Modular Typechecking for Hierarchically Extensible Datatypes and Functions

Todd Millstein Craig Chambers

Department of Computer Science and Engineering
University of Washington
{todd,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. Such a generalization allows existing datatypes to be easily extended with both new operations and new data 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 typechecks to ensure safety. We describe 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 certain restrictions on datatype and function extensibility, but EML is still able to express both traditional functional and OO idioms. EML's extensible functions integrate ML-style pattern matching with OO multimethod dispatching, and EML supports an ML-style module system including signature ascription and functors.

1 Introduction

The functional and object-oriented (OO) styles confer incomparable benefits to programmers. For example, many researchers have noted a difference in the extensibility offered by datatypes and classes [25, 7, 21, 9, 14, 26]. 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 any new datatype constructors). On the other hand, traditional object-oriented approaches allow new data variants to be easily added to existing class hierarchies (by defining 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).

For this and other reasons, there have been several recent research efforts to integrate the benefits of the functional and OO styles in the context of ML. OCaml [23], for example, adds OO features including class and method definitions to ML. The OO constructs essentially form their own sub-language which is largely separate from the existing ML `datatype` and `fun` constructs. In a similar vein, Pizza [21] adds ML-style datatypes and functions (among other things) to Java.

Adding a set of new constructs has the advantage that existing language constructs are only 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 kinds of extensibility and reuse allowable for the abstraction. It may be difficult to determine *a priori* which kinds of extensibility will be required, and it is difficult to change the decision after the fact.

An alternative approach to extending ML with new OO constructs is to generalize existing constructs to support the object-oriented style. OML [24], 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.

$ML_{\leq}$ [2] 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 whose patterns encode OO-style dynamic dispatching. In this approach, programmers need not choose between incomparable forms of expressiveness; a single language mechanism supports the easy addition of both new operations and new variants to existing datatypes.

However, there are several ways in which $ML_{\leq}$ is not well integrated with existing ML language features. First, $ML_{\leq}$ restricts the patterns in function cases to be datatype constructor tests, which are the analogue of dynamic dispatch tests in OO languages. Other ML-style patterns, for example record patterns, cannot be programmed. Second, $ML_{\leq}$ does not address the ML module system. The authors sketch a source-level language including a simple module system, but constructs such as signature ascription and functors are not handled.

Third, extensible datatypes naturally require functions to be extensible, so that as new data variants are declared, existing functions can be updated with cases to handle the new variants. However, $ML_{\leq}$ does not support extensible functions; the typing rules assume that all function cases are provided when the function is declared. The authors sketch a source-level language that supports extensible functions, with function cases spread across different modules. Unfortunately, this critical generalization of their work causes a loss of *modular* static typechecking.

A key component of typechecking is the assurance that all operations are exhaustively and unambiguously implemented, which we call *implementation-side* typechecking [6]. In functional languages, this corresponds to checking each function for *match nonexhaustive* and *match redundant* errors. In OO languages, this corresponds to checking that each class has a unique *most-specific* method implementation to invoke for each supported operation. In $ML_{\leq}$, exhaustiveness and unambiguity of extensible functions can only be assured when all modules are available. For this reason, static typechecking of a program cannot be completed until link-time [2]. This loss of modular typechecking forces programmers to give up important software engineering benefits, 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.

In this work, we describe an ML-like language called Extensible ML (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:

- EML generalizes the OO dispatching semantics in $ML_{\leq}$ to allow arbitrary ML-style patterns. This generalization provides idioms that are not handled by either traditional functional or OO languages.
- EML supports an ML-style module system [15, 19]. *Structures* are module implementations, containing a sequence of declarations, possibly including nested structures. *Signatures* are module interfaces, and *signature ascription* can be used to hide information about a structure from clients. In the presence of extensible datatypes and functions, care must be taken to ensure the type safety of such information hiding. Finally, *functors* are a form of parametrized module. Typechecking a functor is challenging because its module body can create subclasses and functions for a class described in its parameter signature, whose identity is unknown. This combination of functors and classes provides the ability to program some kinds of *mixins* [3, 9, 10], classes parametrized by their superclass.
- EML supports extensible functions while preserving purely modular typechecking: each module can be typechecked given only the *interfaces* (signatures) of the modules it depends upon, with no link-time checks required. We introduce *blocks*, which are modules that contain only OO declarations. Blocks generalize class declarations in traditional OO languages and are a natural unit on which to perform implementation-side typechecking. To make per-block implementation-side typechecking sound without requiring link-time checks, EML's type system imposes certain restrictions via the notion of a *distinguished position* in a function's argument, which serves to coordinate otherwise independent extensions to the function. Despite these restrictions, EML's classes and functions are still able to express both traditional functional and OO idioms. The extensibility restrictions are adapted from our earlier work on Dubious [16, 18], a calculus designed to explore modular typechecking for OO languages based on multimethods.

```
SetMod = block
  abstract class Set of {}
  class ListSet extends Set of {es:int list}
  class CListSet extends ListSet of {count:int}

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

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

  fun isEmpty:Set → bool
  extend fun isEmpty s = (size s)=0
  extend fun isEmpty (CListSet {es=_,count=0}) = true

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

Figure 1: A simple hierarchy of integer sets in EML.

The rest of the paper is organized as follows. Section 2 informally presents the core of the language, which omits polymorphism, signature ascription, and functors. Section 3 discusses the challenges for performing modular static typechecking, and section 4 describes our modular type system that addresses these challenges. Section 5 extends EML to handle parametric polymorphism, and section 6 extends EML to handle signature ascription and functors. Section 7 discusses related work, and section 8 concludes. A companion technical report [17] contains formal dynamic and static semantics and discusses formal properties of the language.

2 The Core of EML

2.1 Examples

Figure 1 shows an EML implementation of integer sets. A `block` contains a sequence of class, function, and function case declarations that form a logical unit. EML also supports ML-style `struct`s, which may be nested and may contain `block`s.

2.1.1 Classes and Representations

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, 12], EML contains *interfaces*; 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}`.

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

```
datatype DT = C1 of T1 | ··· | Cn of Tn
```

(where each T_i is a record type) is syntactic sugar in EML for the following class declarations:

```
abstract class DT of {}
class C1 extends DT of T1
...
class Cn extends DT of Tn
```

For example, the declarations of the `Set` and `ListSet` classes in figure 1 could have been written equivalently as follows:

```
datatype Set = ListSet of {es:int list}
```

Unlike the constructors in ordinary ML datatypes, classes may be used as types, and other classes may subclass from them.

Concrete classes are *instantiated* the same way that values of ML datatypes are created. For example, the result of `ListSet {es=[2,1]}` is an instance of `ListSet` representing the set {2,1}. 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.

2.1.2 Functions and Function Cases

To allow functions to be extensible, we break an ML-style function into two pieces: a `fun` declaration which introduces the function, and zero or more `extend fun` declarations which specify the function's cases. Figure 1 declares functions for adding an element to a set, accessing a set's size, checking whether a set is empty, and accessing a list representation of a set's elements. The `size` function, for example, is declared to accept an instance of `Set` or a subclass and to return an integer. In a traditional OO language, the `size` function would be introduced implicitly via the declaration of an abstract `size` method in the `Set` class. The `#` in the `add` function's argument type is required as part our modular typechecking restrictions, discussed in section 4; it has no dynamic effect. Functions can be passed to and returned from other functions, like ordinary lambdas. However, a function's extensibility is second-class: new cases may only be added to statically known functions.

```
UnionMod = block
  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 {es=merge(sort(e1), sort(e2))}
end
```

Figure 2: Adding new functions in EML.

The `extend fun` declaration specifies the name of the function being extended, a pattern guard, and the new case's body. The 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 independent blocks to extend the same function, just as independent subclasses in OO languages can declare methods that conceptually update the same "generic function." There are two `size` function cases in figure 1, handling `ListSet`s and `CListSet`s, respectively. In a traditional OO language, these `size` cases would be declared as `size` methods in the `ListSet` and `CListSet` class declarations. Because `Set` is declared `abstract`, it need not have an applicable `size` function case. Patterns in EML subsume both OO-style dynamic dispatching and ML-style pattern matching. For example, the second `isEmpty` case in figure 1 is only applicable dynamically if the argument is an instance of `CListSet` or a subclass, whose `count` instance variable has the value 0.

Dynamically, an OO-style "best-match" policy is used to decide which function case to invoke; their order does not matter. For example, both `size` cases are applicable to the invocation `size(CListSet {es=[1,2],count=2})`, 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, in other modules, as new data variants are defined.

An ordinary ML-style function consisting of n function cases is syntactic sugar for a `fun` declaration followed by n `extend fun` declarations. For example, the sugared version of the `size` function and associated cases in figure 1 is as follows:

```
fun size (ListSet {es=es}) = length es
  | size (CListSet {es=_,count=c}) = c
```

2.1.3 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` block. (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.) Two `union` function cases are provided. The first is a *global default* case, as its pattern is applicable to any pair of `Set`s. The second `union` case provides a more efficient implementation for two `ListSet`s.

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

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

```
SortedListSetMod = block
  class SListSet extends ListSet of {}

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

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

  fun getMin:SListSet → int
  extend fun getMin (SListSet {es=es}) =
    if (null es) then raise Empty else hd(es)
end
```

Figure 4: Class hierarchies in EML.

2.1.4 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. Existing applicable `isEmpty` cases are inherited for use by this subclass. In a traditional OO language, `HashSetMod` corresponds to the declaration of a new subclass of `Set` with some overriding methods. `HashSetMod` and `UnionMod` from figure 2 are completely independent: either, both, or neither could be linked into the final program. If both are present, then `HashSet` implicitly inherits any applicable `union` function cases.

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. The `SListSet` class inherits the representation type of `ListSet` (adding no new instance variables) as well as the applicable function cases of `size`, `isEmpty`, 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.

2.2 Syntax

2.2.1 Types, Expressions, and Patterns in EML

Figure 5(a) has the syntax for EML types, expressions, and patterns. For simplicity, we omit many standard constructs, including base types, conditionals, ordinary lambdas, local variables, references, and exceptions.

The T non-terminal represents EML types. A class type C is either a simple class name Cn or has the form $Pn.Cn$, where Pn names a (possibly nested) block that defines a class named Cn. EML includes record types (and expressions); a tuple type (expression) of length n is syntactic sugar for a record type (expression) with labels $1,\ldots,n$. Unlike in ML, the components within an EML record are *ordered*; this restriction simplifies the semantics of class representation inheritance. The Mt non-terminal represents *marked* types, which contain a `#` mark on a single component class type. Marked types are used to implement the modular typechecking restrictions discussed in section 4.

EML expressions are fairly standard. The expression (C E) denotes the instantiation of class C, with representation E. Path names Pn are used to access module component expressions other than extensible functions, while non-terminal F plays a similar role for extensible functions.

The pattern `_` is the wildcard. The pattern (I `as` Pat) binds the identifier I to the given value and matches the value to Pat; the pattern I is syntactic sugar for (I `as` `_`). The class pattern C Pat matches an instance of C or a subclass, whose representation matches Pat. The pattern $\{L_1 = Pat_1, \ldots, L_n = Pat_n\}$ matches a record containing components named $L_1,\ldots,L_n$, each of whose associated value matches the respective sub-pattern.

The subset of expressions that are EML values is described by the following grammar: $v ::= C\ v \mid F \mid \{L_1 = v_1, \ldots, L_n = v_n\}$

2.2.2 Structures and Blocks in EML

The syntax of programs and modules is shown in figure 5(b). An EML program is a sequence of modules followed by a single expression. The `structs` are akin to ML-style structures, containing name bindings for expressions and other modules. EML also includes `blocks`, which contain only class, function, and function case declarations. While `structs` may be nested and may also contain `blocks`, `blocks` are flat. The argument type of an extensible function must be marked; the purpose of the mark is discussed in section 4.

2.3 Function Application

To evaluate an expression (E_1 E_2), E_1 is evaluated to a function F, E_2 to a value v_2, and F's *most-specific applicable case* for v_2 is found and invoked. The most-specific applicable case is defined to be the applicable case that is at least as specific as every other applicable case. A case is *applicable* to a value v if v matches the case's pattern (described below). A case with pattern Pat_1 is *as specific as* a case with pattern Pat_2 if Pat_1 is as specific as Pat_2 (described below).

The rules for matching a value to a pattern are largely straightforward. Every value matches the wildcard pattern, and identifier binding has no effect on pattern matching. A class value (C v) matches a class pattern (C' Pat) if C subclasses C' and v matches Pat. Pattern matching for records supports a form of *width subtyping*, whereby the record value

$$
\begin{array}{rcl}
T & ::= & C \mid \{L_1 : T_1, \ldots, L_n : T_n\} \\
 & \mid & T_1 \rightarrow T_2 \\
Mt & ::= & \texttt{\#}C \mid \{L_1 : T_1, \ldots, L_{i-1} : T_{i-1}, L_i : Mt, \\
 & & \qquad L_{i+1} : T_{i+1}, \ldots, L_n : T_n\} \\
C & ::= & < Pn. > Cn \\
E & ::= & C\ E \mid Pn \mid F \mid E_1\ E_2 \\
 & \mid & \{L_1 = E_1, \ldots, L_n = E_n\} \\
F & ::= & < Pn. > Fn \\
Pat & ::= & \_ \mid I\ \texttt{as}\ Pat \mid C\ Pat \\
 & \mid & \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \\
Pn & ::= & < Pn. > I
\end{array}
$$

(a)

$$
\begin{array}{rcl}
P & ::= & [M]; E \\
M & ::= & I = S \\
S & ::= & \texttt{struct}\ [D]\ \texttt{end} \mid \texttt{block}\ [Ood]\ \texttt{end} \\
D & ::= & I = E \mid I = S \\
Ood & ::= & <\texttt{abstract}>\ \texttt{class}\ Cn \\
 & & \quad <\texttt{extends}\ C >\ \texttt{of}\ T \\
 & \mid & \texttt{fun}\ Fn : Mt \rightarrow T \mid \texttt{extend fun}\ F\ Pat = E
\end{array}
$$

(b)

Figure 5: (a) Types, expressions, and patterns; (b) Programs and modules. Metavariable *Cn* ranges over class names, *Fn* ranges over function names, *L* ranges over labels, and *I* ranges over all other kinds of identifiers. Brackets ([]) denote a comma-separated list of zero or more items. Angle brackets (<>) denote an optional piece of syntax; multiple optional pieces of syntax in a given production are independent of one another.

may have extra components at the end that are not present in the record pattern. This is necessary so that subclass representations can match superclass representation patterns. For example, the value `CListSet {es=[2],count=1}` matches the pattern in the `elems` case of figure 1.

The pattern specificity semantics generalizes OO-style method specificity to handle functional-style patterns. Every pattern is as specific as the wildcard pattern. Identifier binding has no effect on specificity. We define $(C\ Pat)$ to be as specific as $(C'\ Pat')$ if C is a strict subclass of C', or if $C = C'$ and Pat is as specific as Pat'. This semantics favors the class over the representation; if C is a strict subclass of C', the representations are ignored for determining specificity. Other semantics are plausible and would not substantively affect our results. Finally, record pattern specificity requires the component patterns to be pointwise in the specificity relation. This natural rule makes pattern specificity a generalization of the "symmetric" *multimethod* specificity semantics in OO languages [4, 5]. When a record is used to simulate sending multiple arguments to a function, record 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 is dispatched upon.

Consider the function application

```
union(SListSet {es=[1,2]}, ListSet {es=[3,1]})
```

in the context of the code in figures 1 through 4. The `union` function has three cases. The `union` case in figure 4 is not applicable: `SListSet {es=[1,2]}` matches `SListSet {es=e1}`, but `ListSet {es=[3,1]}` does not match `SListSet {es=e2}` since `ListSet` is not a subclass of `SListSet`. Of the two remaining cases, the second one in figure 2 is the unique most-specific applicable case, so it is invoked. If a function application had no most-specific applicable case, a *match nonexhaustive* error would occur. If an application had more than one most-specific applicable case, a *match ambiguous* error would occur. Our static typechecking rules ensure that neither of these errors can occur.

3 Modular Typechecking for EML

This section defines the notion of modular typechecking for EML and describes the challenges for making such checking sound.

3.1 Definition of Modular Typechecking

We say that a language's typechecking scheme is *modular* if each module m can be typechecked given only the module *interfaces* (without requiring access to the associated module implementations) that m *depends upon*. Module m depends upon interface i if one of the following conditions holds:

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

Traditional functional languages support modular typechecking. For example, each structure in ML can be typechecked given only its depended-upon structure interfaces (signatures). A structure's interface is either an explicitly ascribed signature or else the structure's *principal signature*, which is informally the signature that lists all components and gives each component its most-specific type. Similarly, each class in a standard OO language can be typechecked given only the depended-upon class interfaces. Informally, the interface of a class consists of its list of superclasses, the types of its subclass-visible fields, and the headers, but not bodies, of its subclass-visible methods.

Modular typechecking schemes provide several important software-engineering benefits, in contrast with a global typechecking scheme. Programming errors are detected earlier in the software development process. Libraries can be typechecked once and safely used in multiple contexts that meet their interface requirements. Independent developers can create and typecheck their own modules in isolation, with a guarantee that type safety is preserved when the modules are linked into the same application. Modification of a module does not require a global re-typecheck of the program.

To consider a typechecking scheme for EML to be modular, each structure and block must be typechecked completely given only the interfaces it depends upon. Figure 3.1

$$
\begin{array}{rcl}
M & ::= & \ldots \mid I = \mathit{Sig} \\
\mathit{Sig} & ::= & I \mid \texttt{sig}\ [\mathit{Spec}]\ \texttt{end} \\
 & \mid & \texttt{sigblock}\ [\mathit{Oospec}]\ \texttt{end} \\
\mathit{Spec} & ::= & I : T \mid I : \mathit{Sig} \\
\mathit{Oospec} & ::= & \langle\texttt{abstract}\rangle\ \texttt{class}\ \mathit{Cn} \\
 & & \quad \langle\texttt{extends}\ C\ \rangle\ \texttt{of}\ T \\
 & \mid & \texttt{fun}\ \mathit{Fn} : \mathit{Mt} \rightarrow T \mid \texttt{extend fun}\ F\ \mathit{Pat} \\
 & \mid & \texttt{val}\ \mathit{Fn} : \mathit{Mt} \rightarrow T
\end{array}
$$

Figure 6: Signatures and signature blocks in EML.

describes EML's module interfaces. A `sig` is an ML-style signature and is used as the interface of an EML structure, as in ML. EML also contains `sigblock`s, which specify only classes, functions, and function cases (the `val` specification is discussed in section 6.1). Consistent with Standard ML, signatures and signature blocks may only be defined at top-level. The interface of a block b is either an explicitly ascribed *signature block* (block ascription is described in section 6.1) or b's *principal signature block*, the signature block identical to b but with all bodies removed from function cases. This paper focuses on modular typechecking for blocks; modular typechecking for structures is standard.

We say that blocks b and b' are *related* if one depends upon the other's interface; otherwise they are *unrelated*. Classes, functions, and cases declared in a block b are said to be *local* to b; all other classes, functions, and cases are *non-local*. Classes, functions, and cases that are local to a block b or are specified in an interface on which b depends are said to be *available* during the typechecking of b; all other classes, functions, and cases are *unavailable*.

3.2 Implementation-side Typechecking and Modularity

One part of typechecking an EML program is to ensure that no *match nonexhaustive* or *match ambiguous* errors will occur. We refer to this checking as implementation-side typechecking, which requires that for each function, each type-correct argument has a unique most-specific applicable function case to invoke. (Implementation-side typechecking contrasts with *client-side* typechecking, which checks that each function application expression in the program is type-correct, among other things. Client-side typechecking in EML is standard and is not discussed further.)

Each of the three OO declarations in EML impacts implementation-side typechecking. The `fun` declaration introduces a new extensible function to be checked for exhaustiveness and unambiguity. The `extend fun` declaration adds a function case to an existing function, thereby potentially affecting that function's exhaustiveness and unambiguity. The `class` declaration introduces a new class whose instances may be passed to existing functions, thereby potentially affecting their exhaustiveness and unambiguity. Further, these declarations typically interact in order to make a function exhaustive and unambiguous. For example, the `size` function in figure 1 is incomplete as soon as it is declared; it only becomes complete after the declaration of the `size` function case for `ListSet`. The declaration of class `HashSet` in figure 3 causes `size` to become incomplete again, until the `size` function case for `HashSet` is declared.

Therefore, we would like to group several OO declarations together, only performing implementation-side typechecking once, after all the declarations have been processed. In order to make this approach sound, we must disallow function applications within the group of declarations. Otherwise, such applications could result in a function being invoked that is not guaranteed to be exhaustive and unambiguous until the entire group of declarations has been executed, thereby potentially causing a match nonexhaustive or match ambiguous error. It is for this reason that we introduce blocks as an alternative module construct. A block allows a group of related classes, functions, and function cases to be declared, without intervening function applications, and to be implementation-side typechecked as a unit. In this way, blocks generalize the structuring provided by class declarations in a standard OO language.

In the context of a modular typechecking scheme, implementation-side typechecking of a block b does not have access to the entire program, but only to b's available classes, functions, and function cases. For example, the `HashSetMod` block (figure 3) depends on the interface of the `SetMod` block (figure 1) but does not depend on the interfaces of either `UnionMod` (figure 2) or `SortedListSetMod` (figure 4). Therefore, implementation-side typechecking from `HashSetMod` must check the `add`, `size`, `isEmpty`, and `elems` functions for exhaustiveness and unambiguity, but it will not check (or even be aware of) the `union` function. Further, the unavailable `SListSet` class and associated `add` case will not be considered during this implementation-side typechecking.

3.3 A Modular Implementation-Side Typechecking Algorithm

This subsection describes the algorithm for modular implementation-side typechecking of blocks. Implementation-side typechecking is partitioned into exhaustiveness and ambiguity checks. Exhaustiveness checking of a block ensures that for each available function, every available type-correct argument has *at least* one available applicable case. To check each available function F, we generate a set of patterns s with the property that each available type-correct argument to F matches at least one pattern in s. We say that s *covers* each available type-correct argument to F. The algorithm for generating set s is detailed in our companion technical report [17]. Finally, we check that each pattern in s is as specific as some available function case's pattern. If the check succeeds, then every available type-correct argument to F has at least one applicable function case to invoke.

As an example, consider exhaustiveness checking of `size` from `HashMod` in figure 3. The pattern set to check will be {`ListSet {es=es}`, `HashSet {ht=ht}`}, as this set covers the three available concrete subclasses of `Set`. Each of these patterns is as specific as one of the patterns in an available `size` function case, so the check succeeds.

Per-block ambiguity checking ensures that for each available function, every available type-correct argument has *at most* one available most-specific applicable case. To perform ambiguity checks for the available function F, we check each pair of its available function cases for ambiguity in turn. Consider a pair of available cases c_1 and c_2. The cases are not ambiguous if one of the following three conditions holds: (1) The cases are *disjoint*, meaning that their patterns cannot match a common value. (2) One case is strictly more specific than the other. (3) There exists a third available case c that is strictly more specific than c_1 and c_2 and covers their *intersection*, meaning that c applies to every value to which both c_1 and c_2 apply.

For example, consider ambiguity checking for add from HashSetMod in figure 3. The add case in HashSetMod is disjoint from the first add case in SetMod because neither of HashSet and ListSet is a subclass of the other, so the cases are not ambiguous. Similarly, the add case in HashSetMod is disjoint from the second add case in SetMod. The two add cases in SetMod are not disjoint, because they both apply to arguments consisting of a pair of an integer and a CListSet instance. However, the second add case is strictly more specific than the first one, so the cases are not ambiguous.

3.4 Challenges for Modular Implementation-Side Typechecking

Unfortunately, per-block implementation-side typechecking alone is not sufficient to ensure the absence of match nonexhaustive and match ambiguous errors in EML programs. The hierarchy of shapes in figure 7 illustrates the kinds of problems that can occur. Implementation-side typechecking from the ShapeMod block checks the intersect function for exhaustiveness and unambiguity. The check succeeds vacuously: since Shape is abstract, there are no possible type-correct arguments to intersect, and there are no function cases to check for ambiguity.

CircleMod depends on the interface of ShapeMod, so intersect is again checked during implementation-side typechecks from CircleMod. Because CircleMod does not depend on the interface of RectMod, the Rect class is not available. Therefore, the only argument type to check is a pair of two Circles; as above, Shapes need not be considered because they are abstract. The intersect case in CircleMod is applicable to two Circles and ambiguity checking succeeds vacuously, so intersect is found to be exhaustive and unambiguous. By similar reasoning, intersect passes the checks from RectMod, since RectMod does not depend on the interface of CircleMod.

Therefore, a modular implementation-side typechecking scheme will declare the intersect function to be both exhaustive and unambiguous. However, intersect has neither of these properties. Invoking intersect on a pair of a Rect and a Circle (in that order) will cause a *match nonexhaustive* error, since neither intersect case applies to the argument. Passing a pair of a Circle and a Rect (in that order) will cause a *match ambiguous* error, since both intersect cases apply but neither is as specific as the other.

A final problem concerns the print function in RectMod. Implementation-side typechecks from the RectMod block find print to be exhaustive and unambiguous. In particular, there is a single concrete subclass Rect of Shape available to RectMod, and there is a unique applicable case for this argument type. However, if a Circle is ever passed to print, a *match nonexhaustive* error will result.

4 Extensibility Restrictions for Modular Typechecking

As described in the previous section, per-block implementation-side typechecking of *unrestricted* EML programs is not sufficient prevent match nonexhaustive and match ambiguous errors. To resolve the problem, we impose some restrictions on EML's extensibility that, along with per-block implementation-side typechecking, ensure soundness of the modular type system. These restrictions rely on the notion of the *distinguished position* (DP) of a function's argument, generalizing a similar notion in the Dubious language [16]. A function's DP serves to coordinate otherwise independent extensions of the function. As such, it has some properties in common with the receiver position in standard OO languages. Rather than forcing the DP to be the "first" argument to a function, however, the DP can be specified as an arbitrary subcomponent of the argument. The location of a function's DP has no dynamic effect.

Our key observation is that the implicit restrictions on a receiver in standard OO languages (for example, that the receiver of a method must be the enclosing class) allow for modular typechecking in such languages. In a similar vein, we impose restrictions on the DP that allow blocks to be typechecked modularly. These restrictions can be partitioned into those affecting exhaustiveness checking and those affecting ambiguity checking, detailed in sections 4.1 and 4.2, respectively.

The # mark on an extensible function's argument type identifies the DP of the associated function. For example, the second element of the argument pair to intersect in figure 7 is in the DP. In general, the DP may be arbitrarily nested within a function's argument. As a syntactic sugar, the DP is assumed to be the entire function argument if no # mark is specified. This sugar is used in our examples.

4.1 Exhaustiveness Checking

Consider the *match nonexhaustive* error for one Rect and one Circle in the intersect function of figure 7. The incompleteness is not modularly detectable from CircleMod by the per-block exhaustiveness algorithm in section 3.3 because the only available concrete subclass of Shape is Circle, so the pattern set to check will be simply {(Circle _, Circle _)}. One way to resolve the problem would be to force each function to contain a global default case, a case that is applicable to all type-correct arguments. In our example, intersect would be forced to have a case that is applicable to any two Shapes, which thereby also handles one Rect and one Circle. While requiring functions to contain global default cases would make modular exhaustiveness checking sound, it effectively defeats the purpose of allowing classes to be abstract by requiring such classes to be fully implemented. Our observation is that it is safe to instead require a set of *local default* cases. Such cases are effectively ordinary OO-style methods: they have a *single* argument position that dispatches on a particular concrete subclass of the declared type at that position, while every other argument position is applicable to any value of the appropriate type. Each such case is introduced in the same block as its associated concrete subclass. This relaxation is inspired by standard OO languages, which safely allow (singly-dispatched) methods with an abstract receiver to remain unimplemented, with concrete subclasses providing appropriate implementations.

In terms of our exhaustiveness algorithm, this restriction amounts to treating abstract classes as if they were concrete for the purposes of pattern-set generation, in all argument positions except one. We use the DP as the position where abstract classes may be ignored. Exhaustiveness checking on intersect in CircleMod must therefore treat Shape as concrete in the first argument position (but not in the second argument position, which is the DP). The generated pattern set to check will then include the pattern (Shape _, Circle _), thereby causing the exhaustiveness check to fail. Adding the local default case for Circle in CircleMod allows

```
ShapeMod = block
  abstract class Shape of {}
  fun intersect:(Shape * #Shape) → bool
end
CircleMod = block
  class Circle extends Shape
    of {radius:real,center:Point}
  extend fun intersect(Circle _, Shape _) = ···
end
```

```
RectMod = block
  class Rect extends Shape
    of {lowerLt:Point,upperRt:Point}
  extend fun intersect(Shape _, Rect _) = ···
  fun print:Shape → unit
  extend fun print(Rect _) = ···
end
```

Figure 7: Challenges for modular implementation-side typechecking. The implementation of `Point` is not shown.

its exhaustiveness check to succeed and resolves the incompleteness for a pair of one `Rect` and one `Circle`. A global default case need not be written. For example, `intersect` may still be safely unimplemented for two `Shape`s.

Now consider the incompleteness for the `print` function in `RectMod`. Because `Rect` is the only available concrete subclass of `Shape`, the pattern set to check is {`Rect _`}. Therefore, the incompleteness for `Circle` is undetected. Note that `RectMod` already contains a local default `print` case for `Rect`, so the above restriction is not sufficient in this situation. The problem is that `print` was declared in `RectMod`, but its argument type `Shape` was declared non-locally. Therefore it is possible for there to exist concrete subclasses of `Shape` to which `print` is not available, e.g. `Circle`. In this situation, we must require `print` to contain a global default case. In terms of our algorithm, `Shape` must be considered concrete for exhaustiveness checking even at the DP. The pattern set to check will then include the pattern `Shape _`, causing the exhaustiveness check to fail and forcing the global default case to be written. In general, the restriction forces *external functions*, functions whose argument type at the DP is a non-local class, to contain a global default case. Note that standard OO languages do not allow external functions to be written at all (where the receiver is the DP). Functional languages allow external functions lacking global defaults, but only because they disallow data-variant extension (see section 6.1 for EML techniques that recover the functional style).

These two restrictions, local defaults for *internal functions* (whose argument type at the DP is a local class) and global defaults for external functions, are enough to make the modular per-block exhaustiveness checks of section 3.3 sound. At the same time, the allowable abstract-class idioms in EML are still a superset of the abstract-class idioms possible in a standard OO language. All of the per-block exhaustiveness checks in figures 1 through 4 succeed. Therefore, we are assured through purely modular typechecking that *match nonexhaustive* errors cannot occur in that program.

4.2 Ambiguity Checking

The unsoundness described in section 3.4 for ambiguity checking occurs because it is possible for function cases defined in unrelated blocks to be ambiguous with one another. Because the blocks are unrelated, neither block includes the other's case in its ambiguity checks. In figure 7, for example, the two `intersect` cases are ambiguous, but `CircleMod` and `RectMod` are unrelated so the ambiguity is not detected. To handle this problem without sacrificing modularity, we restrict the kinds of cases that may be written such that cases in unrelated blocks are guaranteed to be disjoint. Our restriction is inspired by an implicit restriction in standard OO languages, where every method must dispatch on the enclosing class at the receiver position. This implicit restriction ensures (assuming only single inheritance) that sibling classes cannot define ambiguous methods, because such classes can have no common descendants. In our terms, the restriction ensures that all cases dispatch in the DP on a locally declared class. Our restriction generalizes this implicit one to support writing function cases for external functions. If a case is declared in the same block as its function F, the case may be unrestricted, as the case is necessarily available whenever F is checked for ambiguity. In this way, ordinary ML-style function cases are unrestricted. On the other hand, if a case is declared in a block b other than the block declaring F, we require the case to dispatch in the DP on a locally-declared class.

All of the per-block ambiguity checks in figures 1 through 4 succeed. Therefore, we are guaranteed that *match ambiguous* errors cannot occur. In addition, the ambiguity error in figure 7 is now modularly detected. Ambiguity checking from `CircleMod` signals an error on the block's `intersect` case, since the case fails our restriction: `intersect` was not declared in `CircleMod`, but neither was the `Shape` class, which is dispatched upon at the DP. (An isomorphic error would occur in `RectMod`'s ambiguity checks if the first argument position in the pair were the DP.) Our restriction can be satisfied, for example, by modifying the case's pattern to `(Shape _, Circle _)`, and indeed this resolves the ambiguity for a pair of a `Circle` and a `Rect` since the revised case is no longer applicable.

4.3 Formal Properties

The formal static typechecking rules of EML can be found in our companion technical report [17]. We have proven a type preservation theorem for the core of EML and are currently adapting our proof of progress for the Dubious language [16] to EML.

5 Parametric Polymorphism

This section augments EML to support parametric polymorphism. The revised EML syntax is shown in figure 5. Class, function, and function case declarations and specifications bind *type variables*, as does the class pattern. Each reference to a class or function then specifies a particular *type instantiation*. EML is explicitly typed; supporting practical polymorphic type inference in the presence of subtyping is the subject of ongoing research [11, 13, 20, 22].

Figure 9 shows a simple class hierarchy representing polymorphic lists (distinct from the built-in lists used in previous examples) in EML. `List` is parametrized by a single type variable `'a`, representing the element type. Sub-

$$
\begin{array}{rcl}
\mathit{Ood} & ::= & \texttt{<abstract>}\ \texttt{class}\ ([\mathit{Tv}])\ \mathit{Cn} \\
 & & \quad \texttt{<extends}\ C\ \texttt{>}\ \texttt{of}\ T \\
 & | & \texttt{fun}\ ([\mathit{Tv}])\ \mathit{Fn} : \mathit{Mt} \rightarrow T \\
 & | & \texttt{extend fun}\ ([\mathit{Tv}]) < \mathit{Pn} > .\mathit{Fn}\ \mathit{Pat} = E \\
\mathit{Oospec} & ::= & \texttt{<abstract>}\ \texttt{class}\ ([\mathit{Tv}])\ \mathit{Cn} \\
 & & \quad \texttt{<extends}\ C\ \texttt{>}\ \texttt{of}\ T \\
 & | & \texttt{fun}\ ([\mathit{Tv}])\ \mathit{Fn} : \mathit{Mt} \rightarrow T \\
 & | & \texttt{extend fun}\ ([\mathit{Tv}]) < \mathit{Pn} > .\mathit{Fn}\ \mathit{Pat} \\
 & | & \texttt{val}\ ([\mathit{Tv}])\ \mathit{Fn} : \mathit{Mt} \rightarrow T \\
\mathit{Pat} & ::= & \ldots \mid ([\mathit{Tv}])\ C\ \mathit{Pat} \\
C & ::= & ([T]) < \mathit{Pn} > .\mathit{Cn} \\
F & ::= & ([T]) < \mathit{Pn} > .\mathit{Fn}
\end{array}
$$

Figure 8: Polymorphism in EML. The metavariable Tv ranges over type variables.

```
ListMod = block
  abstract class ('a) List of {}
  class ('b) Nil extends ('b) List of {}
  class ('c,'d) Cons extends ('c) List
    of {hd:'c, tl:('c) List, tag:'d}

  fun ('e) len:  ('e) List → int
  extend fun ('f) len (('f) Nil _) = 0
  extend fun ('g) len
    (('h) (('g,'h) Cons) {hd=_, tl=tl, tag=_})
    = 1 + (('g) len)(tl)
  end
```

Figure 9: A class hierarchy representing lists. For clarity, distinct type variables have been given distinct names.

classes then inherit from type instantiations of `List`. The polymorphic class `('b) Nil` is declared to inherit from the instantiation of `List` with a type parameter of `'b`. For simplicity, type parameters are *invariant*; for example, `(T`$_1$`) Nil` is a subtype of `(T`$_2$`) List` if and only if `T`$_1$=`T`$_2$. Unlike constructors in ML datatypes, classes declare different type variables from their superclasses. For example, `Cons` has two type variables, one for the element type and another for the type of a "tag."

Functions are polymorphic as well, declaring a sequence of type variables. The `len` function is parametrized by the type variable `'e`, which is referred to in the function's type. Function cases are similarly polymorphic, declaring a sequence of type variables which can be referred to in class patterns, as in the first `len` case. Because subclasses have different type variables from their superclasses, class patterns may additionally have to bind new type variables. For example, the second `len` function case's class pattern binds the type variable `'h` for use in the pattern's instantiation of `Cons`.

The dynamic semantics of EML is designed to be insensitive to polymorphism: type variables and type parameters can be erased without affecting the execution of a program. For example, a class pattern dynamically applies to any instance of the associated class or a subclass (whose representation matches the associated representation pattern), regardless of the instance's type instantiation. Static typechecking ensures that an instance will never be matched to a class pattern whose type instantiation is incompatible.

Client-side typechecking in the augmented language is standard, so we focus on implementation-side typechecking. Our observation is that, since polymorphism has no dynamic effect and implementation-side typechecking consists in statically reasoning about the results of dynamic function-case lookup, implementation-side typechecking should be able to ignore polymorphism altogether. That is, we should be able to perform the modular implementation-side typechecking algorithm described in section 4 on the "polymorphism-erased" version of the program, while retaining modular type soundness. Such an approach obviates the need for implementation-side typechecking to explicitly consider all possible type instantiations of classes and cleanly separates distinct parts of the type system.

To make this approach sound, we must ensure that type parameters are never "dispatched" upon. A simple example illustrates the problem. Suppose the first `len` case of figure 9 were instead defined as follows:

```
extend fun ('f) len ((int) Nil _) = 0
```

Implementation-side typechecking would succeed for (the polymorphism-erased version of) `len`, since a unique most-specific applicable case exists for each of `Nil` and `Cons`. However, the revised `Nil` case expects the element type to be an `int` (and its body can make use of this expectation), even though dynamically `len` may be passed an argument like `(bool) Nil {}`, which will invoke the `Nil` case. Disallowing "dispatching" on type parameters rules out our revised `Nil` case and ensures that each function case's pattern is *fully general* with respect to the function's declared argument type. The restriction is enough to achieve type preservation for the core of EML augmented with parametric polymorphism.[1]

6 Signature Ascription and Functors

This section extends EML to handle signature ascription and functors. The augmented EML syntax is as follows:

$$
\begin{array}{rcl}
M & ::= & \ldots \mid \texttt{functor}\ I(I' : \mathit{Sig}) = S \\
S & ::= & \ldots \mid S : \mathit{Sig} \mid I(\mathit{Pn})
\end{array}
$$

The structure expression $S : Sig$ restricts a client's view of structure (block) S to the information specified in the signature (signature block) Sig. A functor takes a module as an argument and returns another module. As in ML, functor definitions are restricted to top-level. The syntax $I(Pn)$ instantiates functor I with the module denoted by the path name Pn.

6.1 Signature Ascription

Signature ascription is used in ML to support information hiding. Clients of a structure expression of the form $S : Sig$ may only access S's components via the interface provided in Sig. For example, a structure can be ascribed to a signature that does not mention some of the structure's components, effectively making those components private. Ascription for structures is standard and does not impact modular typechecking.

Signature ascription for blocks can encode forms of OO-style encapsulation. First, ascription for blocks supports hiding classes, functions, and function cases from clients. However, components cannot be hidden arbitrarily, or else

[1] In the presence of interfaces, which support a form of multiple inheritance, an additional restriction is required [17].

```
ShapeSig = sigblock
  abstract class Shape of {}
end
ShapeMod = block
  abstract class Shape of {}
  fun print:Shape → unit
end :  ShapeSig
CircleMod = block
  class Circle extends Shape of {}
end
```

Figure 10: Difficulties with signature ascription.

modular implementation-side typechecking would become unsound. Figure 10 shows a simple example of the problems that can occur. `ShapeMod` creates the abstract `Shape` class and an associated `print` function. Implementation-side typechecking of `print` in `ShapeMod` will succeed vacuously, since `Shape` is abstract. Ascription to `ShapeSig` hides `print`. Therefore, `print` is not part of the interface of `ShapeMod`, so `print` is not implementation-side typechecked by `CircleMod`. However, if a `Circle` instance were ever passed to `print`, a *match nonexhaustive* error would occur.

This example is purposely similar to the incompleteness example for `print` in figure 7. In that case, our restrictions from section 4 ensure that the incompleteness is caught modularly. The same solution can be used to provide component hiding in blocks via ascription: a set of declarations can be safely hidden if that set could have been written as a separate block whose associated implementation-side typechecks succeed [16]. The `print` function in figure 10 does not satisfy this condition. If `print` were in its own block, the associated exhaustiveness check would force the existence of a global default case for `print` according to the restrictions of section 4, because the `Shape` class is now non-local. If `print` had such a case, then the function (and that case) could be safely hidden via signature ascription, and the incompleteness for `Circle` would be resolved.

Aside from hiding entire declarations, it is useful to hide or restrict certain properties of a declaration. First, we augment class specifications to allow an optional leading `sealed` keyword, which disallows direct subclasses from being created outside of the class's block. This kind of ascription is similar in spirit to the `final` construct in Java and can be used to faithfully model ordinary ML-style (non-extensible) datatypes. Second, a concrete class may be ascribed to an `abstract` class specification, thereby disallowing clients from creating new instances. This feature can be used, for example, to program a *singleton*, a class guaranteed to have a single instance. The restrictions for modular exhaustiveness checking ensure the safety of this form of hiding.

Properties of functions may also be hidden. Like classes, a function can be `sealed`, thereby disallowing cases from being added to the function outside of its block. Sealing a function allows an ordinary (non-extensible) ML function to be faithfully modeled in EML. However, clients that declare new concrete classes whose instances can be passed to the function must perform exhaustiveness checks on the function, to ensure that arguments involving such instances are covered. If the function has a global default case, an alternative form of function sealing that relieves clients from having to perform exhaustiveness checks can be used. Specifically, a declaration of the form `fun` $I : Mt \rightarrow T$, and all associated function cases, can be ascribed to a signature block specification of the form `val` $I : Mt \rightarrow T$, thereby "forgetting" the function's extensibility. As with the use of `sealed`, clients may still invoke the function but may not add new cases.

6.2 Functors

Functors are used in ML to define structures that are parametrized by other structures. For example, a functor could implement a binary search tree, parametrized by a structure that provides the information to be stored in the nodes. A functor body is typechecked with respect to the signature of its parameter. The functor may be instantiated with any structure that can be ascribed to that signature.

Parametrized blocks provide a great deal of flexibility. For example, a functor body can define a subclass of a class specified in the parameter signature, thereby simulating a form of mixins. Similarly, a functor body can define a new function and associated function cases for an argument class. These kinds of idioms are a challenge for implementation-side typechecking of a functor body, because the identities of the classes specified in the parameter signature are unknown until the functor is instantiated. Instead, we have only partial knowledge of such classes, as provided by the signature. The identities of parameter functions are similarly unknown. In this way, functors introduce a form of class and function aliasing.

To handle function aliasing, we simply disallow a functor body from extending argument functions; adding cases to an unknown function appears to have limited utility. To handle class aliasing, we generalize the subclassing relation in the static semantics to be *three-valued*, conservatively saying "don't know" when the partial class hierarchy information is inconclusive. We then generalize implementation-side typechecking to be conservative in the presence of three-valued subclassing. Pattern generation for exhaustiveness checking generates patterns that cover all values that *may* be type-correct arguments, while the pattern specificity rules check that a pattern is definitely as specific as another one. The test for pattern disjointness in ambiguity checking succeeds only if two patterns are definitely disjoint, while the intersection of two patterns conservatively represents all values that *may* match both patterns. An ambiguity error is signaled if there is not enough information to accurately compute such an intersection.

Consider the functors in figure 11. The `Translate` functor creates a new operation for translating rectangles across the x-axis (implementation of `Rect` not shown). Ambiguity checking on `translateX` in `Translate` fails. In particular, the patterns of the two `translateX` cases are not definitely disjoint (since `RMod.R` is possibly the same as `M.R`) but their intersection is also not derivable (since neither class is necessarily a subclass of the other). Indeed, if `Translate` is ever instantiated with `RMod`, the resulting `translateX` function will be ambiguous for `RMod.R`. On the other hand, consider the `Color` functor, which creates a colored subclass of the parameter `Point` class. The functor also creates a new function for displaying a point. Although the identity of `M.Point` is unknown, its relationship to `ColorPoint` is known, and this is enough information for implementation-side typechecking of `display` to succeed.

We can also safely relax the signature ascription rules for the purposes of functor parameter matching, in order to make functors more widely applicable. For example, it is useful to allow the declaration of a class Cn to match a

```
RMod = block
 class R extends Rect of {}
end
RSig = sigblock
 class R extends Rect of {}
end
functor Translate(M:RSig) = block
 fun translateX:Rect → Rect
 extend fun translateX (RMod.R _) = ...
 extend fun translateX (M.R _) = ...
end
```

```
PointSig = sigblock
      class Point of {x:int,y:int}
end
functor Color(M:PointSig) = block
      class ColorPoint extends M.Point of {color:int}
      fun display:M.Point → unit
      extend fun display (M.Point _) = ...
      extend fun display (ColorPoint _) = ...
end
```

Figure 11: Difficulties with functors.

class specification whose `extends` clause refers to a reflexive, transitive superclass of *Cn*, rather than necessarily the direct superclass. This relaxation allows for more general mixin idioms than would otherwise be possible. It is also important to allow functions and function cases in a block to be hidden arbitrarily, so that the presence of an extra function or case does not render a block unable to be passed to a particular functor.

These relaxations can cause functions that are not available to a functor body to become incomplete, similar to the error illustrated in figure 10. To handle this problem, we augment the instantiation constraints which clients of the functor must meet. If the functor's parameter includes an abstract class *Cn*, any functions on *Cn* that do not have a global default case are implicitly considered part of the external interface of the functor. The functor can only be instantiated if the client can ensure that these functions will remain exhaustive. This condition is checked by performing exhaustiveness checks on the affected functions, from the blocks that would result from the functor instantiation. The functor instantiation is only legal if these checks succeed. Making such checking part of the instantiation constraints of a functor increases the burden on functor clients somewhat, while safely allowing functors to be more widely applicable. Further, functor instantiation checking remains purely modular (at the point of instantiation).

7 Related Work

OML's `objtype` is a form of hierarchical, extensible datatype [24]. As described in the introduction, OML includes both OO-style methods and ML-style functions; implementation-side typechecking for each of these constructs is straightforward. Unlike EML, OML retains ML-style type inference.

Zenger and Odersky [26] 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. 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 supporting dynamic dispatch and ML-style functions supporting pattern matching. Zenger's language also retains a distinction between datatype "cases" and regular OO classes.

$ML_{\leq}$ [2] provides hierarchical, extensible datatypes and simulates methods with function cases that use multimethod dispatching semantics, as in EML. Because $ML_{\leq}$'s extensible datatypes and functions are unrestricted, implementation-side typechecking must be performed at link-time. $ML_{\leq}$ also does not support arbitrary ML-style patterns or an ML-style module system. $ML_{\leq}$'s type system includes polymorphic subtype constraints on methods, while EML contains only ML-style parametric polymorphism.

EML's implementation-side typechecking restrictions extend our previous work on Dubious [16, 18], a multimethod-based OO calculus supporting modular typechecking. Dubious' methods do not support ML-style patterns. A Dubious module is essentially a block; structures, signature ascription, and functors are not considered. Finally, the type system of Dubious is monomorphic. EML's exhaustiveness and ambiguity algorithms are similar to the "mountain top" algorithms of Chambers and Leavens [6]. However, those algorithms do not handle ML-style patterns and must be performed globally. Previous work on unified dispatching [8] provides a global implementation-side typechecking algorithm for patterns that are more general than those in EML, including conjunctions, disjunctions, and negations of arbitrary predicates.

8 Conclusions and Future Work

We described EML, 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 in large part a long-standing tension between the functional and object-oriented styles. At the same time, EML retains completely modular typechecking of function implementations, in contrast with previous approaches, which require link-time checks. EML supports ML-style patterns and an ML-style module system.

There are several directions for future work. EML currently lacks particular support for constructing objects and establishing their invariants. Specialized constructor functions may be useful, along with a facility for invoking superclass constructor functions, so that initialization of inherited instance variables need not be duplicated by subclasses. More flexible forms of signature ascription are necessary, including the ability to hide some of a class's instance variables. Hiding instance variables is currently disallowed to prevent subclasses from defining a new instance variable of the same name as an existing one, but a representation for instance variables other than records could solve this problem. We plan to incorporate a more expressive polymorphic type system. We are currently implementing an interpreter for the language. Finally, we plan to explore forms of local type inference [22] for EML, to ease the type-annotation burden.

References

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

[2] François Bourdoncle and Stephan 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 January 1997.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

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

[5] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann 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.

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

[7] William 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.

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

[9] Robert Bruce Findler and Matthew 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.

[10] Matthew Flatt, Shriram Krishnamurthi, and Matthias 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.

[11] You-Chin C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, June 1990.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[13] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 176–185, New York, NY, January 1995. ACM.

[14] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing ojbect-oriented and functional design to promote re-use. In Eric 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.

[15] David MacQueen. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207. ACM, ACM, August 1984.

[16] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In Rachid 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.

[17] Todd Millstein and Craig Chambers. Modular type-checking for hierarchically extensible datatypes and functions. Technical Report UW-CSE-01-07-02, Department of Computer Science and Engineering, University of Washington, July 2001, revised October 2001. Available at `ftp://ftp.cs.washington.edu/tr/2001/07/UW-CSE-01-07-02.PS.Z`.

[18] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, to appear.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[20] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 216–227, 1999.

[21] Martin Odersky and Philip 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 January 1997.

[22] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

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

[24] John Reppy and Jon 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.

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

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