Modular Typechecking for Hierarchically Extensible Datatypes and Functions*

Todd Millstein, Colin Bleckner, and Craig Chambers
Department of Computer Science and Engineering
University of Washington
{todd,colin,chambers}@cs.washington.edu

Abstract

One promising approach for adding object-oriented (OO) facilities to functional languages like ML is to generalize the existing datatype and function constructs to be hierarchical and extensible, so that datatype variants simulate classes and function cases simulate methods. This approach allows existing datatypes to be easily extended with both new operations and new variants, resolving a long-standing conflict between the functional and OO styles. However, previous designs based on this approach have been forced to give up *modular* typechecking, requiring whole-program checks to ensure type safety. We describe Extensible ML (EML), an ML-like language that supports hierarchical, extensible datatypes and functions while preserving purely modular typechecking. To achieve this result, EML's type system imposes a few requirements on datatype and function extensibility, but EML is still able to express both traditional functional and OO idioms. We have formalized a core version of EML and proven the associated type system sound, and we have developed a prototype interpreter for the language.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, data types and structures, procedures, functions, and subroutines*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax, semantics*

General Terms

Design, Languages, Theory

Keywords

extensible datatypes, extensible functions, modular typechecking

*An earlier version of this paper was presented at the Ninth International Workshop on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon, January 19, 2002.

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

1 Introduction

Many researchers have noted a difference in the extensibility benefits offered by the functional and object-oriented (OO) styles [28, 9, 25, 11, 19, 15, 30]. Functional languages like ML allow new operations to be easily added to existing datatypes (by adding new `fun` declarations), without requiring access to existing code. However, new data variants cannot be added without a potentially whole-program modification (since existing functions must be modified in place to handle the new variants). On the other hand, traditional OO approaches allow new data variants to be easily added to existing class hierarchies (by declaring subclasses with overriding methods), without modifying existing code. However, adding new operations to existing classes requires access to the source code for those classes (since methods cannot be added to existing classes without modifying them in place).

There have been several recent research efforts to integrate the benefits of the functional and OO styles in the context of ML. OCaml [26] adds OO features including class and method definitions to ML. The OO constructs essentially form their own sublanguage which is largely separate from the existing ML `datatype` and `fun` constructs. Adding a set of new constructs has the advantage that existing language constructs are minimally affected by the extension, retaining their traditional semantics and typing properties. Further, the augmented language addresses the expressiveness differences of the functional and OO styles in a very simple way, by providing both options. However, such simplicity comes at a cost to programmers, who are forced to choose up front whether to represent an abstraction with datatypes or with classes. As described above, this decision impacts the kind of extensibility allowable for the abstraction. It may be difficult to determine *a priori* which kind of extensibility will be required, and it is difficult to change the decision after the fact. Further, it is not possible for the abstraction to enjoy both kinds of extensibility at once.

An alternative approach is to generalize existing ML constructs to support the OO style. OML [27], for example, introduces an `objtype` construct for modeling class hierarchies. This construct can be seen as a generalization of ML datatypes to be hierarchical and extensible. Therefore, programmers need not decide between datatypes and classes up front; both are embodied in the `objtype` construct. However, OML still maintains a distinction between methods and functions, which have different benefits. New methods may not be added to existing `objtype`s without modifying existing code, while ordinary ML functions may be. Methods dynamically dispatch on their associated `objtype`, while functions support ML-style pattern matching.

$ML_{\leq}$ [3] integrates the OO style further with existing ML constructs. Like OML, $ML_{\leq}$ generalizes ML datatypes to be hierarchical and extensible. Further, methods are simulated via function cases that use OO-style dynamic dispatching semantics. In this approach, programmers need not choose between two forms of extensibility; a single language mechanism supports the easy addition of both new operations and new variants to existing datatypes.

However, there are important ways in which $ML_{\leq}$ is not well integrated with existing ML language features. First, $ML_{\leq}$ does not support ML-style pattern matching. Patterns are essentially restricted to be top-level datatype constructor tests, which are the analogue of dynamic dispatch tests in OO languages. Other common ML-style patterns and patterns on sub-components cannot be programmed.

Second, extensible datatypes are of limited utility without extensible functions, which allow existing functions to be updated with new cases as new data variants are declared. However, $ML_{\leq}$ does not support extensible functions: all function cases are provided when a function is declared. The authors sketch a source-level language that supports extensible functions. Unfortunately, this critical generalization of their work causes a loss of *modular* reasoning: static typechecking of a program cannot be completed until link-time, when all modules are available. Therefore, important software engineering benefits are lost, including early detection of errors, libraries that are guaranteed to be typesafe in any context satisfying their interface requirements, independent development of typesafe modules by separate teams of programmers, and incremental modification (and subsequent incremental re-typechecking) of code.

The checks that must be delayed to link-time in $ML_{\leq}$ constitute what we call *implementation-side typechecking* (ITC), which ensures that each function in the program is completely and unambiguously implemented [8].[1] In traditional functional languages, ITC checks each function for *match nonexhaustive* and *match redundant* errors. Each function can be checked modularly, since a function declaration includes all of its cases and datatypes are not extensible. In traditional OO languages, ITC checks that each class declares or inherits a *most-specific* method for each supported operation. Each class can be checked modularly, since a class declaration includes all of its (non-inherited) methods and new operations cannot be added to existing classes.

The implicit restrictions in the traditional functional and OO settings that allow for modular ITC do not hold in the presence of extensible datatypes and functions. Unlike traditional functional languages, no module is guaranteed to have access to all of a function's cases. Unlike traditional OO languages, no module is guaranteed to have access to all of a datatype variant's associated functions and function cases. Therefore, $ML_{\leq}$ is forced to perform ITC *globally*, when the whole program is available.

In this work, we describe an ML-like language called Extensible ML[2] (EML). EML introduces a `class` construct, which is a form of hierarchical, extensible datatype in the spirit of the constructs in OML and $ML_{\leq}$. As in $ML_{\leq}$, methods are simulated by function cases. In addition:

[1] Implementation-side typechecking contrasts with *client-side typechecking* of functions, which checks that each function *application* in the program is type-correct. Client-side typechecking is standard and can be performed modularly.

[2] not to be confused with *Extended* ML [18]

```
structure SetMod = struct
  abstract class Set() of {}
  class ListSet(es:int list) extends Set()
    of {es:int list = es}
  class CListSet(es:int list, c:int)
    extends ListSet(es) of {count:int = c}

  fun add:(int * #Set) → Set
  extend fun add (i, s as ListSet {es=es}) =
    if (member i es) then s else ListSet(i::es)
  extend fun add (i, s as CListSet {es=es,count=c}) =
    if (member i es) then s else CListSet(i::es,c+1)

  fun size:Set → int
  extend fun size (ListSet {es=es}) = length es
  extend fun size (CListSet {es=_,count=c}) = c

  fun elems:Set → int list
  extend fun elems (ListSet {es=es}) = es
end
```

Figure 1. A hierarchy of integer sets in EML.

- EML generalizes the OO dispatching semantics in $ML_{\leq}$ to allow arbitrary ML-style patterns. This generalization provides idioms that are not expressible by either traditional functional or OO languages.
- EML supports extensible functions while preserving purely modular typechecking: each module can be typechecked given only the *interfaces* of the modules it *statically depends upon* (in a sense described later), with no whole-program checks required. To make per-module implementation-side typechecking sound without necessitating link-time checks, EML's type system imposes certain requirements via the notion of a function's *owner position*, which serves to coordinate otherwise independent extensions to the function. The owner position generalizes some of the properties of a method's receiver in traditional OO languages, shedding new light on how those languages achieve modular typechecking. Despite the imposed requirements, EML's classes and functions are still able to simultaneously express traditional functional and OO extensibility idioms. The requirements are adapted from our earlier work on Dubious [22, 23], a calculus designed to explore modular typechecking for OO languages based on multimethods.

The rest of the paper is organized as follows. Section 2 describes EML by example. Section 3 discusses the challenges for performing modular implementation-side typechecking in EML and presents our solution to these challenges. Section 4 defines MINI-EML, a core language for EML used to formalize our modular type system. Section 5 describes how the features of EML interact with an ML-style module system, including signature ascription and functors. Section 6 discusses related work, and section 7 concludes. We have proven the type system of MINI-EML sound. A companion technical report [21] contains the complete formal dynamic and static semantics of MINI-EML as well as the type soundness proof.

2 EML by Example

Figure 1 shows an EML implementation of integer sets. Classes, functions, and function cases are declared in ML-style `structs`. In our discussion we assume that `structs` contain only those three kinds of declarations. This assumption is lifted in section 5, which describes the interaction of EML's features with an ML-style module system.

2.1 Classes

The `Set` class in figure 1 is the top of the integer set hierarchy. The `ListSet` class inherits from `Set`, implementing sets via lists. The `CListSet` class inherits from `ListSet`, additionally keeping track of the number of elements in the set. A program's subclass relation is the reflexive, transitive closure of the declared `extends` relation. Classes support only single inheritance. However, like Java [1, 16], EML supports a notion of `interface`, and a class can implement multiple interfaces. We ignore `interfaces` in this paper for simplicity. The `Set` class is declared `abstract`, so it may not be instantiated, while its subclasses `ListSet` and `CListSet` are *concrete*.

Each class declares a record type of its instance variables, using the `of` clause. Superclass instance variables are inherited: the *representation type* of a class *C* is the representation type (recursively) of its direct superclass (if any) concatenated with the type in the `of` clause in *C*'s declaration. For example, the representation type of `CListSet` is `{es:int list,count:int}`, since `ListSet`'s representation type is `{es:int list}`.

Each class declaration also implicitly declares a constructor, similar to constructor declarations in OCaml [26] and XMOC [13], a core language for Moby [12]. For example, the `CListSet` constructor expects arguments `es` of type `int list` and `c` of type `int`, initializes inherited instance variables via the call `ListSet(es)` to the superclass constructor, and initializes the new `count` instance variable to `c`. In general, the arguments to the superclass constructor call and the instance-variable initializers may be arbitrary expressions. It would be straightforward to allow a class to have multiple constructors by introducing a separate `constructor` declaration, similar to "makers" in Moby.

Classes can be used to simulate ordinary ML-style datatypes. In particular, an ML datatype of the form

$$\texttt{datatype DT} = \texttt{C}_1 \texttt{ of } \{\texttt{L}_{11}\texttt{:T}_{11},\ldots,\texttt{L}_{1m}\texttt{:T}_{1m}\} \mid \cdots \\ \mid \texttt{C}_r \texttt{ of } \{\texttt{L}_{r1}\texttt{:T}_{r1},\ldots,\texttt{L}_{rn}\texttt{:T}_{rn}\}$$

is encoded in EML by the following class declarations:

$$\begin{array}{l}\texttt{abstract class DT of } \{\} \\ \texttt{class C}_1\texttt{(I}_{11}\texttt{:T}_{11},\ldots,\texttt{I}_{1m}\texttt{:T}_{1m}\texttt{) extends DT()} \\ \quad \texttt{of } \{\texttt{L}_{11}\texttt{:T}_{11}\texttt{=I}_{11},\ldots,\texttt{L}_{1m}\texttt{:T}_{1m}\texttt{=I}_{1m}\} \\ \cdots \\ \texttt{class C}_r\texttt{(I}_{r1}\texttt{:T}_{r1},\ldots,\texttt{I}_{rn}\texttt{:T}_{rn}\texttt{) extends DT()} \\ \quad \texttt{of } \{\texttt{L}_{r1}\texttt{:T}_{r1}\texttt{=I}_{r1},\ldots,\texttt{L}_{rn}\texttt{:T}_{rn}\texttt{=I}_{rn}\}\end{array}$$

Unlike the variants in ordinary ML datatypes, classes are full-fledged types, and other classes may inherit from them.

A concrete class is instantiated by invoking its constructor. For example, the result of `ListSet([5,3])` is an instance of `ListSet` representing the set {5,3}. Like values of ML datatypes, class instances have no special object identity or mutable state; `refs` can be used in a class's representation type for this purpose.

2.2 Functions and Function Cases

To make functions extensible, we break an ML-style function declaration into two pieces. The `fun` declaration introduces a function and specifies its type. The `size` function in figure 1, for example, is declared to accept an instance of `Set` or a subclass and to return an integer. The `#` in the `add` function's argument type signifies that the second argument to `add` is in the *owner position*. As a syntactic sugar, the owner position of a function is assumed to be the entire argument when no `#` is present in the function's argument type. A function and its cases must satisfy several requirements with respect to its owner position, to ensure that the function can be modularly checked for exhaustiveness and unambiguity. These requirements are discussed in section 3. The owner position has no dynamic effect.

The `extend fun` declaration adds a case to an existing function. The declaration specifies the name of the function being extended, a pattern guard, and the new case's body. There are two `size` function cases in figure 1, handling `ListSets` and `CListSets`, respectively. In a traditional OO language, these `size` cases would be declared as `size` methods in the `ListSet` and `CListSet` class declarations. The `extend fun` declaration is *imperative*, updating the set of cases associated with the specified function rather than creating a new function containing the extra case. The imperative semantics allows extensible functions to faithfully model OO-style methods, which conceptually update a "generic function" consisting of all methods that dynamically override some particular "top" method. The imperative semantics is necessary to support common OO idioms. For example, clients of an OO class hierarchy often import only the abstract base class of the hierarchy, with any message sends through that class's interface dynamically dispatched to the appropriate methods of (potentially unknown) concrete subclasses.

An ML-style function consisting of *n* function cases is encoded in EML as a `fun` declaration followed by *n* `extend fun` declarations. EML functions can be passed to and returned from other functions, like lambdas and ML-style functions. However, a function's extensibility is second-class: new cases may only be added to statically known functions.

Patterns in EML subsume both OO-style dynamic dispatching and ML-style pattern matching. For example, the second `size` case in figure 1 is only applicable dynamically if the argument is an instance of `CListSet` or a subclass, whose instance variables match the given *representation pattern* (which in this case is fully general). As usual, the pattern also binds identifiers for use in the case's body.

An OO-style "best-match" policy decides which function case to invoke; their order does not matter. Given an application of function *f* with argument value *v*, first the *applicable* cases of *f* for *v* are retrieved. These are the cases that have a pattern that *v* matches. Of the applicable cases, the unique case that is *more specific* than all other applicable cases is invoked. Intuitively, case c_1 is more specific than case c_2 if the set of values matching c_1's pattern is a subset of the set of values matching c_2's pattern. We call the invoked case the *most-specific applicable* case. If a function application has no applicable cases, a *match nonexhaustive* error occurs. If a function application has at least one applicable case but no most-specific one, a *match ambiguous* error occurs.

For example, consider the invocation `size(CListSet([5,3],2))`. Both `size` cases in figure 1 are applicable to the argument value, and the second case is invoked because it is the more-specific one. The "best-match" semantics contrasts with the traditional "first-match" semantics of function cases in ML. The "first-match" semantics does not generalize naturally to handle extensible datatypes and functions, where typically the more-specific function cases are written *after* the less-specific ones, as new data variants are defined.

```
structure UnionMod = struct
  fun union:(#Set * Set) → Set
  extend fun union (s1, s2) = fold add s2 (elems s1)
  extend fun union (ListSet {es=e1}, ListSet {es=e2}) =
    ListSet(merge(sort(e1), sort(e2)))
end
```

Figure 2. Adding new functions in EML.

```
structure HashSetMod = struct
  class HashSet(ht:(int,unit) hashtable)
    extends Set() of {ht:(int,unit) hashtable = ht}

  extend fun add (i, s as HashSet {ht=ht}) =
    if containsKey(i,ht) then s
    else HashSet(put(i,(),ht))

  extend fun size (HashSet {ht=ht}) = numEntries(ht)

  extend fun elems (HashSet {ht=ht}) = keyList(ht)
end
```

Figure 3. Adding new data variants in EML.

Implementation-side typechecking ensures that *match nonexhaustive* and *match ambiguous* errors cannot occur at run-time. Each module's typechecks include ITC for functions whose exhaustiveness and unambiguity may be affected by the module. These are functions declared in the module, functions with cases declared in the module, and functions that can accept instances of classes declared in the module. For example, ITC of `SetMod` in figure 1 checks the three functions declared there. Consider checking the `size` function for exhaustiveness and unambiguity. Any `ListSet` instance will invoke the first `size` case, and any `CListSet` instance will invoke the second `size` case. The `Set` class need not have a most-specific applicable case, because `Set` is declared `abstract`. Therefore, ITC for `size` succeeds. On the other hand, if the first `size` case were missing, a *match nonexhaustive* error would be signaled statically. Alternatively, if another `size` case with pattern `ListSet {es=es}` were declared, a *match ambiguous* error would be signaled statically.

2.3 Adding New Functions

As with ML datatypes, but unlike traditional classes, EML supports the easy addition of new functions to an existing class hierarchy. For example, figure 2 adds a function for computing the union of two `Set`s, without modifying any code in the `SetMod` module.[3] Two `union` function cases are provided. The first case is applicable to any pair of `Set`s. The second `union` case provides a more efficient implementation for two `ListSet`s. ITC of `UnionMod` checks `union` for exhaustiveness and unambiguity. Any pair of `ListSet`s and `CListSet`s will invoke the second `union` case, so the function's check succeeds.

2.4 Adding New Data Variants

Unlike ML datatypes, classes in EML also support the easy addition of new data variants to existing hierarchies, without modifying existing code. An example is shown in figure 3, which provides a new implementation `HashSet` of sets using an existing implementation (not shown) of hash tables. Implementations of `add`, `size`, and `elems` are provided for the new kind of set. In a traditional OO language, `HashSetMod` corresponds to the declaration of a new subclass of `Set` with some overriding methods. ITC of `HashSetMod` re-checks `add`, `size`, and `elems` to ensure that they handle `HashSet` instances. For example, if the new `size` case were not declared, a *match nonexhaustive* error for `size` would be signaled statically.

[3] Technically, all references to `Set`, `ListSet`, `add`, and `elems` in `UnionMod` should instead be to `SetMod.Set`, `SetMod.ListSet`, `SetMod.add`, and `SetMod.elems`. For readability, we omit the full path names in examples when clear from context.

```
structure SortedListSetMod = struct
  class SListSet(es:int list) extends ListSet(es)
    of {}

  extend fun add (i, s as SListSet {es=es}) =
    if (member i es) then s else
    let (lo,hi) = partition (fn j=>j<i) es
    in SListSet(lo@(i::hi)) end

  extend fun union (SListSet {es=e1},
                    SListSet {es=e2}) =
    SListSet(merge(e1,e2))

  fun getMin:SListSet → int
  extend fun getMin (SListSet {es=es}) = hd(es)
end
```

Figure 4. Class hierarchies in EML.

`HashSetMod` and `UnionMod` from figure 2 illustrate EML's support for both OO and functional forms of extensibility in a single class hierarchy. The original `Set` abstraction is flexibly reused by clients, who add a specialized implementation (subclass) of the abstraction and also augment the abstraction with client-specific functionality, all without modifying existing code. `HashSetMod` and `UnionMod` are completely independent: either, both, or neither module could be linked into the final program. In this way, different versions of the `Set` abstraction may be used in different programs, depending on the needs of a particular application.

If both `UnionMod` and `HashSetMod` are present in a program, then `HashSet` implicitly supports the `union` operation and inherits any applicable cases. This expressiveness is at the heart of the problem of modular ITC. Because the two modules are independent, neither is "aware" of the other during its static typechecks. Therefore, neither module's ITC ensures that `union` is completely and unambiguously implemented for `HashSet`s. In this example, `union` happens to have a case that handles `HashSet`s (by handling any pair of sets). Without extra requirements, however, things do not always work out so well, as we show in section 3.

Another example of data-variant extensibility is illustrated in figure 4. A new subclass of `ListSet` is created, representing an implementation of sets via sorted lists. `SListSet` inherits the representation type of `ListSet` (adding no new instance variables) as well as the applicable function cases of `size` and `elems`. Overriding cases of `add` and `union` are provided, as well as a new operation for accessing the minimum element of a set implemented as a sorted list. ITC of `SortedListSetMod` checks `add`, `size`, `elems`, `union`, and `getMin` to ensure exhaustiveness and unambiguity for `SListSet`s.

2.5 Parametric Polymorphism

EML supports a polymorphic type system. Class, function, and function case declarations optionally bind *type variables*. Refer-

```
abstract class 'a Set() of {}
class 'a ListSet(es:'a list) extends 'a Set()
  of {es:'a list = es}
class 'a CListSet(es:'a list, c:int)
  extends 'a ListSet(es) of {count:int = c}

fun 'a add:
  ('a * # 'a Set * ('a → 'a Set → bool)) → 'a Set
extend fun 'a add (i, s as ListSet {es=es}, member) =
  if (member i s) then s else 'a ListSet(i::es)
extend fun 'a add (i, s as CListSet {es=es,count=c},
                   member) =
  if (member i s) then s else 'a CListSet(i::es,c+1)
```

Figure 5. Polymorphic sets in EML.

ences to a polymorphic class or function specify a particular *type instantiation*. As an example, figure 5 shows some of the declarations for a polymorphic version of the sets in figure 1. Each class in the set hierarchy is now parameterized by the element type, as is the `add` function. Each function case is also explicitly parameterized, allowing its function's type variables to be renamed for use in the case's body. References to classes in a case's pattern do not contain type parameters. The appropriate type instantiation for such classes can be inferred from the declared argument type (for example, the reference to `CListSet` in the second `add` case's pattern is implicitly `'a CListSet`).

EML's polymorphic type system is deliberately simple in several ways. First, EML is explicitly typed. Second, we require that subclasses have the same type variables as their superclasses. This requirement is consistent with polymorphism in ML, where data variants have the same type variables as their associated datatype. Third, type parameters are *invariant*; for example, $\mathtt{T}_1$ `ListSet` is a subtype of $\mathtt{T}_2$ `Set` if and only if $\mathtt{T}_1$=$\mathtt{T}_2$. Finally, there is no support for bounded polymorphism, which would, for example, obviate the need to explicitly pass the membership function to `add`.

We have chosen to make the polymorphic type system simple because polymorphism is orthogonal to the problems of modular ITC that we address in this work. Those problems arise from the fact that some related classes, functions, and function cases are not modularly "aware" of one another; the problems are neither reduced nor exacerbated by polymorphic types. Therefore, our polymorphic type system could be generalized in standard ways without affecting our results. For example, we could adopt $\mathrm{ML}_{\leq}$'s subtype-constrained polymorphic types [3] and associated decidable type system. Recent work [2] has presented a simplified account of $\mathrm{ML}_{\leq}$'s type system and has additionally shown how to incorporate a form of type inference.

3 Modular Implementation-side Typechecking

This section focuses on the problem of modular ITC for EML. First we define our notion of modular typechecking. Next we illustrate the ways in which naive modular ITC is unsound. Finally we describe the requirements we impose to achieve modular type safety.

3.1 Modular Typechecking

We say that a language's typechecking scheme is *modular* if it has two properties. First, each module *m* can be typechecked given only the *interfaces* of other modules (without requiring access to the associated implementations). Second, *m* can be typechecked given only those interfaces that *m statically depends upon*. Module *m* statically depends upon interface *i* if either of the following conditions holds:

- Module *m* refers to a name that is bound in *i*.
- Module *m* statically depends upon module interface i', and i' refers to a name that is bound in *i*.

Traditional functional languages can support modular typechecking. For example, each structure in ML could be typechecked given only its statically depended-upon structure interfaces. A structure's interface is either an explicitly ascribed signature or else the structure's *principal signature*. Similarly, each class in a standard OO language can be typechecked given only the statically depended-upon class interfaces. Informally, the interface of a class consists of its list of superclasses, the types of its visible fields, and the headers, but not bodies, of its visible methods.

A modular typechecking scheme for EML must typecheck each structure given only the interfaces it statically depends upon. We implicitly use a structure's principal signature as its interface. The principal signature of an EML structure includes all of its class and function declarations, as well as the headers (but not the bodies) of all function case declarations. Explicit signatures provide a richer notion of structure interface, as described in section 5. Classes, functions, and cases that are declared in *m* or specified in an interface upon which *m* statically depends are said to be *available* during the typechecking of *m*. All other classes, functions, and cases are *unavailable* and may not be considered during the typechecking of *m*.

Our definition of modular typechecking validates the intuition that `union` of figure 2 and `HashSet` of figure 3 are not "aware" of one another. Neither `UnionMod` nor `HashSetMod` statically depends upon the other's interface. Therefore, `HashSet` is unavailable during modular typechecks on `UnionMod` and `union` is unavailable during modular typechecks on `HashSetMod`, so neither module's typechecks ensure that `union` properly handles `HashSets`.

3.2 Implementation-side Typechecking and Modularity

Consider ITC for an EML module *m*. A straightforward approach to modular ITC checks each of *m*'s available functions *f* for exhaustiveness and unambiguity, given all available function cases and classes. We call this approach *naive modular ITC*. Unfortunately, naive modular ITC is unsound. The hierarchy of EML classes in figure 6 illustrates the kinds of problems that can occur. Naive modular ITC in `ShapeMod` checks `intersect` for exhaustiveness and unambiguity. Since `ShapeMod` doesn't statically depend upon any interfaces (other than its own), the check succeeds vacuously: `Shape` is abstract and so need not have an `intersect` implementation. Since `CircleMod` declares a new `intersect` case, `intersect` is again checked during naive modular ITC in `CircleMod`. `CircleMod` statically depends on the interface of `ShapeMod` but not that of `RectMod`, so `CircleMod`'s check does not consider the `Rect` class.[4] Therefore, the only argument to check from `CircleMod` is a pair of two `Circles`. The `intersect` case in `CircleMod` is most-specific for two `Circles`, so `intersect` is

[4]Indeed, `RectMod` may not even have been written when `CircleMod` is typechecked.

```
structure ShapeMod = struct
  abstract class Shape() of {}
  fun intersect:(#Shape * Shape) → bool
end
structure CircleMod = struct
  class Circle() extends Shape() of {}
  extend fun intersect(Circle _, Shape _) = ···
end
```

```
structure RectMod = struct
  class Rect() extends Shape() of {}
  extend fun intersect(Shape _, Rect _) = ···
  fun print:Shape → unit
  extend fun print(Rect _) = ···
end
```

Figure 6. Challenges for modular implementation-side typechecking.

found to be exhaustive and unambiguous. By similar reasoning, `intersect` passes the checks from `RectMod`, since `RectMod` does not statically depend on the interface of `CircleMod`.

Therefore each module typechecks, with naive modular ITC declaring the `intersect` function to be both exhaustive and unambiguous. However, `intersect` has neither of these properties. If `intersect` is invoked on a pair of a `Rect` and a `Circle` (in that order), a *match nonexhaustive* error will occur since neither `intersect` case is applicable. If `intersect` is invoked on a pair of a `Circle` and a `Rect` (in that order), a *match ambiguous* error will occur since both `intersect` cases apply but neither is more specific than the other.

A final problem concerns the `print` function in `RectMod`. Since `RectMod` does not statically depend on `CircleMod`'s interface, `RectMod`'s naive modular ITC finds `print` to be exhaustive and unambiguous. However, if a `Circle` is ever passed to `print`, a *match nonexhaustive* error will result.

3.3 Achieving Modular ITC

As we have seen, naive modular ITC is too permissive, allowing forms of extensibility that are not typesafe. To address this problem, we augment naive modular ITC with some requirements on EML modules that ensure the soundness of ITC. A fundamental design goal is that the requirements still allow the use of both functional and OO extensibility idioms in a single class hierarchy. We are willing to sacrifice other kinds of extensibility allowed by naive modular ITC to support the traditional functional and OO idioms in a modularly typesafe manner.

Functional languages allow a new function to be added to an existing datatype. Therefore, EML must allow a new function to be added to an existing class. OO languages allow a new subclass to be added to an existing class, along with associated overriding methods that have the new subclass as their receiver. To formulate this idiom in EML we employ a function's owner position, which generalizes a similar notion in the Dubious language [22]. A function's owner position has some properties in common with the receiver position in standard OO languages. Rather than forcing the owner position to be the "first" argument to a function, it can be specified as an arbitrary (and arbitrarily nested) position of the argument, via the `#` in a function's declared argument type. The type at the owner position in a function's argument type must be a class; that class is the function's *owner*. For example, `Set` is the owner of `add` in figure 1. To express the OO extensibility idiom in EML, we must allow a new subclass to be added to an existing class *C*, along with overriding cases of functions for which *C* is the owner.

For the purposes of our modular requirements, we partition functions into two categories. A function is called *internal* if it is declared in the same module as its owner; otherwise the function is *external*. An internal function is guaranteed to be available to all modules that declare subclasses of the function's owner, while that is not true of an external function. Therefore, an internal function can be thought of as part of the "initial" interfaces of its owner class and subclasses, while an external function is a later extension to those interfaces. External functions have no analogue in traditional OO languages, in which a class's methods must all be declared with the class. The special properties of internal functions are exploited in one of our three requirements, which are now discussed in turn.

3.3.1 Completeness Requirement for External Functions

Consider the completeness problem with the `print` function in `RectMod` in figure 6. Because new subclasses can be added to existing classes, some subclasses of a function's owner may not be available in the function's module. Indeed, `Circle` is not available in `print`'s module. On the other hand, because `print` is external, there is no guarantee that `print` will be available to all modules declaring subclasses of `Shape`. Indeed, `print` is not available to `Circle`'s module. Therefore, to modularly ensure that `print` is complete, we require its module to contain a *global default* case. A global default is a case whose pattern is applicable to all type-correct arguments to the function. In general, we require a module that declares an external function to include a global default case for the function.

Therefore, ITC on `RectMod` fails, because the global-default requirement is not satisfied for its external function `print`. If `print` had a case with, for example, pattern `(Shape {})`, then the requirement would be satisfied and the completeness problem for `Circle` would be avoided. As another example, the external function `union` in figure 2 satisfies the requirement because its first case is a global default, thereby handling the unavailable `HashSet` class of figure 3 and any other unavailable `Set` subclasses.

The global-default requirement does not impose an extra burden from the point of view of standard OO languages, as such languages do not even allow external functions to be declared. However, standard functional languages do allow external functions, without requiring global default cases. Those languages disallow data-variant extension, so an external function can be modularly checked against all possible data variants. EML's modular ITC must allow for the possibility of unavailable subclasses of a function's owner, thereby sometimes requiring the declaration of global default cases that will never be used. Section 5 introduces a mechanism for *sealing* class hierarchies, which can obviate the need for global default cases.

3.3.2 Completeness Requirement for Internal Functions

Consider the incompleteness for a pair of one `Rect` and one `Circle` in the internal `intersect` function of figure 6. One way to solve the problem would be to require a global default case, as we require for external functions. Indeed, if `ShapeMod` contained an `intersect` case that is applicable to any pair of `Shapes`, the incompleteness would be resolved. While requiring global default cases solves the problem, it is unnecessarily burdensome. As mentioned earlier, an internal function is guaranteed to be available to all modules declaring subclasses of the function's owner. Therefore, rather than requiring the function's module to handle all unknown subclasses, we can require each module that declares a concrete subclass of the function's owner to ensure completeness for its subclass. This idea is inspired by standard OO languages, in which a method in an abstract class may safely remain unimplemented, with each concrete subclass declaring or inheriting a concrete implementation of the method.

Our requirement is that each module declaring a concrete subclass *C* of an internal function's owner must also declare or inherit a *local default* case for the function. A local default case of a class *C* is a case whose pattern accepts only instances of *C* and subclasses at the owner position, while every other argument position can be passed any value of the appropriate type. Local default cases are the EML analogue of traditional OO methods, which dispatch on the surrounding class at the receiver position and do not dispatch on any other argument position. A class's local default cases ensure that the class completely implements all of the functions in its "initial" interface.

Given the local-default requirement, ITC on `RectMod` fails to typecheck because it does not declare or inherit a local default `intersect` case for `Rect`. (An isomorphic error would occur in `CircleMod` if the second argument position in the pair were designated the owner position.) The requirement would be satisfied, for example, if `RectMod` had an `intersect` case with pattern `(Rect _, Shape _)`, accepting `Rects` at the owner position and accepting all `Shapes` in the other position. That case resolves the incompleteness for a pair of one `Rect` and one `Circle`. A global default case need not be written: `intersect` may still be safely left unimplemented for two `Shapes`. As another example, the internal `add` function in figure 1 does not have a global default case. Instead, it has local default cases for its two concrete subclasses `ListSet` and `CListSet`. When `HashSet` is introduced in figure 3, an associated local default is also declared, satisfying the requirement and ensuring that `add` is complete for `HashSets`.

The local-default requirement does not impose an extra burden from the point of view of standard OO languages. Whenever a local default case of some internal function *f* is required for a class *C*, an OO language would require *C*'s declaration to contain an *f* method, so that *C* is properly implemented. Therefore, the abstract-class idioms of traditional OO languages are preserved in EML. However, standard functional languages do allow internal functions, without requiring local default cases. As above, this is possible because such languages disallow data-variant extension. EML's ITC must always assume the possibility of unavailable subclasses of classes in non-owner positions of a function's argument type, thereby sometimes requiring the declaration of local default cases that will never be used. Again, we can use sealing, discussed in section 5, to obviate the need for local default cases.

3.3.3 Ambiguity Requirement

In figure 6 the two `intersect` cases are ambiguous, but neither `CircleMod` nor `RectMod` statically depends upon the other, so the ambiguity is not modularly detected. We address this problem by restricting EML's function extensibility such that cases declared in modules that do not statically depend upon one another are guaranteed to be *disjoint*: the cases are not applicable to a common value and hence are not ambiguous. Our restriction generalizes the implicit restrictions in standard functional and OO languages. First we introduce the concept of a function case's *owner*, which is the class (if any) at the owner position of the case's pattern. For example, `ListSet` is the owner of the second `union` case in figure 2 because it appears at the owner position, while the first `union` case has no owner.

In functional languages, each case must be declared in the module that declares the associated function. In OO languages, each method must be declared inside the method's receiver. Our requirement is the disjunction of these conditions: every function case must either be declared in the module that declares the case's function or in the module that declares the case's owner (if any).

`RectMod` now fails to typecheck because its `intersect` case does not satisfy our requirement: neither `intersect` nor `Shape`, the case's owner, is declared in `RectMod`. (An isomorphic error would occur in `CircleMod` if the second argument position in the pair were designated the owner position.) Therefore, `RectMod` may not extend `intersect` in that way. The requirement can be satisfied, for example, by modifying the `intersect` case's pattern to `(Rect _, Shape _)`. This modification resolves the ambiguity for a pair of a `Circle` and a `Rect`, since the revised case is no longer applicable. As another example, the `add` cases in `HashSetMod` and `SortedListSetMod` of figures 3 and 4 are never compared for ambiguity, because the two modules do not statically depend upon one another. However, each case satisfies our requirement by following the traditional OO idiom of implementing an overriding method for a newly declared subclass. Therefore the two cases are guaranteed to be disjoint.

Since our ambiguity requirement is the disjunction of the implicit requirements in standard functional and OO languages, our requirement does not restrict those programming styles and allows them to coexist. Therefore, we have achieved our design goal of allowing the functional and OO extensibility idioms in a single class hierarchy while preserving modular type safety.[5] However, other useful kinds of extensibility are disallowed by the ambiguity requirement. For example, a client of both `UnionMod` and `HashSetMod` from figures 2 and 3 may want to implement `union` specially for `HashSets`, so that these independent extensions of the `Set` abstraction will work well together. However, the new case would violate our ambiguity requirement, so `HashSets` are forced to use the default `union` case (or `HashSetMod` must be modified in place to add the new case).

4 Mini-Eml

This section describes MINI-EML, a core language used to formalize the fundamental ideas in EML. We give the full dynamic semantics but only a brief introduction to the static semantics. The complete details of MINI-EML are available in our companion technical report [21].

[5] In the presence of multiple *implementation* inheritance, other kinds of ambiguities that elude modular detection can arise, necessitating an extra requirement [23]. However, multiple *interface* inheritance, as in Java, cannot cause such ambiguities.

4.1 Syntax

Figure 7a defines the syntax of types, expressions, and patterns in MINI-EML. The syntax is essentially that of EML as informally presented so far, but we omit standard constructs including base types, conditionals, lambdas, local variables, references, and exceptions. The domain *Mt* represents *marked types*, which contain a `#` mark on a single component class type. The *instance expression* $Ct\ \{\overline{V} = \overline{E}\}$ is not available at the source level, as instances may only be created via a constructor call $Ct(\overline{E})$. The construct $\{\overline{V} = \overline{E}\}$ differs from an ordinary record in two ways. First, the labels are *scoped*: the name of the structure in which an instance variable was introduced becomes part of the instance variable's name. In the presence of the ability to make instance variables private (see section 5), scoping allows subclasses to introduce a new instance variable without conflicting with the name of a hidden one in the superclass. Instance variables in EML use this mechanism implicitly; regular static scoping rules determine which instance variable is referred to. Second, for simplicity the components of $\{\overline{V} = \overline{E}\}$ are ordered, unlike traditional records.

The notation and semantic style of MINI-EML were influenced by Featherweight Java [17], a core language for Java. As in that language, we formally represent classes by their names. A class is uniquely represented as *Sn.Cn*, where *Cn* is the name of the class and *Sn* is the name of the structure that declares *Cn*. Extensible functions are represented similarly.

The subset of expressions that are MINI-EML values is described by the following grammar, which includes class instances, function values, and tuple values:

$$v ::= Ct\ \{\overline{V} = \overline{v}\} \mid Fv \mid (\overline{v})$$

The syntax of structures and declarations is shown in figure 7b. For convenience in the core language, each structure explicitly names the other structures (often including itself) whose interfaces it statically depends upon, via the `depends upon` $\overline{Sn}$ clause. ITC for a structure employs only the interfaces of the structures named in the `depends upon` clause. The static semantics ensures that the given dependency relation is well-formed, as described below. The syntax of the three declarations is faithful to that of EML, except that cases now contain a *case name Mn*. This name is used in the semantics to uniquely identify each function case declaration (see section 4.2).

Analogous with Featherweight Java, a MINI-EML program is a pair of a *structure table* and an expression. A structure table is a finite function from structure names to the associated structure declarations. The semantics assumes a fixed structure table denoted *ST*. The structure table *ST* is accessed by the dynamic and static semantics rules when information about a given OO declaration is required. The domain of a structure table *ST* is denoted dom(*ST*).

4.2 Dynamic Semantics

MINI-EML's dynamic semantics is defined as a mostly standard small-step operational semantics. The metavariable ρ ranges over *environments*, which are finite functions from identifiers to values. We use $|\overline{D}|$ to denote the length of the sequence $\overline{D}$. The notation $[I_1 \mapsto E_1, \ldots, I_k \mapsto E_k]D$ denotes the expression resulting from the simultaneous substitution of E_i for each occurrence of I_i in D, for $1 \leq i \leq k$, and similarly for $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k]D$. We use $[\overline{I} \mapsto \overline{E}]D$ as a shorthand when $\overline{I}$ and $\overline{E}$ have the same length, and similarly for $[\overline{\alpha} \mapsto \overline{\tau}]D$. In a given inference rule, fragments enclosed in <> must either be all present or all absent, and similarly for <<>>. We sometimes treat sequences as if they were sets. For example, $Ood \in \overline{Ood}$ means that *Ood* is one of the declarations in $\overline{Ood}$. We use $Ood \in ST(Sn)$ as shorthand for $ST(Sn) =$ `structure` $Sn =$ `struct depends upon` $\overline{Sn}\ \overline{Ood}$ `end` and $Ood \in \overline{Ood}$.

Figure 8a contains the rules for evaluating expressions. For simplicity in the semantics, E-NEW defines constructor calls as syntactic sugar for instance expressions. It would be straightforward to instead use a call-by-value semantics for constructor calls, at the cost of some additional mechanism. E-NEW makes use of the first two auxiliary rules in figure 8b. CONCRETE checks that the class to be instantiated was declared without the `abstract` keyword. REP initializes the fields of the new instance as directed by the class's implicit constructor.

The last rule in figure 8b formalizes function-case lookup, used in E-APPRED. The top line of LOOKUP's premises specifies the case to invoke, and the second line ensures that the chosen case is applicable. The remaining premise ensures that the chosen case is most-specific: the case is strictly more specific than any other applicable case. The condition $Sn.Mn \neq Sn'.Mn'$ uses the case names to ensure that the chosen case is not compared for specificity with itself.

The rules for pattern matching and specificity are shown in figure 9, completing the dynamic semantics. The matching rules are straightforward except for E-MATCHCLASS. The notation $C \leq C'$ denotes that (C, C') is in the reflexive, transitive closure of the declared class `extends` relation. E-MATCHCLASS recursively pattern matches on the instance variables, unlike traditional OO languages and $\text{ML}_{\leq}$. We allow an instance to have more instance variables than the given representation pattern, so that subclass instances can match superclass patterns. For example, the value `CListSet {es=[5,3],count=2}` matches the pattern in the `elems` case of figure 1.

The judgment $Pat \leq Pat'$ means that *Pat* is at least as specific as Pat'. The pattern specificity semantics generalizes OO-style "best-match" semantics to support ML-style patterns. Class pattern specificity (SPECCLASS) follows the ordering induced by subclassing. Analogous with E-MATCHCLASS, the more-specific pattern may contain extra instance variables. The natural rule SPECTUP for tuple patterns makes pattern specificity a generalization of the "symmetric" *multimethod* specificity semantics in OO languages [6, 7]. When a tuple is used to send multiple arguments to a function, tuple patterns allow all arguments to be dynamically dispatched upon, and no argument position is more important than the rest. This contrasts with traditional *single dispatch*, as in Java, where only a unique *receiver* argument may be dispatched upon.

4.3 Static Semantics

Figure 10 contains the rules for typechecking structures and OO declarations. Γ is a *type environment*, mapping identifiers to types. The notation $\hat{Mt}$ denotes the type τ equivalent to *Mt*, but with the `#` mark removed. Structures are typechecked (STRUCTOK) by checking each declaration in turn. It is assumed that *S* OK holds for each structure *S* in the range of *ST*.

The rules for typechecking the three OO declarations are largely straightforward. The premises rely on several kinds of judgments. A judgment of the form $\overline{\alpha} \vdash \tau$ OK ensures that τ is a well-formed

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid Ct \mid \tau_1 \to \tau_2 \mid \tau_1 * \cdots * \tau_k \\
Mt & ::= & \# \, Ct \mid \tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k \\
E & ::= & I \mid Fv \mid E_1\ E_2 \mid Ct(\overline{E}) \mid (\overline{E}) \mid Ct\ \{\overline{V} = \overline{E}\} \\
Pat & ::= & \_ \mid I \texttt{ as } Pat \mid C\ \{\overline{V} = \overline{Pat}\} \mid (\overline{Pat}) \\
Ct & ::= & \overline{\tau}\, C \qquad\qquad Fv ::= \overline{\tau}\, F \\
C & ::= & Sn.Cn \qquad\quad V ::= Sn.Vn \\
F & ::= & Sn.Fn
\end{array}
$$

(a)

$$
\begin{array}{rcl}
S & ::= & \texttt{structure } Sn = \\
 & & \quad \texttt{struct depends upon } \overline{Sn}\ \overline{Ood} \texttt{ end} \\
Ood & ::= & \texttt{<abstract> class } \overline{\alpha}\ Cn(\overline{I : \tau}) \\
 & & \quad \texttt{<<extends } Ct(\overline{E}) \texttt{>> of } \{\overline{Vn : \tau_0 = E_0}\} \\
 & \mid & \texttt{fun } \overline{\alpha}\ Fn : Mt \to \tau \\
 & \mid & \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E
\end{array}
$$

(b)

Figure 7. (a) MINI-EML types, expressions, and patterns; (b) MINI-EML structures and declarations. Metavariable α ranges over type variable names, I over identifier names, Sn over structure names, Cn over class names, Vn over instance variable names, Fn over function names, and Mn over case names. $\overline{D}$ denotes a comma-separated list of elements (and is independent of any variable named D). Angle brackets (<>) and double angle brackets (<<>>) denote independent optional pieces of syntax. The notation $\overline{V} = \overline{E}$ abbreviates $V_1 = E_1, \ldots, V_k = E_k$ where $\overline{V}$ is $V_1, \ldots, V_k$ and $\overline{E}$ is $E_1, \ldots, E_k$ for some $k \geq 0$, and similarly for $\overline{V} = \overline{Pat}$, $\overline{Vn : \tau_0 = E_0}$, and $\overline{I : \tau}$.

$\boxed{E \longrightarrow E'}$

$$
\frac{Ct = (\overline{\tau}\, C) \qquad \mathrm{concrete}(C) \qquad \mathrm{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E_1}\}}{Ct(\overline{E_0}) \longrightarrow Ct\ \{\overline{V} = \overline{E_1}\}}\ \textsc{E-New}
$$

$$
\frac{E \longrightarrow E'}{\begin{array}{c} Ct\ \{\overline{V_0} = \overline{v_0}, V = E, \overline{V_1} = \overline{E_1}\} \longrightarrow \\ Ct\ \{\overline{V_0} = \overline{v_0}, V = E', \overline{V_1} = \overline{E_1}\}\end{array}}\ \textsc{E-Rep}
$$

$$
\frac{E \longrightarrow E'}{(\overline{v_0}, E, \overline{E_1}) \longrightarrow (\overline{v_0}, E', \overline{E_1})}\ \textsc{E-Tup}
$$

$$
\frac{E_1 \longrightarrow E_1'}{E_1\ E_2 \longrightarrow E_1'\ E_2}\ \textsc{E-App1} \qquad \frac{E_2 \longrightarrow E_2'}{v_1\ E_2 \longrightarrow v_1\ E_2'}\ \textsc{E-App2}
$$

$$
\frac{\text{most-specific-case-for}(Fv, v) = (\{(\overline{I}, \overline{v})\}, E)}{Fv\ v \longrightarrow [\overline{I} \mapsto \overline{v}]E}\ \textsc{E-AppRed}
$$

(a)

$\boxed{\mathrm{concrete}(C)}$

$$
\frac{(\texttt{class}\ \overline{\alpha}\ Cn \ldots) \in ST(Sn)}{\mathrm{concrete}(Sn.Cn)}\ \textsc{Concrete}
$$

$\boxed{\mathrm{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E}\}}$

$$
\frac{\begin{array}{c}(\texttt{<<abstract>>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I : \tau_1})\ \texttt{<extends}\ Ct(\overline{E_0})\ \texttt{>} \\ \texttt{of}\ \{\overline{Vn : \tau_2 = E_2}\}) \in ST(Sn) \\ <\mathrm{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E_1}\}>\end{array}}{\mathrm{rep}((\overline{\tau}\ Sn.Cn)(\overline{E})) = [\overline{I} \mapsto \overline{E}][\overline{\alpha} \mapsto \overline{\tau}]\{<\overline{V} = \overline{E_1}, > Sn.\overline{Vn} = \overline{E_2}\}}\ \textsc{Rep}
$$

$\boxed{\text{most-specific-case-for } (Fv, v) = (\rho, E)}$

$$
\frac{\begin{array}{c}(\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E) \in ST(Sn) \\ \mathrm{match}(v, Pat) = \rho \\ \forall Sn' \in \mathrm{dom}(ST).\forall(\texttt{extend fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat' \ldots) \in ST(Sn'). \\ \forall \rho'.(\mathrm{match}(v, Pat') = \rho' \wedge Sn.Mn \neq Sn'.Mn' \\ \Rightarrow Pat \leq Pat' \wedge Pat' \not\leq Pat)\end{array}}{\text{most-specific-case-for } ((\overline{\tau}\, F), v) = (\rho, [\overline{\alpha} \mapsto \overline{\tau}]E)}\ \textsc{Lookup}
$$

(b)

Figure 8. (a) Evaluation rules for expressions. (b) Auxiliary inference rules. The notation $(\overline{I}, \overline{v})$ abbreviates $(I_1, v_1), \ldots, (I_k, v_k)$; $Sn.\overline{Vn} = \overline{E}$ abbreviates $Sn.Vn_1 = E_1, \ldots, Sn.Vn_k = E_k$.

$\boxed{\mathrm{match}(v, Pat) = \rho}$

$$
\frac{}{\mathrm{match}(v, \_) = \{\}}\ \textsc{E-MatchWild}
$$

$$
\frac{\mathrm{match}(v, Pat) = \rho}{\mathrm{match}(v, I \texttt{ as } Pat) = \rho \cup \{(I, v)\}}\ \textsc{E-MatchBind}
$$

$$
\frac{C \leq C' \qquad \mathrm{match}(\overline{v}, \overline{Pat}) = \overline{\rho}}{\mathrm{match}(\overline{\tau}\, C\ \{\overline{V} = \overline{v}, \overline{V_1} = \overline{v_1}\}, C'\ \{\overline{V} = \overline{Pat}\}) = \bigcup \overline{\rho}}\ \textsc{E-MatchClass}
$$

$$
\frac{\mathrm{match}(\overline{v}, \overline{Pat}) = \overline{\rho}}{\mathrm{match}((\overline{v}), (\overline{Pat})) = \bigcup \overline{\rho}}\ \textsc{E-MatchTup}
$$

(a)

$\boxed{Pat \leq Pat'}$

$$
\frac{}{Pat \leq \_}\ \textsc{SpecWild}
$$

$$
\frac{Pat_1 \leq Pat_2}{I \texttt{ as } Pat_1 \leq Pat_2}\ \textsc{SpecBind1} \qquad \frac{Pat_1 \leq Pat_2}{Pat_1 \leq I \texttt{ as } Pat_2}\ \textsc{SpecBind2}
$$

$$
\frac{C \leq C' \qquad \overline{Pat_1} \leq \overline{Pat_2}}{C\ \{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\} \leq C'\ \{\overline{V} = \overline{Pat_2}\}}\ \textsc{SpecClass}
$$

$$
\frac{\overline{Pat_1} \leq \overline{Pat_2}}{(\overline{Pat_1}) \leq (\overline{Pat_2})}\ \textsc{SpecTup}
$$

(b)

Figure 9. (a) Pattern matching. (b) Pattern specificity. The notation $\mathrm{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$ abbreviates $\mathrm{match}(v_1, Pat_1) = \rho_1 \cdots \mathrm{match}(v_k, Pat_k) = \rho_k$, and similarly for $\overline{Pat_1} \leq \overline{Pat_2}$.

$\boxed{S \text{ OK}}$

$$\frac{\overline{Sn} \vdash \overline{Ood} \text{ OK in } Sn}{\texttt{structure } Sn = \texttt{struct depends upon } \overline{Sn}\ \overline{Ood} \texttt{ end OK}} \text{ STRUCTOK}$$

$\boxed{\overline{Sn} \vdash Ood \text{ OK in } Sn}$

$$\frac{\begin{array}{c} < Ct = \overline{\alpha}\ Sn.Cn > \qquad < \Gamma;\overline{\alpha} \vdash Ct(\overline{E}) \text{ OK} > \\ \overline{\alpha} \vdash \overline{\tau} \text{ OK} \qquad \overline{\alpha} \vdash \overline{\tau_0} \text{ OK} \qquad \Gamma = \{(\overline{I},\overline{\tau})\} \qquad \Gamma;\overline{\alpha} \vdash \overline{E_0} : \overline{\tau_1} \\ \overline{\tau_1} \leq \overline{\tau_0} \qquad \overline{Sn} \vdash Sn.Cn \text{ transDependedUpon} \\ \text{concrete}(Sn.Cn) \Rightarrow \overline{Sn} \vdash \text{funs-have-ldefault-for } Sn.Cn \end{array}}{\begin{array}{c} \overline{Sn} \vdash <<\texttt{abstract}>> \texttt{ class } \overline{\alpha}\ Cn(\overline{I} : \overline{\tau}) \\ <\texttt{extends } Ct(\overline{E}) > \texttt{ of } \{\overline{Vn} : \overline{\tau_0} = \overline{E_0}\} \text{ OK in } Sn \end{array}} \text{ CLASSOK}$$

$$\frac{\begin{array}{c} \overline{\alpha} \vdash \hat{Mt} \text{ OK} \qquad \overline{\alpha} \vdash \tau \text{ OK} \qquad \text{owner}(Sn.Fn) = Sn'.Cn \\ Sn \neq Sn' \Rightarrow \overline{Sn} \vdash Sn.Fn \text{ has-gdefault} \end{array}}{\overline{Sn} \vdash \texttt{fun } \overline{\alpha}\ Fn : Mt \rightarrow \tau \text{ OK in } Sn} \text{ FUNOK}$$

$$\frac{\begin{array}{c} (\texttt{fun } \overline{\alpha'}\ Fn : Mt \rightarrow \tau) \in ST(Sn') \\ \text{matchType}([\overline{\alpha'} \mapsto \overline{\alpha}]\hat{Mt}, Pat) = (\Gamma, \tau_0) \qquad \Gamma;\overline{\alpha} \vdash E : \tau' \\ \tau' \leq [\overline{\alpha'} \mapsto \overline{\alpha}]\tau \qquad \overline{Sn} \vdash Sn'.Fn \text{ dependedUpon} \\ Sn;\overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat = E \text{ unambiguous} \end{array}}{\overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat = E \text{ OK in } Sn} \text{ CASEOK}$$

Figure 10. Static semantics of structures and OO declarations. The notation $\overline{Sn} \vdash \overline{Ood}$ OK in Sn abbreviates $\overline{Sn} \vdash Ood_1$ OK in Sn $\cdots$ $\overline{Sn} \vdash Ood_k$ OK in Sn; $\overline{\alpha} \vdash \overline{\tau}$ OK abbreviates $\overline{\alpha} \vdash \tau_1$ OK $\cdots$ $\overline{\alpha} \vdash \tau_k$ OK; $(\overline{I},\overline{\tau})$ abbreviates $(I_1,\tau_1),\ldots,(I_k,\tau_k)$; $\Gamma;\overline{\alpha} \vdash \overline{E} : \overline{\tau}$ abbreviates $\Gamma;\overline{\alpha} \vdash E_1 : \tau_1 \cdots \Gamma;\overline{\alpha} \vdash E_k : \tau_k$; $\overline{\tau_1} \leq \overline{\tau_0}$ abbreviates $\tau_{11} \leq \tau_{01}$ $\cdots$ $\tau_{1k} \leq \tau_{0k}$.

type: only type variables in $\overline{\alpha}$ are referred to in τ, and each class in τ has the correct number of type parameters. A judgment of the form $\tau_1 \leq \tau_2$ denotes that τ_1 is a subtype of τ_2; the subtyping relation is completely standard [5]. A judgment of the form $\Gamma;\overline{\alpha} \vdash E : \tau$ ensures that expression E has type τ, in the context of the current type environment and sequence of type variables in scope. A judgment of the form $\Gamma;\overline{\alpha} \vdash Ct(\overline{E})$ OK is used in CLASSOK to ensure that the superclass constructor call is well-formed. A judgment of the form matchType$(\tau, Pat) = (\Gamma, \tau')$ ensures that a case's pattern *Pat* is compatible with the associated function's declared argument type τ; the type environment Γ contains bindings for the identifiers in *Pat* and is used to typecheck the case's body.

The "transDependedUpon" and "dependedUpon" judgments in CLASSOK and CASEOK check properties of a structure's declared `depends upon` relation. The first judgment ensures that a structure containing a class is declared to depend upon all structures that declare a (reflexive, transitive) superclass of the class. The second judgment ensures that a structure containing a function case is declared to depend upon the structure containing the associated function. In either case, if Sn is required to declare a dependency on Sn', then Sn does indeed statically depend upon Sn' according to the definition of static dependency given in section 3.1. The declared dependency relation may include more structures than are statically depended upon, but the soundness proof relies only on the above two properties of the declared dependency relation, thereby ensuring that modularity is respected.

Finally, each of the rules for typechecking declarations enforces one of the three modular requirements described in section 3.3. CLASSOK enforces the local-default requirement on the new class ("funs-have-ldefault-for"), if it is concrete. FUNOK enforces the global-default requirement on the new function ("has-gdefault"), if it is external. CASEOK enforces the ambiguity requirement and explicitly checks ambiguities of the new case with any available function cases ("unambiguous").

```
structure BadMod = struct
  class C() of {}
  fun f:C → unit
  val bad = f(C())
  extend fun f (C {}) = ()
end
```

Figure 11. Value declarations and ITC.

4.4 Type Soundness

We have proven type soundness for MINI-EML. As usual, we prove type preservation and progress theorems. The notation $\vdash E : T$ denotes the typechecking of E in the context of the empty type environment and empty sequence of type variables.

THEOREM 1. *(Type Preservation) If* $\vdash E : \tau$ *and* $E \longrightarrow E'$, *then there exists* τ' *such that* $\vdash E' : \tau'$ *and* $\tau' \leq \tau$.

THEOREM 2. *(Progress) If* $\vdash E : \tau$ *and* E *is not a value, then there exists* E' *such that* $E \longrightarrow E'$.

Proving type preservation is straightforward, as it is completely independent of ITC. Proving progress requires reasoning about modular ITC, in order to show that function applications can always make progress. The key lemma says that a most-specific applicable function case exists for each type-correct application:

LEMMA 1. *If* $\vdash Fv : \tau_2 \rightarrow \tau$ *and* $\vdash v : \tau_2'$ *and* $\tau_2' \leq \tau_2$, *then there exist* ρ *and* E *such that* most-specific-case-for $(Fv,v) = (\rho, E)$.

Details are available in our companion technical report [21].

5 ML-Style Modules

This section discusses how EML's features can interact with an ML-style module system including structures, signatures, and functors.

5.1 Structures

Thus far we have assumed that EML structures contain only a sequence of class, function, and function case declarations. We would also like to accommodate the ordinary ML declarations, including value, type, exception, and structure declarations. The latter three kinds of declarations can be straightforwardly incorporated, but special care is needed to handle value declarations. Figure 11 shows an example of the problems that can occur. ITC on `BadMod` will succeed, because function `f` has an appropriate case for `C`. However, at run-time a *match nonexhaustive* error will occur when the `val` declaration is executed, because `f`'s function case will have not yet been declared.

There are several approaches to handling this problem. We could adopt a two-pass style of structure evaluation. The first pass would evaluate all of the declarations except the value declarations, and the second pass would evaluate the value declarations. In our example, this semantics ensures that `f`'s function case is declared before `f` is invoked. An alternative approach is to make the unit of modularity used in our ITC requirements more fine-grained than an entire structure, with `val` declarations forming the boundaries of

```
signature ShapeSig = sig
  abstract class Shape() of {}
  fun bad:Shape → unit
  extend fun bad s
end
structure ShapeMod = struct
  abstract class Shape() of {}
  fun print:Shape → unit
  fun bad:Shape → unit
  extend fun bad s = print s
end :  ShapeSig
structure CircleMod
  class Circle() extends Shape() of {}
end
```

Figure 12. Unsoundnesses with hiding OO declarations.

these units. For example, `BadMod` would consist of two units, one of which contains the first two declarations and the other containing the last declaration. When ITC is performed on the first unit, the incompleteness of `f` for `C` would result in a static error. Our prototype EML interpreter uses a variant of this approach. Instead of inferring the modular units, we introduce a new kind of OO declaration of the form *Ood* `and` *Ood*′ (similar syntactically, but not semantically, to the `and` construct in ML), which groups a sequence of class, function, and function case declarations. A group of `and`ed OO declarations is treated as a unit for the purposes of modular ITC.

5.2 Signature Ascription

Signature ascription provides information hiding in ML. Clients of a structure expression of the form $S : Sig$, where *Sig* is a signature, may only access *S*'s components via the interface provided in *Sig*. Signature ascription for EML provides forms of OO-style encapsulation. For example, classes, functions, and function cases can be hidden from clients, making them private to their enclosing structure. However, these declarations cannot be hidden arbitrarily, or else modular ITC would become unsound. Figure 12 shows a simple example of the problems that can occur. `ShapeMod` creates the abstract `Shape` class and two associated functions. ITC in `ShapeMod` finds `print` to be exhaustive and unambiguous, since `Shape` is abstract. Ascription to the `ShapeSig` signature hides `print`. Therefore, `print` is not part of `ShapeMod`'s interface, so `print` is not available to `CircleMod` and is therefore not checked again for exhaustiveness and unambiguity. If a `Circle` instance is passed to `bad`, however, `print` will be invoked, causing a *match nonexhaustive* error.

Our example is purposely similar to the `print` example in figure 6. In that case, the ITC requirements ensure that the problem is modularly detected. The same solution can be used here: a set of declarations can be safely hidden if that set could have been written as a separate module that passes modular ITC [23]. The `print` function in figure 12 does not satisfy this condition. If `print` were in its own module, the type system would force the existence of a global default case for `print`, which is now an external function. If `print` had such a case, then the function (and that case) could be safely hidden via signature ascription, and the problem for `Circle` would be resolved.

Aside from hiding entire declarations, it is useful to hide certain properties of a declaration. Several properties of classes may be hidden. First, a subset of a class's instance variables may be hidden. As mentioned in section 4, instance variables are scoped — the name of the structure declaring an instance variable is implicitly part of the name of the instance variable. Therefore, there is no conflict if a subclass in a new module creates an instance variable of the same name as a hidden one in the superclass. A concrete class can also be viewed as an abstract one, thereby disallowing clients from instantiating the class. Finally, a signature can declare a class *C* `sealed` [29], which prevents classes declared outside of *C*'s module from directly subclassing *C*. This construct can be used to faithfully model ML-style (non-extensible) datatypes. Our modular requirements can be relaxed in the presence of sealed hierarchies. For example, if an external function's owner and all available subclasses are sealed, then the function need not have a global default case, as in ML.

A function may be sealed by ascribing it and all associated cases to an ordinary ML-style value specification. Clients may still invoke the function but its extensibility is hidden, so clients may not add new cases. Therefore, function sealing allows us to model ML-style (non-extensible) functions. Function sealing is allowed under the same circumstances that the function and its cases may be hidden. Finally, a specification of the form `val` $I : \tau$ may be replaced by `val` $I : \tau'$, where τ' is a supertype of τ.

Several forms of information hiding are not captured by our ascription rules. It would be useful to ascribe a class declaration to one that specifies only a transitive, rather than direct, superclass. Unfortunately, this flexibility makes modular ITC unsound. For example, a client of two classes C and C' can write ambiguous function cases that appear to be disjoint, and therefore pass static checks, if the fact that C subclasses C' is hidden from the client. It would also be useful to ascribe a class declaration to a type declaration, possibly augmented with Modula-3-style *partial revelations* [24] to reveal some of the class's underlying structure.

```
structure PointMod = struct
  abstract class Point()
  fun draw:Point → unit
end
signature APointSig = sig
  class APoint(x:int,y:int)
    extends Point of {x:int,y:int}
  extend fun draw (APoint {x=x,y=y})
end
functor Colorize(M:APointSig) = struct
  class ColorPoint(x:int,y:int,color:int)
    extends M.APoint(x,y) of {color:int=color}
  extend fun draw
    (ColorPoint {x=x,y=y,color=color}) = ...
  fun getColor:ColorPoint → int
  extend fun getColor
    (ColorPoint {x=x,y=y,color=color}) = color
end
```

Figure 13. Idioms involving EML functors.

5.3 Functors

In the presence of EML's features, functors can provide a great deal of flexibility. Figure 13 illustrates the kinds of idioms we would like to express. The `Colorize` functor implements a form of *mixin* [4, 11, 14], which is a class parameterized by its superclass. The functor creates a colored version of some unknown subclass `APoint` of `Point`. An overriding case for the existing `draw` function is given, in order to draw colored points specially. The functor also introduces a new function for accessing the color of a colored point, with an associated case.

We would like to perform modular ITC once on a functor body, guaranteeing completeness and unambiguity of all relevant functions no matter how the functor is instantiated. The major challenge for modular ITC of functors like `Colorize` is the fact that the identities of some classes, for example `M.APoint`, are unknown. Instead we have only partial information about the relationship between `M.APoint` and other classes. To address this challenge, we can generalize the subclass relation in the static semantics to be *three-valued*, conservatively saying "don't know" when the partial class hierarchy information is inconclusive. We then appropriately generalize modular ITC to be conservative with respect to three-valued subclassing. Consider performing ITC on the body of `Colorize`. Although the identity of `M.APoint` is unknown, its relationship to `ColorPoint` is known, and this is enough information for modular ITC on `draw` to succeed. We have formalized this three-valued semantics in an earlier version of MINI-EML but have not proven it sound.

The restrictions on signature ascription described earlier limit the expressiveness of our `Colorize` functor. For example, the functor can only be instantiated with a class `APoint` that is a direct subclass of `Point`, rather than a transitive one. Also, `APoint`'s module must contain a `draw` case with exactly the pattern described in `APointSig`, and the module can have no other `draw` cases for `APoint` (e.g. a special case to handle the origin). However, we can safely remove these restrictions if we are willing to move some of the burden of ITC to clients of the functor. For example, we can allow `APoint` to be instantiated with a transitive subclass of `Point` on the condition that the resulting structure passes modular ITC. In the limit, this approach performs modular ITC once per instantiation of the functor, where the identities of all classes are known, rather than once on the functor body. However, it is possible that most of ITC could still be performed on the functor body in isolation, with only a few additional checks performed per instantiation.

6 Related Work

OML [27] and $ML_{\leq}$ [3] were described earlier. Zenger and Odersky [30] describe an extensible datatype mechanism in the context of an OO language. Extending a datatype has the effect of creating a new datatype that subtypes from the original one. To ensure exhaustiveness in the presence of datatype extension, all functions on extensible datatypes must include a global default case, while EML often requires only local defaults. Because Zenger's functions are not extensible, if new data variants require overriding function cases, a new function must be created that inherits the existing function cases and clients must be modified to invoke the new function. Like OML, Zenger's language includes both OO-style methods and ML-style functions. Zenger's language also retains a distinction between datatype "cases" and regular OO classes. Because Zenger's language supports subtyping between entire datatypes (rather than individual variants), it can provide more precise types than EML.

Garrigue shows how to use *polymorphic variants*, which are variants defined independent of any particular datatype, to obtain both modular data-variant and function extensibility in ML [15]. However, unlike EML, both kinds of extensibility require advance planning. When defining a type as a set of polymorphic variants, an extra type parameter must be used in place of recursive references to the type, to allow for future extension. Similarly, a function must take an extra parameter function to invoke in place of recursive references. As in Zenger's language, when a function is extended any clients that require the new functionality must be modified. Unlike EML, polymorphic variants preserve ML-style type inference.

Previous work on unifying functional and OO dispatching [10] provides ITC for patterns that are more general than those in EML, including conjunctions, disjunctions, and negations of arbitrary predicates. However, the ITC algorithm requires access to the entire program.

Jiazzi [20], a component system for Java, addresses issues of signature ascription and parameterized modules in the context of a traditional OO language. Jiazzi disallows hiding abstract methods because of problems analogous to the one shown in figure 12. Jiazzi also restricts the hiding of a superclass relationship, like EML, but Jiazzi allows such hiding if the superclass itself is also hidden. EML and Jiazzi each have challenges for information hiding that have no analogue in the other system: EML's unique challenges arise from its generalization of OO and functional dispatching semantics, and Jiazzi's unique challenges arise from cyclic linking.

EML's modular requirements are adapted from our previous work on Dubious [22, 23], a multimethod-based OO calculus supporting modular typechecking. In EML, we have generalized the requirements to fit an ML context and have also substantially simplified both their informal and formal presentations. The notion of modularity in Dubious is coarser than EML's static dependency relation: a Dubious module requires access to more of the program to soundly perform ITC than does an EML module. Dubious does not consider patterns, polymorphism, or ML-style modules.

7 Conclusions and Future Work

We described Extensible ML, an ML-like language that supports hierarchical, extensible datatypes and functions. Such constructs allow for the easy addition of both new data variants and new operations to existing abstractions, resolving a long-standing tension between the functional and object-oriented styles. At the same time, EML retains completely modular typechecking of function implementations. This contrasts with previous languages based on extensible datatypes and functions, which require link-time checks to ensure type safety. We have formalized EML in MINI-EML and proven its type system sound.

There are several directions for future work. We have built a prototype interpreter for the core of EML, and we plan to pursue case studies to gauge the utility of our modular type system in practice. Currently EML does not allow aliasing of classes or extensible functions. A general approach to handling aliasing would allow classes and extensible functions to be less second-class. Finally, more work is needed to integrate EML with ML-style modules, particularly functors. We will pursue the ideas presented in section 5, formalize this extension in MINI-EML, and implement it in our interpreter.

8 Acknowledgments

Thanks to Jonathan Aldrich, Sorin Lerner, and Vass Litvinov for helpful comments on the paper. This work was supported in part by NSF grant CCR-9970986, NSF Young Investigator Award CCR-9457767, gifts from Sun Microsystems and IBM, and a Wilma Bradley Graduate Fellowship.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[2] D. Bonniot. Type-checking multi-methods in ML (a modular approach). In *The Ninth International Workshop on Foundations of Object-Oriented Languages, FOOL 9*, Portland, Oregon, USA, January 2002.

[3] F. Bourdoncle and S. Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, Paris, France, 15–17 Jan. 1997.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[5] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, Feb. 1988. Also appeared in Semantics of Data Types, LNCS 173, 1984.

[6] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, Feb. 1995. A preliminary version appeared in *ACM Conference on LISP and Functional Programming*, June 1992 (pp. 182–192).

[7] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[8] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, Nov. 1995.

[9] W. R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, NY, 1991.

[10] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In E. Jul, editor, *ECOOP '98–Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

[11] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104. ACM, June 1998.

[12] K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 37–49, Atlanta, Georgia, May 1–4, 1999.

[13] K. Fisher and J. Reppy. Extending Moby with inheritance-based subtyping. In *14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 83–107, June 2000.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[15] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[18] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 28 Feb. 1997.

[19] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, editor, *ECOOP'98–Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer-Verlag, July 1998.

[20] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: new-age components for old-fashioned java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 211–222. ACM Press, 2001.

[21] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. Technical Report UW-CSE-02-07-05, Department of Computer Science and Engineering, University of Washington, July 2002. `ftp://ftp.cs.washington.edu/tr/2002/07/UW-CSE-02-07-05.pdf`.

[22] T. Millstein and C. Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *ECOOP '99 – Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer-Verlag, New York, NY, June 1999.

[23] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[24] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[25] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 Jan. 1997.

[26] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[27] J. Reppy and J. Riecke. Simple objects for Standard ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 171–180, Philadelphia, Pennsylvania, 21–24 May 1996.

[28] J. C. Reynolds. User defined types and procedural data structures as complementary approaches to data abstraction. In D. Gries, editor, *Programming Methodology, A Collection of Articles by IFIP WG2.3*, pages 309–317. Springer-Verlag, New York, NY, 1978.

[29] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[30] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*. ACM, September 3-5 2001.