligned}
P &::= \texttt{(}\overline{P}\texttt{)} \mid C\ \{\overline{L \texttt{=} P}\} \mid x\ \texttt{as}\ P \mid \_ \\
\tau &::= \texttt{(}\overline{\tau}\texttt{)} \mid C\ \{\} \mid \tau\ \texttt{->}\ \tau' \\
\tau^{P} &::= \texttt{(}\overline{\tau^{P}}\texttt{)} \mid C\ \{\overline{L : \tau^{P}}\} \mid \tau\ \texttt{->}\ \tau' \mid \texttt{bottom} \\
\tau^{\#} &::= \texttt{(}\overline{\tau}, \tau^{\#}, \overline{\tau'}\texttt{)} \mid \texttt{\#}C\ \{\} \\
e &::= \texttt{(}\overline{e}\texttt{)} \mid C\ \{\overline{L \texttt{=} e}\} \mid C\ \texttt{(}\overline{e}\texttt{)} \mid F \mid e\ e' \mid x \\
v &::= \texttt{(}\overline{v}\texttt{)} \mid C\ \{\overline{L \texttt{=} v}\} \mid F
\end{aligned}$$

Method names (bare and qualified); qualified names; fresh names

$$q ::= \mathit{identifier} \qquad Q ::= \hat{M}.q \qquad Y ::= C \mid F \mid Q \qquad \phi ::= \overline{Y}$$

Fig. 5. Syntax of Mini-F(EML) (diff from Fig. 3)

We summarize the changes (made for technical convenience) as follows. First, in F(EML), methods do not have names, and are referenced by profile only; in Mini-F(EML), for convenience, each method is named by an identifier q. Second, we omit `val` bindings, as these can be simulated by functions with a dummy argument and exactly one case. Third, we eliminate named signature expressions, and require signatures to be expanded inline. Fourth, the `as` P clause in binding patterns is mandatory. Fifth, we separate types into two syntactic kinds: we restrict first-class types τ to tuples, functions, and class types, with tracking of field types, whereas τ^P (the type of a pattern) may include more precise information about fields. Restricting the type syntax in this manner simplifies our proof strategy, while still requiring us to deal with the essence of the ambiguity and incompleteness problems arising from extensibility and multiple dispatch. Sixth, lists of fresh names ϕ are fully qualified, and may include method names. Lastly, we include instances $C\ \{\overline{L = e}\}$ in the grammar of expressions; these are not available at source level, but arise when defining small-step reduction.

The challenge in designing a type system that is both useful and sound arises from the combination of F(Eml)'s uniform, symmetric dispatching model and its powerful extensibility constructs. In Section 3.1, we elaborate on the dynamic semantics of dispatching, focusing on how evaluation can go wrong. In Section 3.2, we describe typechecking. Section 3.3 states the soundness theorems. The full formalization of F(Eml) will appear in our companion report [22].

3.1 Linkage and Evaluation

A Mini-F(Eml) program consists of a list of module declarations $\overline{Md}$, followed by a "main expression" e. Execution has two phases: first, $\overline{Md}$ is linked to produce a *dynamic context* Δ, and then e is evaluated in the context of Δ.

Δ is a finite map $\overline{M \mapsto Me}$ from module names to ("compiled") module expressions. Fig. 6 shows a subset of the linkage rules. $[\overline{x \mapsto v}]e$ denotes the substitution of each v_i for its respective x_i in an expression e.

$$\boxed{\Delta \vdash \overline{Md} \Downarrow^* \Delta'}$$

$$\frac{}{\Delta \vdash \epsilon \Downarrow^* \Delta}\ (\textsc{Link-Empty})$$

$$\frac{\Delta \vdash \overline{Md} \Downarrow^* \Delta' \qquad \Delta' \vdash M = Me \Downarrow Mv}{\Delta' \vdash \overline{Md}; (\texttt{module}\ M\ \texttt{uses}\ \overline{M}\ \texttt{=}\ Me) \Downarrow^* \Delta', M \mapsto Mv}\ (\textsc{Link-Mod})$$

$$\boxed{\Delta \vdash M = Me \Downarrow Mv}$$

$$\frac{}{\Delta \vdash M = (M' : Sv)\ \texttt{->}\ Me \Downarrow (M' : Sv)\ \texttt{->}\ Me}\ (\textsc{L-Funct})$$

$$\frac{\Delta, \texttt{ThisMod} \mapsto \{\ \overline{Mb}\ \} \vdash \mathrm{dealias}(\{\ \overline{Mb}\ \}) = \{\ \overline{Mb'}\ \}}{\Delta \vdash M = \{\ \overline{Mb}\ \} \Downarrow [\texttt{ThisMod} \mapsto M]\{\ \overline{Mb'}\ \}}\ (\textsc{L-Struct})$$

$$\frac{\Delta(M) = (M_1 : Sv)\ \texttt{->}\ Me_1 \qquad Me' = [M_1 \mapsto M']Me_1 \qquad \Delta \vdash M_0 = Me' \Downarrow Mv'}{\Delta \vdash M_0 = M(M') \Downarrow Mv'}\ (\textsc{L-App})$$

Fig. 6. Selected linkage rules

Linkage performs three operations. First, it expands functor applications into module values (L-App); since we restrict applications to named module expressions, we simply substitute the actual argument name for the formal argument name in the body, and then link the body if necessary. Second, for ground modules, linkage eliminates references to alias declarations; we omit the definition of the $\Delta \vdash \mathrm{dealias}(Me)$ judgment, but informally, for every name that refers to an alias, it (transitively) "chases aliases" until it finds a fresh declaration, and replaces the reference to the alias with a reference to that fresh declaration source. Third, for structures, linkage rewrites self-references via `ThisMod` to refer to the module's linked name.

Fig. 7 gives the (small-step, operational) semantics of core expression evaluation and auxiliary judgments. Execution uses the dynamic context Δ, but otherwise these rules are exactly analogous to those for EML [28]. We include fairly complete rules here for reference, but we will only discuss those parts absolutely necessary to explain the typechecking problems that follow.

Note that some syntactic sequences with an overbar have a superscripted range, e.g. $\overline{v}^{1..n}$; this is shorthand for $v_1, \ldots, v_n$. We use the set membership operator $\in$ on syntactic sequences, e.g. $Mb \in \overline{Mb}$ indicates that the Mb is an element of the sequence $\overline{Mb}$. We write $Mb \in \Delta(M)$ as shorthand for $(\Delta(M) = \{\ \overline{Mb}\ \}) \wedge (Mb \in \overline{Mb})$. We use a long double arrow $\Longrightarrow$ for logical implication, to distinguish it from $\rightarrow$ (for small-step evaluation) and $\Rightarrow$ (for signature generation, which we will see in in Section 3.2). Superscripted brackets $[\ ]^k$ around a part of the rule indicate that those parts are optional, but either all bracketed portions superscripted with the same k must be present, or all must be absent.

Evaluation uses the dynamic subpattern and subclass relations, which are given in Fig. 7. Note that these judgments are entirely distinct from the *static* relation deduction that we describe later.

Evaluation can get stuck in two cases. First, the program could attempt to construct a class marked `abstract`; call this an *abstract instantiation* error. Second, the program could send a message for which $\Delta \vdash \text{lookup}(F, v) = \langle q, B, e \rangle$ is not derivable, which can occur in two ways. Informally, the premises of LOOKUP specify that there must exist some (fresh) method in Δ such that (1) its pattern P matches the argument value, and (2) P is strictly more specific than the patterns of all other matching methods in Δ. Therefore, this rule can fail either if there are *zero* applicable methods, or if there are *multiple* applicable methods, none of which is strictly more specific than all the others. The former case is a *message not understood* error; the latter case is an *ambiguous message* error.

3.2 Typechecking

In this section, we first describe the general structure of typechecking; then, in later subsections, we describe in more detail those portions of the semantics most directly relevant to supporting parameterized modules. Fig. 8 summarizes the major static judgment forms.

A signature context Γ is a finite map from module names $\hat{M}$ to signature values Sv; the dependency context $\mathcal{D}$ is a finite map from module names M to depended-upon module names $\overline{M}$. The relation context $\mathcal{K}$ is a pair $\langle \phi, \rho \rangle$ where ϕ is a set of fresh names and ρ is a set of binary relations. Auxiliary rules used by these judgments will also use the contexts β (mapping pattern-bound variables x to types τ) and R (mapping class names C to representations $\{\overline{L : \tau}\}$).

The top-level typing judgments (the first two lines in Fig. 8) essentially typecheck each module declaration in $\overline{Md}$ from left to right (i.e., they construct Γ

$$\frac{\Delta \vdash e_1 \rightarrow e_2}{\Delta \vdash e_1\ e' \rightarrow e_2\ e'}\ (\textsc{E-App-L}) \qquad \frac{\Delta \vdash e_1 \rightarrow e_2}{\Delta \vdash v\ e_1 \rightarrow v\ e_2}\ (\textsc{E-App-R}) \qquad \boxed{\Delta \vdash e \rightarrow e'}$$

$$\frac{\begin{array}{c}\Delta \vdash \text{concrete}(C) \\ \Delta \vdash \text{rep}(C\ \texttt{(}\overline{v}\texttt{)}) = \{\overline{L = e'}\}\end{array}}{\Delta \vdash C\ \texttt{(}\overline{v}\texttt{)} \rightarrow C\ \{\overline{L = e'}\}}\ (\textsc{E-New}) \qquad \frac{\Delta \vdash e_1 \rightarrow e_2}{\begin{array}{c}\Delta \vdash C\ \{\overline{L = v}, L = e_1, \overline{L' = e'}\} \\ \rightarrow C\ \{\overline{L = v}, L = e_2, \overline{L' = e'}\}\end{array}}\ (\textsc{E-Rep})$$

$$\frac{\Delta \vdash e_1 \rightarrow e_2}{\Delta \vdash \texttt{(}\overline{v}, e_1, \overline{e'}\texttt{)} \rightarrow \texttt{(}\overline{v}, e_2, \overline{e'}\texttt{)}}\ (\textsc{E-Tuple}) \qquad \frac{\Delta \vdash \text{lookup}(F, v) = \langle q, B, e\rangle}{\Delta \vdash F\ v \rightarrow [B]e}\ (\textsc{E-App-Red})$$

$$\boxed{\Delta \vdash \text{concrete}(C)}$$

$$\frac{(\texttt{class}\ c\ \_\ [\texttt{extends}\ \_\ \_]\ \texttt{of}\ \{\_\}) \in \Delta(M)}{\Delta \vdash \text{concrete}(M.c)}\ (\textsc{Concrete})$$

$$\boxed{\Delta \vdash \text{rep}(C\ \texttt{(}\overline{v}\texttt{)}) = \{\overline{L = e}\}}$$

$$\frac{\begin{array}{c}([\texttt{abstract}]\ \texttt{class}\ c\ \texttt{(}\overline{x : \tau}^{1..n}\texttt{)}\ [\texttt{extends}\ C\texttt{(}\overline{e''}\texttt{)}]^1 \\ \texttt{of}\ \{\overline{L' : \tau' = e'}\}) \in \Delta(M) \\ [\Delta \vdash \text{rep}(C\ \texttt{(}[\overline{x \mapsto v}^{1..n}]\overline{e''}\texttt{)}) = \{\overline{L = e'''}\}]^1\end{array}}{\Delta \vdash \text{rep}(M.c\ \texttt{(}\overline{v}^{1..n}\texttt{)}) = \{[\overline{L = e'''}]^1, \overline{M.l' = [\overline{x \mapsto v}^{1..n}]e'}\}}\ (\textsc{Rep})$$

$$\boxed{\Delta \vdash \text{lookup}(F, v) = \langle q, B, e\rangle}$$

$$\frac{\begin{array}{c}\Delta \vdash \text{match}(P, v) = B \\ (\texttt{extend fun}\ F\ \texttt{with}\ q\ P\ \texttt{=}\ e) \in \Delta(M) \\ (\forall M' \in \text{dom}(\Delta).\forall(\texttt{extend fun}\ F\ \texttt{with}\ q'\ P'\ \texttt{=}\ e') \in \Delta(M'). \\ ((\Delta \vdash \text{match}(P', v) = B') \wedge (M.q \neq M'.q')) \implies ((\Delta \vdash P \leq P') \wedge \neg(\Delta \vdash P' \leq P)))\end{array}}{\Delta \vdash \text{lookup}(F, v) = \langle q, B, e\rangle}\ (\textsc{Lookup})$$

$$\boxed{\Delta \vdash \text{match}(P, v) = B}$$

$$\frac{\Delta \vdash \text{match}(P, v) = B}{\Delta \vdash \text{match}(x\ \texttt{as}\ P, v) = x \mapsto v, B}\ (\textsc{Match-Bind}) \qquad \frac{}{\Delta \vdash \text{match}(\_, v) = \epsilon}\ (\textsc{Match-Wild})$$

$$\frac{\Delta \vdash C' \leq C \qquad \forall_{i=1}^{n}.(\Delta \vdash \text{match}(P_i, v_i) = B_i)}{\Delta \vdash \text{match}(C\ \{\overline{L = P}^{1..n}\}, C'\ \{\overline{L = v}^{1..n}, \overline{L' = v'}\}) = \cup_1^n B}\ (\textsc{Match-Class})$$

$$\frac{\forall_{i=1}^{n}.(\Delta \vdash \text{match}(P_i, v_i) = B_i)}{\Delta \vdash \text{match}(\texttt{(}\overline{P}^{1..n}\texttt{)}, \texttt{(}\overline{v}^{1..n}\texttt{)}) = \cup_1^n B}\ (\textsc{Match-Tuple})$$

$$\boxed{\Delta \vdash P \leq P'}$$

$$\frac{\Delta \vdash P \leq P'}{\Delta \vdash (x\ \texttt{as}\ P) \leq P'}\ (\textsc{PSub-Bind-L}) \qquad \frac{\Delta \vdash P \leq P'}{\Delta \vdash P \leq (x\ \texttt{as}\ P')}\ (\textsc{PSub-Bind-R}) \qquad \frac{}{\Delta \vdash P \leq \_}\ (\textsc{PSub-Wild})$$

$$\frac{\forall_{i=1}^{n}.\Delta \vdash P_i \leq P_i'}{\Delta \vdash \texttt{(}\overline{P}^{1..n}\texttt{)} \leq \texttt{(}\overline{P'}^{1..n}\texttt{)}}\ (\textsc{PSub-Tuple}) \qquad \frac{\Delta \vdash C \leq C' \qquad \forall_1^n i.\Delta \vdash P_i \leq P_i''}{\Delta \vdash C\ \{\overline{L = P}^{1..n}, \overline{L' = P'}\} \leq C'\ \{\overline{L = P''}^{1..n}\}}\ (\textsc{PSub-Class})$$

$$\boxed{\Delta \vdash C \leq C'}$$

$$\frac{([\texttt{abstract}]\ \texttt{class}\ c\ \_\ \texttt{extends}\ C\ \_\ \texttt{of}\ \{\_\}) \in \Delta(M)}{\Delta \vdash M.c \leq C}\ (\textsc{CSub-Ext})$$

Fig. 7. Dynamic semantics: Evaluation and auxiliary rules

$\Gamma, \mathcal{D} \vdash \overline{Md} \Rightarrow^{*} \Gamma', \mathcal{D}'$	Program typechecking
$\Gamma, \mathcal{D} \vdash Md \Rightarrow M : \langle Sv, \overline{M'} \rangle$	Module declaration typechecking
$\Gamma; \overline{M} \vdash Me : Sv$	Module principal signatures
$\Gamma, \overline{M} \vdash Sv$ OK arg	OK functor argument signature
$\text{declRels}(\overline{Mb}) = \mathcal{K}$	Relation context formation
$\langle \Gamma, \mathcal{K}, \overline{Mb} \rangle \vdash Mb : Sb$	Signature of a module body decl
$\langle \Gamma, \mathcal{K}, \overline{Mb} \rangle \vdash Y : \langle \hat{M}, Sb \rangle$	Lookup or compute sig for a name
$\Gamma, \overline{M} \vdash Mb$ OK in $\overline{Sb}$	Module body decl well-formedness
$\Gamma \vdash Sv \leq Sv'$	Signature subsumption
$\Gamma, \mathcal{K}, \overline{Sb} \vdash Sb \leq Sb'$	Sig body decl subsumption
$\Gamma, \mathcal{K}, \overline{Sb} \vdash Sb$ droppable	Sig body width subsumption
$\mathcal{K} \vdash C_1 \ \mathcal{R}_C \ C_2$	Class relation deduction
$\mathcal{K} \vdash F_1 \ \mathcal{R}_F \ F_2$	Function relation deduction
$\mathcal{K} \vdash Q_1 \ \mathcal{R}_Q \ Q_2$	Method relation deduction
$\mathcal{K} \vdash \tau_1 \ \mathcal{R}_\tau \ \tau_2$	Type relation deduction
$\Gamma, \mathcal{K}, \beta \vdash e : \tau$	Expression typing
$\mathcal{K}, R \vdash \text{ptype}(P, \tau) = \langle \tau^P, \beta \rangle$	Type and bindings of a pattern
$\langle \Gamma, \overline{Mb} \rangle \vdash \text{rep}(C) = \{\overline{L : \tau}\}$	Class representation lookup

Fig. 8. Static semantics: Selected judgment forms

and $\mathcal{D}$ with a left-to-right fold on the module declaration list), so we skip directly to the "meat" of module expression typechecking, shown in Fig. 9. $DN(\overline{Mb})$ is an auxiliary function that extracts the set of class, function, and method names introduced in $\overline{Mb}$. There are three cases for module expression typechecking: structures, functors, and functor applications.

For structures, informally, the premises of Mod-Struct specify that: (line 1) the module's declared names must be unique; (line 2) we extract a "relation context" $\mathcal{K} = \langle \phi, \rho \rangle$ from the members $\overline{Mb}$, and a principal signature can be generated for $\overline{Mb}$; (lines 3-4) in the context enriched by the relation and declaration signatures, each Mb is well-formed.

For functors, we typecheck the body in the context extended with the formal argument's signature. Informally, the Sv OK arg judgment checks that the `fresh` ϕ clause in Sv is empty, since declarations in functor arguments are never fresh (declarations in a functor formal argument are always potentially aliases).

For functor applications, we check that an alias of the actual argument's signature would be subsumed by the formal argument signature. (Informally, the aliasOf function, whose definition we omit, erases freshness information and adds equality relations between declarations in the actual and formal parameters.) We then substitute the actual argument name for the formal name in the signature body. Notice that we do not need to typecheck the functor body again.

Recall the major technical innovations that F(Eml) adds relative to Eml: generalized relations, alias declarations, and a non-trivial definition of signature subsumption. Before describing the mechanics of these features, we must first show how signatures are constructed, and summarize certain implementation restrictions inherited from Eml; we do this in the next two subsections. Then,

$$\boxed{\mathit{\Gamma}; \overline{M} \vdash Me : Sv}$$

$$\frac{\begin{array}{c}\forall_1^n i.\mathrm{DN}(Mb_i) \cap \mathrm{DN}(\overline{Mb}^{1..(i-1)}; \overline{Mb}^{(i+1)..n}) = \emptyset \\ \mathrm{declRels}(\overline{Mb}^{1..n}) = \langle \phi, \rho \rangle \qquad \forall_1^n i.\langle \mathit{\Gamma}, \langle \phi, \rho \rangle, \overline{Mb} \rangle \vdash Mb_i : Sb_i \\ \mathit{\Gamma}' = \mathit{\Gamma}, \texttt{ThisMod} \mapsto (\texttt{sig}\ \{\ \overline{Sb}^{1..n}\ \texttt{fresh}\ \phi\ \texttt{where}\ \rho\ \}) \\ \forall Mb_i \in \overline{Mb}^{1..n}.\mathit{\Gamma}' \vdash Mb_i \text{ OK in } \overline{Sb}^{1..n}\end{array}}{\mathit{\Gamma}; \overline{M} \vdash \{\ \overline{Mb}^{1..n}\ \} : \texttt{sig}\ \{\ \overline{Sb}^{1..n}\ \texttt{fresh}\ \phi\ \texttt{where}\ \rho\ \}}\ (\textsc{Mod-Struct})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}; \overline{M} \vdash Sv \text{ OK arg} \\ (\mathit{\Gamma}, M \mapsto [\texttt{ThisMod} \mapsto M]Sv); (\overline{M}, M) \vdash Me : Sv'\end{array}}{\mathit{\Gamma}; \overline{M} \vdash (\texttt{(}M : Sv\texttt{)}\ \texttt{->}\ Me) : (\texttt{(}M : Sv\texttt{)}\ \texttt{->}\ Sv')}\ (\textsc{Mod-Funct})$$

$$\frac{\begin{array}{c}\mathit{\Gamma}(M_1) = \texttt{(}M : Sv_1\texttt{)}\ \texttt{->}\ Sv_1' \qquad \mathit{\Gamma}(M_2) = Sv_2 \\ \mathit{\Gamma} \vdash \mathrm{aliasOf}(Sv_2, M_2) \leq Sv_1\end{array}}{\mathit{\Gamma}; \overline{M} \vdash M_1\texttt{(}M_2\texttt{)} : [M \mapsto M_2]Sv_1'}\ (\textsc{Mod-App})$$

Fig. 9. Static semantics: Module typechecking

we describe how typechecking must be adjusted to accommodate aliases and generalized relations. Finally, we summarize our rules for signature subsumption.

Building Signatures. Fig. 10 shows selected rules for generating the signatures of module body declarations, and the extraction of initial relation information: fresh declarations generate an element of ϕ; alias declarations generate equality relations; and a subclass generates a direct subclassing ($<^1$) relation.

Function signatures (S-Fun) are trivial; the auxiliary function unmark($\tau^{\#}$), whose definition we omit, simply erases the hash mark from a marked type.

To generate a method signature (S-Method), we first compute a finite map R from all visible class names C to representation types $\{\overline{L : \tau}\}$ (informally, the reps function iterates over all classes in $\mathit{\Gamma}$ and $\overline{Mb}$, and builds the mapping by accumulating field lists). Then, we compute the type of the argument pattern. Lastly, we sanity-check that the function to be extended exists. Note that this last check uses the judgment for signature lookup *or* computation from Fig. 8; this looks either in the global context $\mathit{\Gamma}$ for the signature, or computes the signature from $\overline{Mb}$ if it refers to a locally defined name.

Signatures for fresh class declarations are more involved. The premises of S-Class and S-Abs-Class compute the class's representation and abstract functions. Representation computation involves looking up the superclass representation (if a superclass is declared) and "copying it down" into the current class's signature. Abstract function computation involves looking up all functions "owned" by this class and checking whether there is a default implementing case; if no such default exists, then the function is abstract for this class, and must appear in the class's `abstract on` clause. We revisit owners in Section 3.2.

We omit the rules that generate signatures for alias declarations, as they are verbose but straightforward. Informally, these lookup or compute the signature of their right-hand side, and then substitute the alias declaration's name for the referred-to declaration's name. For example, for `alias class C1 = M.C2`, we would look up the signature of `M.C2` in the environment, and `C1`'s signature

$$\boxed{\langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash Mb : Sb}$$

$$\frac{\mathrm{unmark}(\tau^{\#}) = \tau}{\langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash (\texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \tau') : (\texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \tau'\ \texttt{open below}\ \tau)}\ (\textsc{S-Fun})$$

$$\frac{\begin{array}{c} R = \mathrm{reps}(\mathit{\Gamma}, \overline{Mb}) \qquad \mathrm{unmark}(\tau^{\#}) = \tau_f \qquad \mathcal{K}, R \vdash \mathrm{ptype}(P, \tau_f) = \langle \tau^P, \beta \rangle \\ \langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash \hat{M}.f : \langle \hat{M}, \texttt{fun}\ f : \tau^{\#}\ \texttt{->}\ \_\ \texttt{open below}\ \_ \rangle \end{array}}{\langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash (\texttt{extend fun}\ \hat{M}.f\ \texttt{with}\ q\ P\ \texttt{=}\ e) : (\texttt{extend fun}\ \hat{M}.f\ \texttt{with}\ q\ \tau^P)}\ (\textsc{S-Method})$$

$$\frac{[\langle \mathit{\Gamma}, \overline{Mb} \rangle \vdash \mathrm{rep}(C) = \{\overline{L''' : \tau'''}^{1..k}\}]^1 \qquad \langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash \mathrm{abstractFuns}(c[, C]^1) = \emptyset}{\begin{array}{c} \langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash \texttt{class}\ c\ (\overline{x : \tau}^{1..m})\ [\texttt{extends}\ C\ (\overline{e})]^1 \texttt{of}\ \{\overline{l : \tau'' = e''}^{1..n}\} \\ : \texttt{class}\ c\ (\overline{\tau}^{1..m})\ \texttt{of}\ \{\overline{\texttt{ThisMod}.l : \tau''}^{1..n} [, \overline{L''' : \tau'''}^{1..k}]^1\} \end{array}}\ (\textsc{S-Class})$$

$$\frac{[\langle \mathit{\Gamma}, \overline{Mb} \rangle \vdash \mathrm{rep}(C) = \{\overline{L''' : \tau'''}^{1..k}\}]^1 \qquad \langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash \mathrm{abstractFuns}(c[, C]^1) = \overline{F}}{\begin{array}{c} \langle \mathit{\Gamma}, \mathcal{K}, \overline{Mb} \rangle \vdash \texttt{abstract class}\ c\ (\overline{x : \tau}^{1..m})\ [\texttt{extends}\ C\ e]^1\ \texttt{of}\ \{\overline{l : \tau'' = e''}^{1..n}\} \\ : \texttt{class}\ c\ (\overline{\tau}^{1..m})\ \texttt{of}\ \{\overline{\texttt{ThisMod}.l : \tau''}^{1..n} [, \overline{L''' : \tau'''}^{1..k}]^1\}\ \texttt{abstract on}\ \overline{F} \end{array}}\ (\textsc{S-Abs-Class})$$

$$\boxed{\mathrm{declRels}(\overline{Mb}) = \mathcal{K}}$$

$$\frac{\forall_{i=1}^{n}.\mathrm{fresh}(Mb_i) = \phi_i \qquad \forall_{i=1}^{n}.\mathrm{rel}(Mb_i) = \rho_i \qquad \langle \phi, \rho \rangle = \langle \cup_{i=1}^{n} \phi_i, \cup_{i=1}^{n} \rho_i \rangle}{\mathrm{declRels}(Mb_1, \ldots, Mb_n) = \langle \phi, \rho \rangle}\ (\textsc{Decl-Rels})$$

Mb	fresh(Mb)	rel(Mb)
[`abstract`] `class` c(_) `of` {_}	`ThisMod`.c	–
[`abstract`] `class` c(_) `extends` C(_) `of` {_}	`ThisMod`.c	`ThisMod`.$c <^1 C$
`alias class` c `=` C	–	`ThisMod`.$c <^0 C$
`fun` f : _ `->` _	`ThisMod`.f	–
`alias fun` f `=` F	–	`ThisMod`.$f = F$
`extend fun` F `with` q P `->` e	`ThisMod`.q	–
`alias extend fun` F `with` q `=` Q	–	`ThisMod`.$q = Q$

Fig. 10. Static semantics: Principal signatures (selected rules)

would have the same representation, constructor (if present), and `abstract on` clause (if present), but with `C1` substituted for `C2`.

Well-Formedness of Module Declarations. After a module's principal signature is generated, each of its declarations is checked for well-formed implementation ($\mathit{\Gamma} \vdash Mb$ OK in $\overline{Sb}$). The well-formedness rules contain much that is standard — for example, part of the well-formedness rule for methods typechecks the method body in the environment formed by the bindings in the method's argument. In this section, we focus only on the (relatively) non-standard requirements imposed by the unusual mechanisms of F(Eml) (note that some of these requirements are adapted with only minor changes from Eml).

Recall, from Section 3.1, the three kinds of dynamic errors: abstract instantiations, messages not understood, and ambiguous messages. Abstract instantiations can be prevented relatively easily: when typechecking a constructor invocation, verify that the constructor is visible and that class's signature does not have an `abstract on` clause.

However, preventing message-not-understood and ambiguous message errors is harder, because modular typechecking context does not, in general, contain all the concrete classes and methods in the program. New subclasses and new methods can be added by modules that are not visible in any given scope. Hence, a function may appear to be implemented on all concrete subtypes of its argument, but other concrete subtypes may still exist; similarly, all the visible cases of a function may appear to be unambiguous with each other, but other ambiguous methods may still exist. Therefore, F(EML) adapts from EML several restrictions that, taken together, prevent these errors.

Recall that function argument types must be *marked types* $\tau^{\#}$. Define the *owner position* of $\tau^{\#}$ as the position in its abstract syntax tree that is marked with a hash `#`; define a function's *owner* as the class at the owner position in its argument type (note that, unlike a *receiver* class, the owner is a purely static notion; dynamic dispatch remains symmetric); and define a method's owner as the class at the owner position of the method's argument pattern's type. Then, the following well-formedness conditions must hold for methods, functions, and classes respectively. First, each method must be defined in either the same module as its owner, or the same module as the function it extends. Second, for any function F declared in a different module from its owner, a *global default* case (which covers F's declared argument type) must be defined in the same module as F. Third, any concrete subclass C of an abstract class C' must define a *local default* case for each function F that appears in the `abstract on` clause of C''s signature; the local default case for each F must cover the argument type $\tau^{\#}$ of F, but with C substituted at the owner position of $\tau^{\#}$.

Previous work [28] has shown how the above restrictions intuitively support (more than) the union of object-oriented and functional styles of extensibility — they are crafted to permit extension with both (a) new subclasses of existing classes, and (b) new functions on existing types.

The restrictions above rule out incompleteness errors. To completely rule out ambiguity errors, we must add one further condition to method well-formedness: we must check that each method is *pairwise unambiguous* with all other visible methods. Informally, two methods are pairwise unambiguous if either: (1) they extend different functions, (2) they have *disjoint* argument types, (3) one has an argument type that *strictly subtypes* the other's argument type, (4) their argument types share a common subtype, for which a *disambiguating* case exists that is more specific than both, or (5) they are aliases of the same method.

Finally, F(EML) imposes one further requirement on function aliases. If a module M aliases a function F from module M', then that M must also contain aliases of all F's methods from M'. The reason for this is subtle; there are cases (as we shall see in Section 3.2) when subsumption may not safely hide a method. Our rules check for these conditions before allowing a method to be hidden; however, if it were possible to alias functions freely without aliasing their methods, then those methods would be hidden from clients of the alias function, bypassing these subsumption conditions and rendering typechecking unsound.

Deducing Relations. We have seen that typechecking requires several kinds of knowledge about the relationships between classes, types, functions, and methods. In most object-oriented languages with nominal subtyping — e.g., in Java — subtyping is the only type relation relevant to typechecking, and the typechecker computes subtyping by inspecting the actual inheritance graph of classes. In F(EML), we can make use of richer information about types — e.g., the fact that classes are disjoint can be used to prove two methods unambiguous — and we must also deduce function and method relations. F(EML) performs all such deductions with a set of judgments that depend only on a relation context $\mathcal{K} = \langle \phi, \rho \rangle$. To form this context, we gather the union of all ϕ and ρ from all structure signatures `sig` $\{$ `_ fresh` ϕ `where` ρ $\}$ in the range of the context Γ (during principal signature generation, we also add the initial declRels($\overline{Mb}$), as computed in Fig. 10), and run the deduction rules in this context.

Fig. 11 gives a sampling of rules for deducing class and type relations. The class deduction rules should be fairly intuitive upon inspection. Notice that CREL-NEQ implements the rule, mentioned in Section 2.4, that all fresh classes are known to be distinct from each other. The type deduction rules simply then "lift" the various class relations to the level of structured types.

$$\frac{\mathcal{K} \vdash C_1 <^i C_2 \qquad \mathcal{K} \vdash C_2 <^j C_3}{\mathcal{K} \vdash C_1 <^{i+j} C_3} \text{(CRel-Trans-Count)} \qquad \frac{C_1\ \mathcal{R}_C\ C_2 \in \rho}{\langle \phi, \rho \rangle \vdash C_1\ \mathcal{R}_C\ C_2} \text{(CRel-Lookup)} \qquad \boxed{\mathcal{K} \vdash C_1\ \mathcal{R}_C\ C_2}$$

$$\frac{\{\hat{M}.c, \hat{M}'.c'\} \subseteq \phi \quad (\hat{M} \neq \hat{M}') \vee (c \neq c')}{\langle \phi, \rho \rangle \vdash \hat{M}.c \neq \hat{M}'.c'} \text{(CRel-Neq)} \qquad \frac{\mathcal{K} \vdash C_1 \neq C_2 \quad \mathcal{K} \vdash C_1 <^k C \qquad \mathcal{K} \vdash C_2 <^k C}{\mathcal{K} \vdash C_1 \not\wedge C_2} \text{(CRel-Dis)}$$

$$\frac{\forall_{i=1}^{n}.(\mathcal{K} \vdash \tau_i \ \leq \ \tau_i')}{\mathcal{K} \vdash (\overline{\tau}^{1..n}) \ \leq \ (\overline{\tau'}^{1..n})} \text{(R-Tuple-Sub)} \qquad \boxed{\mathcal{K} \vdash \tau_1\ \mathcal{R}_\tau\ \tau_2}$$

$$\frac{\mathcal{K} \vdash C \leq C' \qquad \forall_1^n i.\mathcal{K} \vdash \tau_i \leq \tau_i'}{\mathcal{K} \vdash C\ \{\overline{L : \tau}^{1..n}, \overline{L : \tau}^{(n+1)..m}\} \ \leq \ C'\ \{\overline{L : \tau'}^{1..n}\}} \text{(R-Class-Sub)}$$

Fig. 11. Static relation deduction (selected rules)

We do not show function and method relation deduction rules, but these are straightforwardly parallel to a subset of the class relation rules. For example, FREL-LOOKUP looks up a function relation $F_1\ \mathcal{R}_F\ F_2$ in ρ, and FREL-NEQ deduces that all function names in ϕ refer to (pairwise) distinct functions.

Signature Subsumption and Selective Sealing. To be reusable, a functor should accept actual arguments whose signatures have "more information than" its formal argument signature. However, defining signature subtyping is not as simple as it would seem at first. Intuitively, subsumption hides information from a client, and unrestricted information hiding would sometimes grant a client permission to perform actions that would be prohibited by the more informative signature. In particular, hiding a function F on which a class is abstract could permit a client to create a concrete subclass of that class without providing an implementing case for F; and hiding a method Q could permit a client to define a method that is ambiguous with Q without providing a disambiguating case.

F(Eml)'s signature language therefore contains features that *selectively revoke* the privileges to perform potentially harmful actions — in particular, to subclass a class, and to extend a function — and permits hiding only when the client does not possess dangerous privileges. Fig. 12 and gives the subsumption rules that bear directly these problems. Note that relsInContext(Γ) simply extracts all the relations ϕ and ρ from each structure signature in Γ. We now describe how these rules manage the two kinds of potentially unsafe subsumption we have just mentioned — hiding functions, and hiding method cases.

First, a client can conflict with a hidden function by defining a new subclass of an abstract class C, while failing to implement the corresponding cases for a hidden abstract function. Therefore, we cannot *both* permit a client to subclass an abstract class, *and* hide a function on which that class is abstract. Notice that SB-Closed-Abs only permits abstract functions to be forgotten if the class is closed, so that clients cannot subclass it (this rule also permits the `abstract on` clause to be forgotten entirely, provided the client forgoes the privilege of invoking the constructor as well). Then, Drop-Fun requires that any dropped function not be referenced anywhere in the signature (including the `abstract on` clause of a class). Taken together, these rules encode the constraint we require — a client cannot forget about a function *and* create a concrete subclass of a class abstract on that function.

Second, a client can conflict with a hidden method by defining a new method that is ambiguous with the hidden method. Therefore, we cannot *both* permit a client to extend a function on some type, *and* hide a case that may be ambiguous with that type. Now, recall that a class may not extend a function F from outside F's enclosing module, except on a *strict subtype* of F's extension type. The Drop-Method rule requires that a method can be hidden only if it extends a local function on a *supertype* of its extension type, guaranteeing that future methods will not be ambiguous with the hidden method. By itself, this rule would be overly restrictive, since functions use their argument type as the default extension type (see S-Fun in Fig. 10). However, SB-Seal permits us to seal a function to a subtype of that function's original extension type; one can apply SB-Seal to make a method droppable, and then Drop-Method to hide it.

3.3 Soundness

Previous work [28] established the soundness of Mini-Eml (the formal core of Eml, analogous to Mini-F(Eml)) via the following standard theorems:

Theorem 1 (Mini-Eml Subject Reduction). *Given: (1)* $\forall Bn \in dom(BT)$. $BT(Bn)$ *OK, (2)* $\vdash E : T$ *in the context of* BT*, and (3)* $E \longrightarrow E'$ *in the context of* BT*, then* $\vdash E' : T'$ *for some* T' *such that* $T' \leq T$.

Theorem 2 (Mini-Eml Progress). *Given: (1)* $\forall Bn \in dom(BT)$. $BT(Bn)$ *OK, (2)* $\vdash E : T$ *in the context of* BT*, and (3)* E *is not a value, then* $\exists E'.E \longrightarrow E'$.

$$\boxed{\Gamma \vdash Sv \leq Sv'}$$

$$\frac{\begin{array}{c}\Gamma, \mathcal{K}, (\overline{Sb}; \overline{Sb'}) \vdash Sb \text{ droppable} \\ \text{relsInContext}(\Gamma) = \langle \phi', \rho' \rangle \qquad \mathcal{K} = \langle (\phi, \phi'), (\rho, \rho') \rangle\end{array}}{\Gamma \vdash \texttt{sig } \{\ \overline{Sb}; Sb; \overline{Sb'} \texttt{ fresh } \phi \texttt{ where } \rho\ \} \leq \texttt{sig } \{\ \overline{Sb}; \overline{Sb'} \texttt{ fresh } \phi \texttt{ where } \rho\ \}} \ (\textsc{Sub-Width})$$

$$\frac{\begin{array}{c}\Gamma, \mathcal{K}, (\overline{Sb}; \overline{Sb'}) \vdash Sb \leq Sb' \\ \text{relsInContext}(\Gamma) = \langle \phi', \rho' \rangle \qquad \mathcal{K} = \langle (\phi, \phi'), (\rho, \rho') \rangle\end{array}}{\Gamma \vdash \texttt{sig } \{\ \overline{Sb}; Sb; \overline{Sb'} \texttt{ fresh } \phi \texttt{ where } \rho\ \} \leq \texttt{sig } \{\ \overline{Sb}; Sb'; \overline{Sb'} \texttt{ fresh } \phi \texttt{ where } \rho\ \}} \ (\textsc{Sub-Depth})$$

$$\boxed{\Gamma, \mathcal{K}, \overline{Sb} \vdash Sb \leq Sb'}$$

$$\frac{}{\begin{array}{l}\Gamma, \mathcal{K}, \overline{Sb} \vdash \texttt{class } c\ \texttt{(}\overline{\tau}\texttt{)} \texttt{ of } \{\overline{L : \tau}\}\ [\texttt{abstract on } \overline{F}]^1 \\ \leq \texttt{closed class } c\ \texttt{(}\overline{\tau}\texttt{)} \texttt{ of } \{\overline{L : \tau}\}\ [\texttt{abstract on } \overline{F}]^1\end{array}} \ (\textsc{SB-Close})$$

$$\frac{[\overline{F'} \subseteq \overline{F}]^1}{\begin{array}{l}\Gamma, \mathcal{K}, \overline{Sb} \vdash \texttt{closed class } c\ \texttt{(}\overline{\tau}\texttt{)} \texttt{ of } \{\overline{L : \tau}\} \texttt{ abstract on } \overline{F} \\ \leq \texttt{closed class } c\ [\texttt{(}\overline{\tau}\texttt{)}]^1 \texttt{ of } \{\overline{L : \tau}\}\ [\texttt{abstract on } \overline{F'}]^1\end{array}} \ (\textsc{SB-Closed-Abs})$$

$$\frac{\mathcal{K} \vdash \tau' \leq \tau}{\Gamma, \mathcal{K}, \overline{Sb} \vdash \texttt{fun } f : \tau^{\#} \texttt{ -> } \tau_r \texttt{ open below } \tau \leq \texttt{fun } f : \tau^{\#} \texttt{ -> } \tau_r \texttt{ open below } \tau'} \ (\textsc{SB-Seal})$$

$$\boxed{\Gamma, \mathcal{K}, \overline{Sb} \vdash Sb \text{ droppable}}$$

$$\frac{\begin{array}{c}(\texttt{fun } f : \_ \texttt{ -> } \_ \texttt{ open below } \tau_f^P) \in \overline{Sb} \\ \texttt{ThisMod}.q \notin \text{freeNames}(\overline{Sb}) \qquad \mathcal{K} \vdash \tau_f^P \leq \tau^P\end{array}}{\Gamma, \mathcal{K}, \overline{Sb} \vdash (\texttt{extend fun ThisMod}.f \texttt{ with } q\ \tau^P) \text{ droppable}} \ (\textsc{Drop-Method})$$

$$\frac{\texttt{ThisMod}.c \notin \text{freeNames}(\overline{Sb})}{\Gamma, \mathcal{K}, \overline{Sb} \vdash (\ [\texttt{abstract}]\ \texttt{class } c \ldots) \text{ droppable}} \ (\textsc{Drop-Class})$$

$$\frac{\texttt{ThisMod}.f \notin \text{freeNames}(\overline{Sb})}{\Gamma, \mathcal{K}, \overline{Sb} \vdash (\texttt{fun } f : \_ \texttt{ -> } \_ \texttt{ open below } \_) \text{ droppable}} \ (\textsc{Drop-Fun})$$

Fig. 12. Static semantics: Signature subsumption (selected rules)

Here, the "block table" BT is a finite map from block names Bn to *blocks* (module values), E is a Mini-EML core expression, and T is a Mini-EML type. $BT(Bn)$ OK denotes the Mini-EML judgment that the block $BT(Bn)$ is well-formed. $\vdash E : T$ denotes that E has the Mini-EML type T. $E \longrightarrow E'$ is the Mini-EML small-step evaluation relation. Now, we define a function $\lfloor\ \rfloor$ which translates Mini-F(EML) syntax into Mini-EML: $\lfloor \mathcal{D}; \Delta; e \rfloor$ denotes the translation of a compiled Mini-F(EML) program into a Mini-EML program $BT; E$, assuming the module dependency relation $\mathcal{D}$. We then require two extra properties:

Theorem 3 (Well-Formed Translation). *If (1) $\emptyset, \emptyset \vdash \overline{Md} \Rightarrow^* \Gamma, \mathcal{D}$, (2) $\emptyset \vdash \overline{Md} \Downarrow^* \Delta$, and (3) $\lfloor \mathcal{D}, \Delta \rfloor = BT$, then (G1) $\forall Bn \in dom(BT).BT(Bn)$ OK.*

Theorem 4 (Type Preservation). *If (1) $\emptyset, \emptyset \vdash \overline{Md} \Rightarrow^* \Gamma, \mathcal{D}$, (2) $\emptyset \vdash \overline{Md} \Downarrow^* \Delta$, (3) $\lfloor \mathcal{D}, \Delta; e \rfloor = BT; E$, and (4) $\Gamma, \emptyset, \emptyset \vdash e : \tau$, then $(G1) \vdash E : \lfloor \tau \rfloor$ in BT.*

Provided the above properties hold, it follows that if a Mini-F(EML) program typechecks, then its Mini-EML translation typechecks, and the translated program does not go wrong. We working towards completion of the proofs, which will appear in a companion technical report [22].

4 Related Work

As previously mentioned, the direct predecessor to F(Eml) is Eml [28]. A sibling of Eml is MultiJava [10, 29], which explores many of the same issues and could be extended with parameterized modules in closely analogous ways. Nice [3] resembles Eml (though it is built on a different formalism) in providing multiple dispatch and a form of modular typechecking, without parameterized modules.

A *mixin* [5, 17] is a class that inherits from a parameter to be provided later. Bracha and Cook first proposed mixins [5] for a single-dispatch object-oriented language. Statically typed mixin languages prior to our work generally have not supported multiple dispatch, or permitted addition of dispatching functions from outside the receiver class. *Traits* [36, 38, 32] are a mixin-like multiple inheritance mechanism wherein classes can inherit *one* ordinary superclass and *multiple* traits, where traits may not define constructors or state. Traits languages would still gain additional flexibility if combined with functors: a class defining constructors and state could (by functorization of the containing module) be parameterized by a superclass that also defined constructors and state.

Many languages allow general multiple inheritance, which can support mixin-like idioms. Multiple inheritance comes with a number of known problems, e.g., the "diamond inheritance" problem. Like traditional mixin languages, F(Eml) sidesteps these problems (with some loss of expressiveness) by offering single inheritance, plus the alternative composition mechanism of parameterization.

Virtual types (or *virtual classes* [25]) extend class-level inheritance with *overridable type members* nested inside classes. Virtual types can statically typecheck many idioms like those supported by parameterized classes and modules [7, 40, 14]. In languages like **gbeta** [13], Scala [32], Jx [31], and CaesarJ [2], virtual types also support *family polymorphism* [13], an idiom for writing code that is generic over multiple instantiations of related groups of types. Virtual and parametric types share deep connections, and we suspect that any given language feature raises closely analogous issues in either style of system. For example, if one added multiple dispatch to virtual type systems, then determining whether a type member could be safely overridden in a subclass might raise issues like those that F(Eml) encounters in defining subsumption for classes in functor argument signatures. Conversely, adding family polymorphism support to F(Eml) might require dependent type mechanisms akin to those in virtual type systems.

F(Eml)'s functors are inspired by ML's parameterized module system [19]. Many extensions to ML parameterized modules have been proposed [23, 18, 11], but none have incorporated extensible data types, extensible functions, *and* symmetric multiple dispatch. OML [34], OCaml [33], and Moby [15] combine ML-style modules orthogonally with object-oriented classes, but these classes are traditional receiver-oriented constructs: dispatching methods can only be declared with their receiver class, and cannot be externally added without modifying the original declaration. $ML_{\leq}$ [4] generalizes ML datatypes with subtyping and symmetric dispatch, but does not support addition of new cases to existing functions from outside of the extended declaration's original module. Several proposals extend ML with *mixin modules* [12, 20]; these systems do not currently support subtyping among datatype cases, ruling out object-oriented idioms.

Jiazzi [26] (based on Units [16]) and JavaMod [1] extend Java with parameterized modules that support many idioms, including mixins. These languages only support single dispatch, so in this sense they are more restrictive than F(EML); however, conversely, they support recursive module linkage, which our work does not (although we believe recursive linkage could be added to F(EML)). Jiazzi also supports the addition of dispatching functions externally to a class, through an *open class* design pattern, though this requires more advance planning than in F(EML), where external functions can be added directly.

Classes in C++ templates [39] can inherit from a template parameter, but templates do not support separate typechecking of template bodies. Parameterized classes in GJ [6] support separate typechecking, but disallow inheritance from the type parameter, ruling out idioms like mixins.

5 Conclusions and Future Work

We have described a parameterized module system with several novel features in the module and signature language. The module language includes aliasing declarations, which permit potential arguments to be adapted to the naming and modularization requirements of a parameterized module. The signature language allows a parameterized module to specify two important kinds of requirements of its argument: how its declarations are *related* to each other, and how *extensible* the classes and functions must be. These constraints enable the body of the parameterized module to be typechecked separately from instantiations, even in the face of extensible classes, extensible functions, and methods with symmetric multiple dispatching. At the same time, these constraints remain weak enough to allow the parameterized module to be applied to a wide range of arguments.

In the future, we would like to study relaxing F(EML)'s modular typechecking restrictions, along the lines of Relaxed MultiJava [29], to give the programmer more control over the trade-off between modular typechecking and programming flexibility. We also think it would be interesting to explore the ideas in this paper in the context of a virtual type-based system. Finally, we plan to adapt and implement these ideas in Diesel, a language which adds a module system to an underlying core language based on Cecil [8, 9].

Acknowledgments. This work has been supported in part by NSF grants CCR-0204047 and ACI-0203908. We wish to thank Erik Ernst, Todd Millstein, the University of Washington WASP group, and anonymous reviewers of this work (and its earlier incarnations) for their invaluable feedback and discussions.

References

1. D. Ancona, E. Zucca. True Modules for Java-like Languages. *15th ECOOP*, 2001.
2. I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann. An Overview of CaesarJ. *Trans. on Aspect-Oriented Development I*, LNCS 3880 pp. 135-173, Feb. 2006.

3. D. Bonniot. Type-checking multi-methods in ML (A modular approach). *FOOL 9*, 2002.
4. F. Bourdoncle, S. Merz. Type checking higher-order polymorphic multi-methods. *24th POPL*, 1997.
5. G. Bracha, W. Cook. Mixin-based Inheritance. In *OOPSLA*, 1990.
6. G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. *OOPSLA*, 1998.
7. K. B. Bruce, M. Odersky, P. Wadler. A Statically safe alternative to virtual types. *12th ECOOP*, 1998.
8. C. Chambers. Object-Oriented Multi-Methods in Cecil. *6th ECOOP*, 1992.
9. C. Chambers, Cecil Group. The Cecil Language: Specification and Rationale. Univ. of Washington Technical Report UW-CSE-93-03-05, 1993-2004.
10. C. Clifton, G. T. Leavens, C. Chambers, T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. *OOPSLA*, 2000.
11. D. Dreyer, K. Crary, R. Harper. A Type System for Higher-Order Modules. *30th POPL*, 2003.
12. D. Duggan, C. Sourelis. Mixin modules. In *First ICFP*, Philadelphia PA, 1996.
13. E. Ernst. Family Polymorphism. *15th ECOOP*, June 2001.
14. E. Ernst, K. Ostermann, W. R. Cook. A Virtual Class Calculus. *POPL*, 2006.
15. K. Fisher, J. Reppy. The design of a class mechanism for Moby. *PLDI*, June 1999.
16. M. Flatt, M. Felleisen. Units: Cool modules for HOT languages. *PLDI*, 1998.
17. M. Flatt, S. Krishnamurthi, M. Felleisen. Classes and Mixins. *25th POPL*, 1998.
18. R. Harper, M. Lillibridge. A Type-theoretic approach to higher-order modules with sharing. *POPL*, 1994.
19. R. Harper, C. Stone. A Type-theoretic interpretation of Standard ML. Carnegie Mellon Dept. of CS Technical Report CMU-CS-97-147, 1997.
20. T. Hirschowitz, X. Leroy. Mixin modules in a call-by-value setting. *European Symp. on Programming*, LNCS 2305, D. Le Metayer, ed., 2002.
21. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. *11th ECOOP*, 1997.
22. K. Lee, C. Chambers. Parameterized modules for extensible classes and functions. Univ. of Washington Technical Report UW-CSE-2005-07-01, 2006 (forthcoming).
23. X. Leroy. Manifest types, modules, and separate compilation. *21st POPL*, 1994.
24. R. E. Lopez-Herrejon, D. Batory, W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. *19th ECOOP*, 2005.
25. O. L. Madsen, B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conf. OOPSLA*, 1989.
26. S. McDirmid, M. Flatt, W. C. Hsieh. Jiazzi: New age modules for old-fashioned Java. *16th OOPSLA*, pp. 211-222, Tampa Bay FL, 2001.
27. T. Millstein, C. Chambers. Modular Statically Typed Multimethods. *13th ECOOP*, 1999.
28. T. Millstein, C. Bleckner, C. Chambers. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. *ACM TOPLAS* 26(5):836-889, 2004.
29. T. Millstein, M. Reay, C. Chambers. Relaxed MultiJava: Balancing Extensibility and Modular Typechecking. In *OOPSLA*, Oct. 2003.
30. R. Milner, M. Tofte, R. Harper, D. MacQueen. *Def. of Standard ML (Revised).* MIT Press, 1997.
31. N. Nystrom, S. S. Chong, A. C. Myers. Scalable Extensibility via Nested Inheritance. *OOPSLA*, 2004.

32. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger. An Overview of the Scala Programming Language. EPFL Technical Report IC/2004/64. EPFL Lausanne, 2004.
33. D. Rémy, J. Vouillon. Objective ML: a simple object-oriented extension of ML. *24th POPL*, 1997.
34. J. Reppy, J. Riecke. Simple objects for Standard ML. *1996 PLDI*, 1996.
35. J. C. Reynolds. User defined types and procedural data structures as complementary approaches to data abstraction. In *Programming Methodology, A Collection of Articles by IFIP WG2.3*, D. Gries, ed., Springer-Verlag, 1978.
36. N. Schärli, S. Ducasse, O. Nierstrasz, A. Black. Traits: Composable Units of Behavior. *18th ECOOP*, LNCS 2743, July 2003.
37. Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration Designs. *ACM TSEM* 11(2):215-255, April 2002.
38. C. Smith, S. Drossopoulou. Chai: Traits for Java-like Languages. *ECOOP*, 2005.
39. B. Stroustrup. *The C++ Programming Language, 3rd Ed.* Addison-Wesley, 2000.
40. K. K. Thorup, M. Torgersen. Unifying genericity – combining the benefits of virtual types and parameterized classes. *13th ECOOP*, 1999.
41. Philip Wadler. The Expression Problem. Java-genericity email list, Nov. 1998.