Modular Typechecking for Hierarchically Extensible Datatypes and Functions

TODD MILLSTEIN
University of California, Los Angeles
and
COLIN BLECKNER and CRAIG CHAMBERS
University of Washington

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

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Extensible datatypes, extensible functions, modular typechecking

---

An earlier version of this article appears in the Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02), 37:9.
This work was performed while the first author was at the University of Washington. 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.
Authors' addresses: T. Millstein, Computer Science Department, 4531D Boelter Hall, University of California, Los Angeles, Los Angeles, CA 90095-1596; email: todd@cs.ucla.edu; C. Bleckner and C. Chambers, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350; email: {colin,chambers}@cs.washington.edu.

1. INTRODUCTION

Many researchers have noted a difference in the extensibility benefits offered by the functional and object-oriented (OO) styles [Reynolds 1978; Cook 1991; Odersky and Wadler 1997; Krishnamurthi et al. 1998; Findler and Flatt 1998; Garrigue 2000; Zenger and Odersky 2001]. 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 [Rémy and Vouillon 1998] adds OO features including class and method definitions to ML. The OO constructs essentially form their own sublanguage that 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 [Reppy and Riecke 1996], 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 `objtypes` without modifying existing code, while ordinary ML functions may be. Methods dynamically dispatch on their associated `objtype`, while functions support ML-style pattern matching.

$ML_{\leq}$ [Bourdoncle and Merz 1997] 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 dispatch 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 subcomponents 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 retypechecking) of code.

The checks that must be delayed to link time in $ML_{\leq}$ constitute *implementation-side typechecking* (ITC) [Chambers and Leavens 1995], which ensures that each function in the program is exhaustively and unambiguously implemented. 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 in $ML_{\leq}$ is standard and can be performed modularly.

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 (noninherited) 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[1] (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

[1]Not to be confused with *Extended* ML [Kahrs et al. 1997].

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 expressible by either traditional functional or traditional 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 is inspired by the properties of a method's receiver that achieve modular typechecking in traditional OO languages. 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 [Millstein and Chambers 1999, 2002], 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 challenges for the interaction of EML's constructs with an ML-style module system, including signature ascription and functors. Section 6 discusses related work, and section 7 concludes. The appendix contains the type soundness proof for MINI-EML.

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 EML's new 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. 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)

```
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

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

Fig. 1. A hierarchy of integer sets in EML.

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 [Rémy and Vouillon 1998] and XMOC [Fisher and Reppy 2000], a core language for Moby [Fisher and Reppy 1999]. 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 are as expressive as ordinary ML-style datatypes. An ML datatype of the form

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

is encoded in EML by the following class declarations:

$$\texttt{abstract class DT of \{\}}$$
$$\texttt{class C}_1\texttt{(I}_{11}\texttt{:T}_{11},\ldots,\texttt{I}_{1m}\texttt{:T}_{1m}\texttt{) extends DT() of }\{\texttt{L}_{11}\texttt{:T}_{11}\texttt{=I}_{11},\ldots,\texttt{L}_{1m}\texttt{:T}_{1m}\texttt{=I}_{1m}\}$$

```
...
class C_r(I_r1:T_r1,...,I_rn:T_rn) extends DT() of {L_r1:T_r1=I_r1,...,L_rn:T_rn=I_rn}
```

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}
```

Classes additionally generalize ML-style datatypes to be *extensible*, whereby new variants can be written in modules other than the one declaring the datatype, and *hierarchical*, whereby variants can have their own "subvariants." The `CListSet` subclass of `ListSet` in Figure 1 illustrates class hierarchies; datatype extensibility is described in Section 2.4 below. In addition to being extensible and hierarchical, classes are also full-fledged types while ML variants are not. For example, classes can appear in a function's argument or return type.

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; `ref`s can be used in a class's representation type for this purpose.

Classes support only single inheritance. Single inheritance of classes is compatible with the ML style, in which each data variant conceptually singly inherits from the corresponding datatype, as shown in the above encoding of datatypes into classes. However, EML can support multiple *interface* inheritance, like Java [Arnold et al. 2000; Gosling et al. 2000]. Interfaces in EML have a syntax similar to classes, but they do not have instance variables and do not declare a constructor. For example, the abstract `Set` class in Figure 1 could instead be declared to be an interface as follows:

```
interface Set
```

Classes may be declared to `implement` interfaces. Given the above declaration, the declaration of `ListSet` would be modified to look as follows:

```
class ListSet(es:int list) implements Set of {es:int list = elems}
```

Like abstract classes, interfaces may not be instantiated. Unlike (abstract or concrete) classes, an interface can inherit from multiple interfaces via its `extends` clause, and a class can inherit from multiple interfaces via its `implements` clause. As described below, interfaces must obey additional constraints in exchange for this extra expressiveness.

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 `add` function in Figure 1, for example, is declared to accept one integer and one instance of `Set` or a subclass, and to return an instance of `Set` or a subclass. The `#` in the function's argument type signifies that the second argument to `add` is in the *owner position*. Every EML function declaration must specify exactly one owner position, and the type at that position must be a class or interface.

The owner position can be arbitrarily nested within a function's argument type. As a syntactic sugar, if a function's entire argument type is simply a class or interface, then that sole argument position is assumed to be the owner. This sugar is used in the declarations of the other functions in Figure 1. 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 cases would be declared as `size` methods in the `ListSet` and `CListSet` class declarations.

The `extend fun` declaration updates the set of cases associated with the specified function, instead of creating a new function containing the extra case. The chosen semantics is necessary for extensible functions to faithfully model OO-style methods. For example, suppose a new structure `HashSetMod` creates a `Set` subclass `HashSet`, which represents sets using hash tables, along with an overriding `size` function case for `HashSet`. (Such a structure is described in Section 2.4 below.) The semantics of `extend fun` allows `HashSet`'s `size` case to be invoked from the existing `size` application in the first `isEmpty` case in Figure 1, even though `HashSet` and its `size` case are not known in `SetMod`. Similarly, clients of the set hierarchy need not be aware of all `Set` subclasses and need not be modified as new `Set` subclasses are introduced.

A regular 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 allow the expression of both OO-style dynamic dispatch and functional-style patterns. OO dispatch is encoded by a *class pattern*. For example, the second `size` case is only applicable dynamically if the argument is an instance of `CListSet` or a subclass. Class patterns contain a *representation pattern*, which supports dispatch on instance variables recursively. While the second `size` case's representation pattern matches the representation of any instance of `CListSet` (or a subclass), the second `isEmpty` case requires the argument's `count` instance variable to have the value zero. As in ML, a pattern may bind identifiers for use in its case's body. For example, the first `add` case binds the identifiers `i`, `s`, and `es`.

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 `isEmpty(CListSet([],0))`. Both `isEmpty` 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.

Implementation-side typechecking of a function ensures that *match nonexhaustive* and *match ambiguous* errors cannot occur at run time. Conceptually this checking entails ensuring that each type-correct argument to the function has a most-specific applicable case to invoke. 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 four 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 statically signaled. Alternatively, if another `size` case with pattern `ListSet {es=es}` were declared, a *match ambiguous* error would be statically signaled.

Unlike (both concrete and abstract) classes, interfaces may not appear in patterns. This restriction is the EML analogue of Java's restriction that an interface have no concrete methods. Both restrictions remove the potential for dynamic-dispatch ambiguities caused by multiple inheritance. Because of EML's restriction, interfaces do not impact ITC any differently from abstract classes. Therefore we ignore interfaces in the remainder of the paper.

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 `Sets`, without modifying any code in the `SetMod` module.[2] Two `union` function cases are provided. The first case is applicable to any pair of `Sets`. The second `union` case provides a more efficient implementation for two `ListSets`. ITC of `UnionMod` checks `union` for exhaustiveness and unambiguity. Any pair consisting of `ListSet` and/or `CListSet` instances will invoke the second `union` case, so the function's check succeeds.

[2]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 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
```

Fig. 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
```

Fig. 3. Adding new data variants in EML.

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, while the first `isEmpty` case in Figure 1 is inherited. In a traditional OO language, `HashSetMod` corresponds to the declaration of a new subclass of `Set` with some overriding methods. ITC of `HashSetMod` rechecks the four functions declared in Figure 1 to ensure that they handle `HashSet` instances properly. For example, if the new `size` case were not declared, a *match nonexhaustive* error for `size` would be signaled statically.

`UnionMod` and `HashSetMod` from Figures 2 and 3 illustrate EML's support for both functional and OO forms of extensibility in a single class hierarchy. The original `Set` abstraction is flexibly reused by clients, who augment the abstraction with client-specific functionality and also add a specialized implementation (subclass) of the abstraction, all without modifying existing code. `UnionMod` and `HashSetMod` 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 each 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

```
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
```

Fig. 4. Class hierarchies in EML.

```
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)
```

Fig. 5. Polymorphic sets in EML.

typechecks. Therefore, neither module's ITC ensures that `union` is exhaustively and unambiguously implemented for `HashSets`. In this example, `union` happens to have a case that handles `HashSets` (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. 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 the four functions declared in Figure 1 as well as the new `getMin` function to ensure exhaustiveness and unambiguity for `SListSets`.

2.5 Parametric Polymorphism

EML supports a polymorphic type system. Class, function, and function case declarations optionally bind *type variables*. References 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. As a convenience, 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 unique typesafe instantiation for such classes is determined 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. First, subclasses must 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. Also, 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* [Cardelli and Wegner 1985]. 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, EML's polymorphic type system could be generalized in standard ways without affecting modular typechecking. For example, EML could adopt $\mathrm{ML}_{\leq}$'s subtype-constrained polymorphic type system for extensible datatypes and functions [Bourdoncle and Merz 1997].

EML is also explicitly typed. This contrasts with ML's polymorphic type system, which supports type inference. Unfortunately, supporting both subtyping and polymorphic type inference is known to be difficult [Fuh and Mishra 1990; Hoang and Mitchell 1995; Nordlander 1999]. It would be useful to explore forms of *local* type inference [Pierce and Turner 2000] to ease the type-annotation burden somewhat. Recent work of Bonniot Bonniot [2002] has presented a simplified account of $\mathrm{ML}_{\leq}$'s type system and shown how to incorporate a form of local type inference, so this could be a promising foundation for augmenting EML.

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 straightforward 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 *signatures*[3] of other modules (without requiring access to the associated implementations). Second, *m* can be typechecked given only those signatures that *m statically depends upon*. Module *m* statically depends upon signature *s* if either of the

[3]We employ the ML terminology *signature*, rather than the more generic *interface*, to avoid confusion with the notion of `interface` in languages like Java and EML.

following conditions holds:

- Module m refers to a name that is bound in s.
- Module m statically depends upon module signature s', and s' refers to a name that is bound in s.

Traditional functional languages can support modular typechecking. For example, each structure in ML could be typechecked given only its statically depended-upon signatures. A structure's signature is either an explicitly ascribed one 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 signatures. Informally, the signature of a class consists of its list of superclasses and superinterfaces, the types of its visible fields, and the headers, but not bodies, of its visible methods. Signatures for modular typechecking in the context of Java have been formally defined by others [Drossopoulou et al. 1999; Ancona et al. 2002].

A modular typechecking scheme for EML must typecheck each structure given only the signatures it statically depends upon. An EML signature contains a sequence of class, function, and function-case *specifications*. The syntax of each specification is identical to that of its associated declaration, but with all expressions removed. In particular, each `class` specification does not include the arguments to the superclass constructor or the instance-variable initializer expressions, and each `extend fun` specification does not include the case's body.

For now, EML does not support explicit signature ascription; this ability is discussed in Section 5. Instead, each structure is implicitly ascribed to its principal signature. This signature has the same name as its associated structure and contains a specification for each declaration in the structure. Further, each specification is identical to its associated declaration, but with all expressions removed. For example, Figure 6 shows the principal signature of `SetMod` from Figure 1.

Classes, functions, and cases that are declared in a module m or specified in a signature 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 signature. 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`.

One undesirable feature of an EML signature is the fact that it reveals information about a function's cases. Ideally, clients of an EML structure could be safely typechecked given only type information about the classes and functions declared in a module, without requiring any knowledge of individual `extend fun` declarations. Such a revised EML signature would adhere to Cardelli's notion of modular typechecking, which requires each structure to be typechecked given only the types of its free variables [Cardelli 1997]. However,

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

  fun add:(int * #Set) → Set
  extend fun add (i, s as ListSet {es=es})
  extend fun add (i, s as CListSet {es=es,count=c})

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

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

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

Fig. 6. The principal signature of `SetMod`.

EML's expressiveness makes this stronger notion of modularity infeasible. A client of a structure may need access to the patterns in a structure's `extend fun` declarations in order to ensure that the associated functions remain exhaustive and unambiguous in the face of the client's declared classes and function cases. Importantly, clients need never have access to the *bodies* of those `extend fun` declarations.

EML is not alone in suffering from this problem. Indeed, EML's notion of modularity generalizes the notion of modularity in traditional OO languages, which also does not meet Cardelli's definition. As mentioned above, the signature for a class in such languages includes the headers of individual methods. Importantly, the signature for a class C reveals which of the operations that C supports are implemented by a method that dynamically dispatches on C at the receiver; methods for other operations are inherited from superclasses. This information about individual methods is necessary so that subclasses can ensure the exhaustiveness and unambiguity of all operations they support. For example, an abstract class's concrete method headers determine which operations must be implemented by concrete subclasses and which need not be. As another example, a class that inherits from multiple other classes requires knowledge of the method implementations in those classes in order to detect potential ambiguities. Revealing the patterns of function cases is the EML generalization of this revealing of individual methods' receivers in traditional OO languages.

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,

```
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
```

Fig. 7. Challenges for modular implementation-side typechecking.

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 7 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 signatures (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 signature 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 `Circle`s. The `intersect` case in `CircleMod` is most specific for two `Circle`s, so `intersect` is found to be exhaustive and unambiguous. By similar reasoning, `intersect` passes the checks from `RectMod`, since `RectMod` does not statically depend on the signature of `CircleMod`.

Therefore each module's naive modular ITC declares 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 signature, `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 modularly typesafe. To address this problem, we impose some additional requirements on EML modules that ensure the soundness of modular ITC. This section informally describes EML's modular ITC algorithms and requirements. ITC is divided into checks for exhaustiveness and checks for unambiguity; we discuss each in turn.

A fundamental design goal for EML's modular type system is that it 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 expressible with

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

extensible datatypes and functions in order 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 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 [Millstein and Chambers 2002]. The type at the owner position in a function's argument type 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.

3.3.1 *Modular Exhaustiveness Checking.* For the purposes of modular exhaustiveness checking, 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 signature" of its owner class, while an external function is a later extension to that signature. External functions have no analogue in traditional OO languages, in which a class's methods must all be declared with the class. Modular exhaustiveness checking consists in enforcing two requirements, one for external functions and the other for internal functions, which are discussed in turn.

Exhaustiveness Requirement for External Functions. Consider the exhaustiveness problem with the `print` function in `RectMod` of Figure 7. 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 exhaustive, 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 each 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 the pattern `(Shape {})`, for example, then the requirement would be satisfied and the exhaustiveness 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 ability to write an appropriate global default case depends heavily on the functionality available in the "initial signature" of a class. In particular, the global-default requirement only works well for external functions whose behavior can be expressed solely in terms of that signature. For example, the

requirement is very natural for `union` in Figure 2: the given global default is appropriate for all unseen `Set` subclasses. However, had `SetMod` in Figure 1 not included the `elems` function, it is unlikely that a reasonable default case for `union` could be written. Instead, the implementer may have no choice but to make the global default case simply throw an exception, which is not much different from the run-time *match nonexhaustive* error being prevented by the global-default requirement.

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 like ML 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. In contrast, EML's modular ITC must always allow for the possibility of unavailable subclasses of a function's owner. Section 5 introduces a mechanism for *sealing* class hierarchies, which can be used to obtain the semantics of ordinary (nonextensible) datatypes in EML. Analogous with ML, external functions on a sealed class hierarchy need not include a global default case.

Exhaustiveness Requirement for Internal Functions. Consider the exhaustiveness problem for a pair of one `Rect` and one `Circle` in the internal `intersect` function of Figure 7. 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 problem 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 exhaustiveness 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 that declares 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 is one whose pattern accepts 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 exhaustively implements all of the functions in its "initial" signature.

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.) A local default case is required regardless of what other `intersect` cases exist, like the one shown in `RectMod`. The

requirement would be satisfied, for example, if `RectMod` included 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 exhaustiveness problem 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 the two concrete `Set` subclasses. When `HashSet` is introduced in Figure 3, an associated local default for `add` is declared, satisfying the requirement and ensuring that `add` is exhaustive 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. In contrast, EML's ITC must always assume the possibility of unavailable concrete subclasses of classes in nonowner positions of a function's argument type. Again, we can use sealing, discussed in Section 5, to remove the burden of writing local default cases.

Precise Default Checking. One practical issue that arises with checking for (global and local) defaults in EML is the need to avoid being overly conservative. For example, a simple way to enforce the global-default requirement is to check that every external function has a case whose pattern is the wildcard pattern (`_`). While using this algorithm in EML would be safe, it would also cause many exhaustive functions to nonetheless be rejected. For example, the `union` function in Figure 2 has a global default case, but it fails the above check. To remedy this imprecision, EML instead checks that each external function has a case whose pattern is some *global default pattern*: any argument of the function's type matches the pattern. In the case of `union`, there are several possible global default patterns, including `_`, `(_, _)`, `(s1, s2)`, `(Set _, _)`, and `(Set {}, Set _)`.

EML performs the revised check by generating a precise global default pattern and then requiring that this pattern be at least as specific as some case's pattern; that case is a global default. The algorithm retains precision via the notion of a pattern's *depth*, which is essentially the height of the pattern's abstract syntax tree representation. A valid global default case will not be overlooked by the algorithm as long as the generated global default pattern has a depth no smaller than that of the case's pattern. Therefore, EML generates the pattern to have a depth equal to the maximum depth of any available case on the function.[5] In the `union` example, exhaustiveness checking generates the global default pattern `(Set {}, Set {})`, matching the depth of the second case's pattern. The check

[5]Because class patterns allow pattern matching on a class's representation, which may recursively involve more class patterns, there is in general no *a priori* maximal depth for the patterns of a given function.

succeeds because the generated pattern is at least as specific as the first case's pattern, so that case is a valid global default. A similar algorithm is used for checking for local default cases. These algorithms are implemented in the EML prototype interpreter and are formalized in Section 4.

3.3.2 *Modular Ambiguity Checking*

Ambiguity Requirement. In Figure 7 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 unambiguous. 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 1) be declared in the module that declares the case's function or 2) have an owner and be declared in the module that declares that owner.

`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. Removing that `intersect` case resolves the ambiguity for a pair of a `Circle` and a `Rect`. 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 case for a newly declared subclass of `add`'s owner. Therefore the two cases are guaranteed to be unambiguous.

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.[6] 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 a more efficient version of `union` for `HashSets`. However, the new case would violate our ambiguity requirement, so `HashSets`

[6]In the presence of multiple *implementation* inheritance, other kinds of ambiguities that elude modular detection can arise, necessitating an extra requirement [Millstein and Chambers 2002]. However, multiple *interface* inheritance, as in Java and EML, cannot cause such ambiguities.

are forced to use the default `union` case (or `HashSetMod` must be modified in place to add the new case).

Pairwise Ambiguity Checking. The above ambiguity requirement guarantees that each case declared in a structure S is not ambiguous with cases that are *unavailable* from S. To complete modular ambiguity checking, each case c declared in S is checked for ambiguity with each *available* case c' other than itself. Let the patterns of c and c' be Pat and Pat', respectively. Cases c and c' are checked for ambiguity as follows:

- If Pat and Pat' are *congruent*, meaning that they are identical when all identifier bindings are removed, then the cases are ambiguous.
- Else if the patterns are *disjoint*, meaning that no value can match both Pat and Pat', then the cases are unambiguous.
- Else there is some pattern Pat'' that represents the *intersection* of Pat and Pat': all values matching both Pat and Pat' also match Pat''. The cases are unambiguous only if there exists a case c'' whose pattern is congruent to Pat''. We call case c'' the *resolving* case, because it resolves the ambiguity between c and c'.

A degenerate form of the third scenario above occurs when one of Pat and Pat' is strictly more specific than the other, so the resolving case is one of the original two. For example, consider checking `size` for ambiguity in `SetMod` of Figure 1. The two `size` cases in Figure 1 are neither congruent nor disjoint. The intersection pattern is congruent to the second case's pattern, and that case itself is the resolving case.

3.4 Discussion

EML supports the extensibility idioms of traditional functional and OO languages: both new subclasses and new operations can be added to existing classes without modifying existing code, resolving a longstanding tension between these two forms of extensibility. At the same time, EML retains completely modular typechecking by imposing the modularity requirements described above.

The requirements rely heavily on the notion of a function's owner position. Although the owner position of a function f does not affect f's invocation semantics, it has a large impact on the ways in which clients can interact with and extend the function. Through the local-default requirement, the owner position determines the default cases that clients must implement. Through the ambiguity requirement, the owner position determines what cases clients are forbidden from implementing. Therefore, effective use of f by clients requires some advance planning by the original implementer of f, who must choose an appropriate owner position based on the kinds of extensibility that are anticipated. This advance planning is analogous to the need to choose which argument of a method should be the receiver in traditional OO languages. That choice has the same effects as the choice of the owner position in EML and additionally determines the sole argument that may be dynamically dispatched upon.

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

Fig. 8. MINI-EML types, expressions, and patterns.

Intuitively, a class C in f's argument type is a good choice as the function's owner if the following conditions hold:

- It is expected that f's behavior will depend on the particular subclass of C that is passed as an argument.
- It is expected that clients will declare new subclasses of C.

The first condition implies that methods will likely need to dispatch at C's position. The second condition then implies that clients will likely need to declare overriding methods of f for new subclasses of C. Making C the owner allows these overriding methods to satisfy the ambiguity requirement. The first condition also implies that it may be difficult to implement f for C itself, if C is abstract. Making C the owner will obviate the need for such a default implementation, instead requiring concrete subclasses of C to provide their own local defaults.

The EML exhaustiveness and ambiguity requirements together enforce an important *monotonicity* property on EML programs: the local view of a program from one structure is always consistent with the global view in a particular sense. Specifically, if the view from one structure suggests that some available type-correct argument a to an available function f will invoke the available case c, then that will be true at run time, no matter what other structures are part of the complete program. This property ensures that ITC can be safely performed piecewise on a function, from each structure's partial view of the program. It also validates a programmer's understanding given only a partial view of the program, ensuring that a type-correct argument cannot be "hijacked" by an unseen function case.

4. MINI-EML

This section describes MINI-EML, a core language that formalizes EML and its modular type system.

4.1 Syntax

Figure 8 defines the syntax of types, expressions, and patterns in MINI-EML. The syntax is essentially that of EML as informally presented so far, but some standard constructs are omitted, including base types, conditionals, anonymous and lexically nested functions, local variables, references, and exceptions. Metavariable α ranges over type variable names, I over identifier names, Sn over structure names, Cn over class names, Vn over instance variable names, and Fn over

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

(a)

$$
\begin{array}{lcl}
Sig & ::= & \texttt{signature}\ Sn = \texttt{sig}\ \overline{Sp}\ \texttt{end} \\
Sp & ::= & \texttt{<abstract> class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau}) \\
 & & \quad \texttt{<<extends}\ Ct\ \texttt{>> of}\ \{\overline{Vn} : \overline{\tau_0}\} \\
 & \mid & \texttt{fun}\ \overline{\alpha}\ Fn : Mt \rightarrow \tau \\
 & \mid & \texttt{extend}\ \texttt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat
\end{array}
$$

(b)

Fig. 9. (a) MINI-EML structures and declarations. (b) MINI-EML signatures and specifications.

function names. $\overline{X}$ denotes a comma-separated sequence of elements of the domain X (and is independent of any variable named X); the empty sequence is denoted $\bullet$. 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 $\overline{V} = \overline{Pat}$.

MINI-EML types include type variables, class types, function types, and tuple types. The domain Mt represents *marked types*, which contain a # mark on a single component class type. Expressions include identifiers, function values, function application, constructor calls, tuples, and *instance expressions*. 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. Scoping allows a class to introduce an instance variable with the same name as one in a superclass declared in another module. While this ability provides only a minor convenience in MINI-EML, scoping provides a foundation for allowing a class to safely hide instance variables, as described in Section 5. 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.

Patterns include the wildcard pattern, identifier binding, class patterns, and tuple patterns. For simplicity, the representation pattern $\{\overline{V} = \overline{Pat}\}$ within a class pattern must mention all of the associated class's instance variables. A pattern of the form I, used in our earlier examples, is syntactic sugar for $(I\ \texttt{as}\ \_)$.

The notation and semantic style of MINI-EML were influenced by Featherweight Java [Igarashi et al. 2001], a core language for Java. As in that language, classes are formally represented 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. 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 9a. A structure consists of a sequence of class, extensible function, and function case declarations. The syntax of the three declarations is faithful to that of EML, except that cases now contain a *case name*, ranged over by metavariable Mn. This name is used in the semantics to uniquely identify each function case

declaration. Angle brackets (<>) and double angle brackets (<<>>) denote independent optional pieces of syntax. The notation $\overline{Vn} : \overline{\tau} = \overline{E}$ abbreviates $Vn_1 : \tau_1 = E_1, \ldots, Vn_k : \tau_k = E_k$, and similarly for $\overline{I} : \overline{\tau}$. The declared inheritance graph is assumed to be acyclic. The class, function, and case names introduced in a given structure are assumed to be distinct. The type variables parameterizing a given declaration are assumed to be distinct. All the instance variable names introduced in a given structure are assumed to be distinct. The identifiers introduced in a given function case's pattern are assumed to be distinct.

MINI-EML also includes an explicit notion of signature, as defined in Figure 9b. In ML, the name of a signature is completely independent of the names of structures that conform to the signature. However, there is a one-to-one mapping between structures and signatures in a MINI-EML program (see below), so for simplicity the name of a MINI-EML structure is also used as the name of its associated signature. A signature contains a sequence of class, function, and function-case specifications. The syntax for each specification is identical to its corresponding declaration, but with all expressions removed. The notation $\overline{Vn} : \overline{\tau}$ abbreviates $Vn_1 : \tau_1, \ldots, Vn_k : \tau_k$.

Inspired by Featherweight Java, a MINI-EML program is a pair of a *structure table* and an expression to be evaluated. A structure table is a finite function from structure names to the associated structure declarations. The formal semantics assumes a fixed structure table ST. The domain of ST is denoted $\mathrm{dom}(ST)$. The structure table is assumed to satisfy some sanity conditions: (1) $ST(Sn) = (\texttt{structure}\ Sn\ = \texttt{struct} \cdots \texttt{end})$ for every $Sn \in \mathrm{dom}(ST)$; (2) for every structure name Sn appearing anywhere in the program, $Sn \in \mathrm{dom}(ST)$.

The static semantics also relies on a fixed *signature table* $SigT$, which maps each structure name Sn in $\mathrm{dom}(ST)$ to the *principal signature* of $ST(Sn)$. $SigT$ is easily computed from ST: the principal signature of a structure S is simply the signature of the same name as S whose body is identical to S's body, but with all expressions removed. Each structure in the range of ST is typechecked in the context of $SigT$, without access to ST, thereby ensuring that the first criterion for modular typechecking is met, as defined in Section 3.1: each structure is typechecked only against the signatures, rather than the implementations, of other structures. The domain of $SigT$ is denoted $\mathrm{dom}(SigT)$.

4.2 Dynamic Semantics

MINI-EML's dynamic semantics is defined as a mostly standard small-step operational semantics. The structure table is accessed whenever information about a declaration is required in order to execute an expression. The metavariable ρ ranges over *environments*, which are finite functions from identifiers to values. $|\overline{X}|$ denotes the length of the sequence $\overline{X}$. The notation $[I_1 \mapsto E_1, \ldots, I_k \mapsto E_k]X$ denotes the expression resulting from the simultaneous substitution of E_i for each occurrence of I_i in X, for $1 \leq i \leq k$, and similarly for $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k]X$. $[\overline{I} \mapsto \overline{E}]X$ is used as a shorthand when $\overline{I}$ and $\overline{E}$ have the same length, and similarly for $[\overline{\alpha} \mapsto \overline{\tau}]X$. In a given inference rule, fragments enclosed in <> must either be all present or all absent, and similarly

$\boxed{E \longrightarrow E'}$

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

$$\frac{E \longrightarrow E'}{Ct\{\overline{V_0} = \overline{v_0}, V = E, \overline{V_1} = \overline{E_1}\} \longrightarrow Ct\{\overline{V_0} = \overline{v_0}, V = E', \overline{V_1} = \overline{E_1}\}}\ \text{E-REP}$$

$$\frac{E \longrightarrow E'}{(\overline{v_0}, E, \overline{E_1}) \longrightarrow (\overline{v_0}, E', \overline{E_1})}\ \text{E-TUP}$$

$$\frac{E_1 \longrightarrow E_1'}{E_1\ E_2 \longrightarrow E_1'\ E_2}\ \text{E-APP1} \qquad \frac{E_2 \longrightarrow E_2'}{v_1\ E_2 \longrightarrow v_1\ E_2'}\ \text{E-APP2}$$

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

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

$$\frac{(\texttt{class}\ \overline{\alpha}\ Cn \ldots) \in ST(Sn)}{\text{concrete}(Sn.Cn)}\ \text{CONCRETE}$$

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

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

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

$$\frac{\begin{array}{c}(\texttt{extend}\ \ \texttt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E) \in ST(Sn) \qquad \text{match}(v, Pat) = \rho \\ \forall Sn' \in \text{dom}(ST).\forall(\texttt{extend}\ \ \texttt{fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat' \ldots) \in ST(Sn').\forall \rho'. \\ ((\text{match}(v, Pat') = \rho' \wedge Sn.Mn \neq Sn'.Mn') \Rightarrow Pat < Pat')\end{array}}{\text{most-specific-case-for}\ ((\overline{\tau}\ F), v) = (\rho, [\overline{\alpha} \mapsto \overline{\tau}]E)}\ \text{LOOKUP}$$

Fig. 10. Evaluation rules for MINI-EML expressions.

for <<>>. Sequences are sometimes treated as if they were sets. For example, $D \in \overline{D}$ means that D is one of the declarations in $\overline{D}$. Finally, $D \in ST(Sn)$ is shorthand for the two facts $ST(Sn) = (\texttt{structure}\ Sn\ = \texttt{struct}\ \overline{D}\ \texttt{end})$ and $D \in \overline{D}$, and similarly for $Sp \in SigT(Sn)$.

Figure 10 contains the rules for evaluating expressions. The notation $(\overline{I}, \overline{v})$ abbreviates $(I_1, v_1), \ldots, (I_k, v_k)$, and $Sn.\overline{Vn} = \overline{E}$ abbreviates $Sn.Vn_1 = E_1, \ldots, Sn.Vn_k = E_k$. For simplicity in the semantics, a constructor call is treated as syntactic sugar for the instance expression obtained by expanding the constructor's definition. Rule E-NEW specifies this semantics, and Rule E-REP evaluates instance expressions. It would be straightforward to instead use a call-by-value semantics for constructor calls, at the cost of some additional mechanism. Rule E-NEW uses the first two auxiliary rules at the bottom of the figure. Rule CONCRETE checks that the class to be instantiated was declared without the `abstract` keyword. Rule REP initializes the fields of the new instance as directed by the class's constructor, substituting the actual arguments to the

constructor call for the formals. The rule also substitutes the new instance's type parameters for the class's type variables. Types have no dynamic effect in MINI-EML, but maintaining types in the dynamic semantics eases the type system's proof of soundness.

The last rule in Figure 10 formalizes function-case lookup, used in E-APPRED. The first premise of LOOKUP specifies the case to invoke. The second premise ensures that this case is applicable: the argument value matches the case's pattern. That premise also produces an environment mapping each identifier in the case's pattern to the appropriate "pieces" of the argument value. This environment is used by E-APPRED to evaluate the case's body. 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 11. The notation $\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$ abbreviates $\text{match}(v_1, Pat_1) = \rho_1 \cdots \text{match}(v_k, Pat_k) = \rho_k$, and similarly for $\overline{Pat_1} \leq \overline{Pat_2}$. The matching rules are straightforward except for E-MATCHCLASS. The judgment $C \leq C'$ is defined at the bottom of Figure 11 as the reflexive, transitive closure of the declared class `extends` relation. Therefore, an instance of class C matches a class pattern of class C' if C subclasses C' and the instance's representation recursively matches the given representation pattern. The instance may have more instance variables than are mentioned in the given representation pattern, so that subclass instances can match superclass patterns.

The judgment $Pat \leq Pat'$ means that Pat is at least as specific as Pat'. Rule LOOKUP uses $Pat < Pat'$ as shorthand for the two facts $Pat \leq Pat'$ and $Pat' \not\leq Pat$. The pattern specificity semantics generalizes OO-style best-match semantics to support ML-style patterns. Any pattern is at least as specific as the wildcard, and identifier binding has no effect on specificity. 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 specificity of tuple patterns is analogous to the symmetric dispatching semantics of multimethod-based OO languages like Dubious [Millstein and Chambers 2002]. 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.

4.3 Static Semantics

Figure 12 contains the rules for typechecking structures and declarations. Γ is a *type environment*, mapping identifiers to types. The notation $\hat{Mt}$ denotes the type τ identical to Mt, but with the # mark removed. The notation $\overline{Sn} \vdash \overline{D}$ OK in Sn abbreviates $\overline{Sn} \vdash D_1$ OK in $Sn \cdots \overline{Sn} \vdash D_k$ 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 $\tau_{11} \leq \tau_{01} \cdots \tau_{1k} \leq \tau_{0k}$.

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

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

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

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

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

$\boxed{Pat \leq Pat'}$

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

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

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

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

$\boxed{C \leq C'}$

$$\frac{}{C \leq C}\ \textsc{SubRef}$$

$$\frac{C_1 \leq C_2 \qquad C_2 \leq C_3}{C_1 \leq C_3}\ \textsc{SubTrans}$$

$$\frac{(\texttt{<abstract>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I_1} : \overline{\tau_1})\ \texttt{extends}\ \overline{\tau}\ C \ldots) \in ST(Sn)}{Sn.Cn \leq C}\ \textsc{SubExt}$$

Fig. 11. EML pattern matching, pattern specificity, and subclassing.

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. $\overline{Sn}$ in StructOK denotes those signatures in $SigT$ that may be accessed during ITC on the current structure. As defined in Section 3.1, ITC is only modular if the current structure statically depends upon each signature in $\overline{Sn}$. Later judgments ensure the well-formedness of the chosen $\overline{Sn}$, as described below.[7]

[7]This "guess and check" style is used for simplicity. An alternative would be to take an initial pass over each structure's declarations in the static semantics, in order to compute the appropriate $\overline{Sn}$ before typechecking the structure.

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

$$\frac{\overline{Sn} \subseteq \mathrm{dom}(SigT) \qquad \overline{Sn} \vdash \overline{D} \text{ OK in } Sn}{\texttt{structure}\ Sn = \texttt{struct}\ \overline{D}\ \texttt{end OK}}\ \textsc{StructOK}$$

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

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

$$\frac{\overline{\alpha} \vdash \hat{M}t \text{ OK} \qquad \overline{\alpha} \vdash \tau \text{ OK} \qquad \text{owner}(Sn.Fn) = Sn'.Cn \qquad Sn \neq Sn' \Rightarrow \overline{Sn} \vdash Sn.Fn \text{ has-gdefault}}{\overline{Sn} \vdash \texttt{fun}\ \overline{\alpha}\ Fn : Mt \rightarrow \tau \text{ OK in } Sn}\ \textsc{FunOK}$$

$$\frac{\begin{array}{c} (\texttt{fun}\ \overline{\alpha'}\ Fn : Mt \rightarrow \tau) \in SigT(Sn') \qquad \text{matchType}([\overline{\alpha'} \mapsto \overline{\alpha}]\hat{M}t, Pat) = (\Gamma, \tau_0) \\ \Gamma; \overline{\alpha} \vdash E : \tau' \qquad \tau' \leq [\overline{\alpha'} \mapsto \overline{\alpha}]\tau \qquad \overline{Sn} \vdash Sn'.Fn \text{ ITCUses} \qquad Sn; \overline{Sn} \vdash \texttt{extend}\ \texttt{fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat \text{ unambiguous} \end{array}}{\overline{Sn} \vdash \texttt{extend}\ \texttt{fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat = E \text{ OK in } Sn}\ \textsc{CaseOK}$$

Fig. 12. Static semantics of MINI-EML structures and declarations.

The formalism does not explicitly enforce modularity of the rest of static typechecking, as those checks are standard and are naturally modular.

The rules for typechecking the three declaration forms are largely straightforward. Rule CLASSOK checks that a class's superclass constructor call is well-typed, that all types mentioned in the class declaration are well-formed, and that the instance-variable initializer expressions have the appropriate types. The first premise in the rule ensures that the new class has the same type variables as its superclass, as mentioned in Section 2.5. Rule FUNOK checks that a function's declared type is well-formed. Rule CASEOK ensures that the case's pattern and body are compatible with the associated function's declared type. The "ITCTransUses" and "ITCUses" judgments in CLASSOK and FUNOK ensure well-formedness of the signatures $\overline{Sn}$ to be accessed during ITC of the enclosing structure; these judgments are described below. Finally, each rule enforces one of the three requirements for modular ITC: CLASSOK enforces the local-default requirement ("funs-have-ldefault-for") if the class is concrete; FUNOK enforces the global-default requirement ("has-gdefault") if the function is external; CASEOK performs ambiguity checking ("unambiguous") for the given case, which includes enforcement of the ambiguity requirement as described in Section 3. The judgment for each requirement has $\overline{Sn}$ in the context, to ensure that only signatures in this sequence are accessed during enforcement of the requirement.

Figure 13 contains the static semantics of types. The judgment $\overline{\alpha} \vdash \tau$ OK ensures that τ refers only to type variables in $\overline{\alpha}$ and that each class in τ has the correct number of type parameters. The subtyping relation $\tau \leq \tau'$ is completely standard [Cardelli 1988].

Figure 14 contains the rules for typechecking expressions. The notation $Sn.\overline{Vn} : \overline{\tau}$ abbreviates $Sn.Vn_1 : \tau_1, \ldots, Sn.Vn_k : \tau_k$. The judgment $\Gamma; \overline{\alpha} \vdash E : \tau$ ensures that an expression is well-typed in the context of the type environment and sequence of type variables currently in scope. Most of the rules are standard. Rule T-FUN looks up a function's declared type in the signature table and substitutes the given use's type parameters for the function's type variables. Rule T-NEW uses T-CONSTR to check that a constructor invocation includes a

$\boxed{\overline{\alpha} \vdash \tau \text{ OK}}$

$$\frac{\alpha \in \overline{\alpha}}{\overline{\alpha} \vdash \alpha \text{ OK}} \text{ TVarOK}$$

$$\frac{\begin{array}{c}\overline{\alpha} \vdash \overline{\tau} \text{ OK} \\ (\texttt{<abstract>}\ \texttt{class}\ \overline{\alpha_0}\ Cn \ldots) \in SigT(Sn) \\ |\overline{\alpha_0}| = |\overline{\tau}|\end{array}}{\overline{\alpha} \vdash \overline{\tau}\ Sn.Cn \text{ OK}} \text{ ClassTypeOK}$$

$$\frac{\overline{\alpha} \vdash \tau_1 \text{ OK} \qquad \overline{\alpha} \vdash \tau_2 \text{ OK}}{\overline{\alpha} \vdash \tau_1 \to \tau_2 \text{ OK}} \text{ FunTypeOK}$$

$$\frac{\overline{\alpha} \vdash \tau_1 \text{ OK} \cdots \overline{\alpha} \vdash \tau_k \text{ OK}}{\overline{\alpha} \vdash \tau_1 * \cdots * \tau_k \text{ OK}} \text{ TupTypeOK}$$

$\boxed{\tau \leq \tau'}$

$$\frac{}{\tau \leq \tau} \text{ SubTRef} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \text{ SubTTrans}$$

$$\frac{\begin{array}{c}(\texttt{<abstract>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I_1} : \overline{\tau_1}) \\ \texttt{extends}\ Ct \ldots) \in SigT(Sn)\end{array}}{\overline{\tau}\ Sn.Cn \leq [\overline{\alpha} \mapsto \overline{\tau}]Ct} \text{ SubTExt}$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \text{ SubTFun}$$

$$\frac{\tau_1 \leq \tau_1' \ \cdots \ \tau_k \leq \tau_k'}{\tau_1 * \cdots * \tau_k \leq \tau_1' * \cdots * \tau_k'} \text{ SubTTup}$$

Fig. 13. Static semantics of Mini-Eml types.

$\boxed{\Gamma; \overline{\alpha} \vdash E : \tau}$

$$\frac{(I, \tau) \in \Gamma}{\Gamma; \overline{\alpha} \vdash I : \tau} \text{ T-Id}$$

$$\frac{(\texttt{fun}\ \overline{\alpha_0}\ Fn : Mt \to \tau) \in SigT(Sn) \qquad \overline{\alpha} \vdash \overline{\tau_0} \text{ OK}}{\Gamma; \overline{\alpha} \vdash \overline{\tau_0}\ Sn.Fn : [\overline{\alpha_0} \mapsto \overline{\tau_0}](\hat{M}t \to \tau)} \text{ T-Fun}$$

$$\frac{\Gamma; \overline{\alpha} \vdash E_1 : \tau_2 \to \tau \qquad \Gamma; \overline{\alpha} \vdash E_2 : \tau_2' \qquad \tau_2' \leq \tau_2}{\Gamma; \overline{\alpha} \vdash E_1\ E_2 : \tau} \text{ T-App}$$

$$\frac{\Gamma; \overline{\alpha} \vdash Ct(\overline{E}) \text{ OK} \qquad Ct = (\overline{\tau}\ C) \qquad \text{concrete}(C)}{\Gamma; \overline{\alpha} \vdash Ct(\overline{E}) : Ct} \text{ T-New}$$

$$\frac{\Gamma; \overline{\alpha} \vdash E_1 : \tau_1 \ \cdots \ \Gamma; \overline{\alpha} \vdash E_k : \tau_k}{\Gamma; \overline{\alpha} \vdash (E_1, \ldots, E_k) : \tau_1 * \cdots * \tau_k} \text{ T-Tup}$$

$$\frac{\overline{\alpha} \vdash Ct \text{ OK} \qquad Ct = (\overline{\tau_0}\ C) \qquad \text{concrete}(C) \qquad \text{repType}(Ct) = \{\overline{V} : \overline{\tau}\} \qquad \Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau_1} \qquad \overline{\tau_1} \leq \overline{\tau}}{\Gamma; \overline{\alpha} \vdash Ct\{\overline{V} = \overline{E}\} : Ct} \text{ T-Rep}$$

$\boxed{\Gamma; \overline{\alpha} \vdash Ct(\overline{E}) \text{ OK}}$

$$\frac{\begin{array}{c}\overline{\alpha} \vdash Ct \text{ OK} \\ Ct = (\overline{\tau_0}\ Sn.Cn) \qquad (\texttt{<abstract>}\ \texttt{class}\ \overline{\alpha_0}\ Cn(\overline{I} : \overline{\tau}) \ldots) \in SigT(Sn) \qquad \Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau_1} \qquad \overline{\tau_1} \leq [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau}\end{array}}{\Gamma; \overline{\alpha} \vdash Ct(\overline{E}) \text{ OK}} \text{ T-Constr}$$

$\boxed{\text{repType}(Ct) = \{\overline{V} : \overline{\tau}\}}$

$$\frac{\begin{array}{c}(\texttt{<<abstract>>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau_1})\ \texttt{<extends}\ Ct\texttt{>}\ \texttt{of}\ \{\overline{Vn} : \overline{\tau_2}\}) \in SigT(Sn) \\ < \text{repType}(Ct) = \{\overline{V} : \overline{\tau_3}\} >\end{array}}{\text{repType}(\overline{\tau}\ Sn.Cn) = [\overline{\alpha} \mapsto \overline{\tau}]\{ < \overline{V} : \overline{\tau_3}, > Sn.\overline{Vn} : \overline{\tau_2}\}} \text{ RepType}$$

Fig. 14. Static semantics of Mini-Eml expressions.

well-formed class type and that the actual arguments have appropriate types, as dictated by the class's specification. Rule T-Rep ensures that an instance expression includes a well-formed class type and that the instance-variable expressions have appropriate types, as dictated by the class's specification. Rule T-Rep uses RepType, which computes a class's representation type.

Figure 15 contains the rules for typechecking patterns. The notation $\text{matchType}(\overline{\tau}, \overline{Pat}) = (\overline{\Gamma}, \overline{\tau'})$ abbreviates $\text{matchType}(\tau_1, Pat_1) = (\Gamma_1, \tau_1') \cdots$

$\boxed{\text{matchType}(\tau, Pat) = (\Gamma, \tau')}$

$$\frac{}{\text{matchType}(\tau, \_) = (\{\}, \tau)}\ \text{T-MatchWild}$$

$$\frac{\text{matchType}(\tau, Pat) = (\Gamma, \tau')}{\text{matchType}(\tau, I \texttt{ as } Pat) = (\Gamma \cup \{(I, \tau')\}, \tau')}\ \text{T-MatchBind}$$

$$\frac{C \leq C' \qquad \text{repType}(\overline{\tau}\, C) = \{\overline{V} : \overline{\tau_0}\} \qquad \text{matchType}(\overline{\tau_0}, \overline{Pat}) = (\overline{\Gamma}, \overline{\tau_1})}{\text{matchType}((\overline{\tau}\, C'), C\{\overline{V} = \overline{Pat}\}) = (\bigcup \overline{\Gamma}, (\overline{\tau}\, C))}\ \text{T-MatchClass}$$

$$\frac{\text{matchType}(\tau_1, Pat_1) = (\Gamma_1, \tau_1') \ \cdots\ \text{matchType}(\tau_k, Pat_k) = (\Gamma_k, \tau_k')}{\text{matchType}(\tau_1 * \cdots * \tau_k, (Pat_1, \ldots, Pat_k)) = (\Gamma_1 \cup \ldots \cup \Gamma_k, \tau_1' * \cdots * \tau_k')}\ \text{T-MatchTup}$$

Fig. 15. Static semantics of Mini-Eml patterns.

$\boxed{\overline{Sn} \vdash Sn.Cn \text{ ITCTransUses}}$

$$\frac{(\texttt{<<abstract>>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau})\ \texttt{<extends}\ (\overline{\tau_0}\ C) > \ldots) \in SigT(Sn) \qquad Sn \in \overline{Sn} \qquad < \overline{Sn} \vdash C \text{ ITCTransUses} >}{\overline{Sn} \vdash Sn.Cn \text{ ITCTransUses}}\ \text{ClassITCTransUses}$$

$\boxed{\overline{Sn} \vdash F \text{ ITCUses}}$

$$\frac{Sn \in \overline{Sn}}{\overline{Sn} \vdash Sn.Fn \text{ ITCUses}}\ \text{FunITCUses}$$

Fig. 16. Well-formedness of the signatures to be accessed during ITC of a structure.

$\text{matchType}(\tau_k, Pat_k) = (\Gamma_k, \tau_k')$. The judgment $\text{matchType}(\tau, Pat) = (\Gamma, \tau')$ checks that a pattern is compatible with type τ. The judgment produces a type environment mapping the identifiers bound in *Pat* to their types. This type environment is used in CaseOK (Figure 12) to typecheck the associated case's body. The type τ' represents the particular subtype of τ to which *Pat* conforms; it is used to give precise types to any identifiers bound to *Pat*, as shown in rule T-MatchBind.

Figure 16 contains the well-formedness rules for the signatures $\overline{Sn}$ (which are "guessed" in StructOK) to be accessed during ITC of a structure Sn. Rule ClassITCTransUses is used by ClassOK in Figure 12 to ensure that $\overline{Sn}$ contains all the signatures that specify a (reflexive, transitive) superclass of a class declared in Sn. Rule FunITCUses is used by CaseOK to ensure that $\overline{Sn}$ contains the signature specifying the associated function for a case declared in Sn. In either case, if $\overline{Sn}$ is required to include some signature Sn', then Sn does indeed statically depend upon Sn' according to the definition of static dependency given in Section 3.1. The rules do not ensure that all statically depended upon signatures are in $\overline{Sn}$, but only those required for precise modular ITC. The rules also do not explicitly forbid $\overline{Sn}$ from including signatures that are not statically depended upon. However, the type soundness proof for Mini-Eml can only rely on the two properties of $\overline{Sn}$ enforced by rules ClassITCTransUses and FunITCUses. Therefore, the proof validates the safety of modular ITC.

$\boxed{\overline{Sn} \vdash F \text{ has-gdefault}}$

$$\frac{\text{owner}(F) = C \qquad \overline{Sn} \vdash F \text{ has-default-for } C}{\overline{Sn} \vdash F \text{ has-gdefault}} \ \textsc{GDefault}$$

$\boxed{\overline{Sn} \vdash \text{funs-have-ldefault-for } C}$

$$\frac{\forall F, C'.[(\overline{Sn} \vdash F \text{ ITCUses} \wedge \text{owner}(F) = C' \wedge C \leq C') \Rightarrow \overline{Sn} \vdash F \text{ has-default-for } C]}{\overline{Sn} \vdash \text{funs-have-ldefault-for } C} \ \textsc{LDefault}$$

$\boxed{\overline{Sn} \vdash F \text{ has-default-for } C}$

$$\frac{\begin{array}{c}(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \rightarrow \tau) \in SigT(Sn) \\ \text{defaultPat}(Mt, C, d) = Pat \qquad (\texttt{extend}\ \texttt{fun}_{Mn}\ \overline{\alpha_0}\ Sn.Fn\ Pat') \in SigT(Sn') \qquad Pat \leq Pat' \qquad Sn' \in \overline{Sn}\end{array}}{\overline{Sn} \vdash Sn.Fn \text{ has-default-for } C} \ \textsc{Default}$$

$\boxed{\text{defaultPat}(T, C, d) = Pat}$

$$\frac{}{\text{defaultPat}(T, C, 0) = \_} \ \textsc{DefZero}$$

$$\frac{d > 0}{\text{defaultPat}(\alpha, C, d) = \_} \ \textsc{DefTypeVar}$$

$$\frac{\text{repType}(\overline{\tau}\ C') = \{\overline{V} : \overline{\tau_0}\} \qquad \text{defaultPat}(\overline{\tau_0}, C, d-1) = \overline{Pat} \qquad d > 0}{\text{defaultPat}((\overline{\tau}\ C'), C, d) = (C'\{\overline{V} = \overline{Pat}\})} \ \textsc{DefClassType}$$

$$\frac{\text{repType}(\overline{\tau}\ C) = \{\overline{V} : \overline{\tau_0}\} \qquad \text{defaultPat}(\overline{\tau_0}, C, d-1) = \overline{Pat} \qquad d > 0}{\text{defaultPat}(\#(\overline{\tau}\ C'), C, d) = (C\{\overline{V} = \overline{Pat}\})} \ \textsc{DefOwnerClassType}$$

$$\frac{\text{defaultPat}(T_1, C, d-1) = Pat_1 \ \cdots \ \text{defaultPat}(T_k, C, d-1) = Pat_k \qquad d > 0}{\text{defaultPat}(T_1 * \ldots * T_k, C, d) = (Pat_1, \ldots, Pat_k)} \ \textsc{DefTupType}$$

$$\frac{d > 0}{\text{defaultPat}(\tau_1 \rightarrow \tau_2, C, d) = \_} \ \textsc{DefFunType}$$

Fig. 17. Modular exhaustiveness checking for MINI-EML.

Figure 17 formalizes the portion of modular ITC that ensures functions are exhaustive. The notation $\text{defaultPat}(\overline{\tau}, C, d) = \overline{Pat}$ abbreviates $\text{defaultPat}(\tau_1, C, d) = Pat_1 \ \cdots \ \text{defaultPat}(\tau_k, C, d) = Pat_k$. Metavariable T ranges over both types and marked types, and metavariable d ranges over nonnegative integers. Rule GDEFAULT checks that a given function has a global default case, and LDEFAULT checks that all available functions whose owners are superclasses of a given class C have a local default case for C. Since a global default case of F is equivalent to a local default case of F for C, where C is the owner of F, the two requirements are able to share the helper rule DEFAULT that performs the checks.

The global- and local-default requirements are enforced by the algorithm described in Section 3 above. Rule DEFAULT generates a (global or local) default pattern and checks that this pattern is at least as specific as the pattern of some available function case. The judgment $\text{defaultPat}(T, C, d) = Pat$ generates a default pattern of (possibly marked) type T. The default pattern dispatches on C in the marked position (if any) of T and accepts any type-correct argument in the other positions. The integer d represents the depth that the generated

$\boxed{Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat \text{ unambiguous}}$

$$\frac{\begin{array}{c}(\texttt{fun}\ \overline{\alpha_1}\ Fn : Mt \to \tau) \in SigT(Sn_1) \qquad Sn = Sn_1 \vee \text{owner}(Mt, Pat) = Sn.Cn \\ \forall Sn' \in \overline{Sn}.\forall(\texttt{extend fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn'). \\ (Sn.Mn \neq Sn'.Mn' \Rightarrow \overline{Sn} \vdash (Pat, Pat') \text{ unambiguous})\end{array}}{Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn_1.Fn\ Pat \text{ unambiguous}}\ \textsc{Amb}$$

$\boxed{\overline{Sn} \vdash (Pat, Pat') \text{ unambiguous}}$

$$\frac{\begin{array}{c}Pat \ncong Pat' \\ \forall Pat_0.((Pat \cap Pat' = Pat_0) \Rightarrow \\ \exists Sn'' \in \overline{Sn}.\exists(\texttt{extend fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'') \in SigT(Sn'').(Pat_0 \cong Pat''))\end{array}}{\overline{Sn} \vdash (Pat, Pat') \text{ unambiguous}}\ \textsc{PairAmb}$$

$\boxed{Pat_1 \cap Pat_2 = Pat}$

$$\frac{}{\_ \cap Pat = Pat}\ \textsc{PatIntWild} \qquad \frac{Pat_1 \cap Pat_2 = Pat}{I \texttt{ as } Pat_1 \cap Pat_2 = Pat}\ \textsc{PatIntBind}$$

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

$$\frac{\overline{Pat_1} \cap \overline{Pat_2} = \overline{Pat}}{(\overline{Pat_1}) \cap (\overline{Pat_2}) = (\overline{Pat})}\ \textsc{PatIntTup} \qquad \frac{Pat_2 \cap Pat_1 = Pat}{Pat_1 \cap Pat_2 = Pat}\ \textsc{PatIntRev}$$

Fig. 18. Modular ambiguity checking for Mini-Eml.

pattern should have. Rule Default chooses the depth non-deterministically, and the Mini-Eml type soundness proof implies that any depth can safely be used. However, as discussed in Section 3, an implementation of the algorithm should choose a large-enough depth to ensure precision.

Figure 18 formalizes the portion of modular ITC that ensures functions are unambiguous. The notation $\overline{Pat'} \cap \overline{Pat''} = \overline{Pat}$ abbreviates $Pat'_1 \cap Pat''_1 = Pat_1 \cdots Pat'_k \cap Pat''_k = Pat_k$. The top-level rule is Amb. The second premise enforces the ambiguity requirement, ensuring that the given function case is declared in the same module as either its associated function or its owner. The final premise in Amb uses PairAmb to check that the given case is unambiguous with each available function case other than itself.

PairAmb uses the pairwise ambiguity algorithm described in Section 3. $Pat \cong Pat'$ denotes that Pat is congruent to Pat'; it is an abbreviation for the two facts $Pat \leq Pat'$ and $Pat' \leq Pat$. Pattern intersection is formalized by the judgment $Pat \cap Pat' = Pat''$. If Pat and Pat' are disjoint then there is no Pat'' such that $Pat \cap Pat' = Pat''$. The rules for pattern intersection are straightforward. The intersection of class patterns (rule PatIntClass) is simplified by the fact that classes support only single inheritance. In particular, if two class patterns are applicable to a common value, then one class must be a subclass of the other.

Figure 19 contains the helper judgments for accessing the class at the owner position of a function, type, and pattern. Finally, the rules defining the

$\boxed{\text{owner}(F) = C}$

$$\frac{(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \rightarrow \tau) \in SigT(Sn) \qquad \text{owner}(Mt) = C}{\text{owner}(Sn.Fn) = C}\ \textsc{OwnerFun}$$

$\boxed{\text{owner}(Mt) = C}$

$$\frac{}{\text{owner}(\#\overline{\tau}\ C) = C}\ \textsc{OwnerClass}$$

$$\frac{\text{owner}(Mt) = C}{\text{owner}(\tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k) = C}\ \textsc{OwnerTup}$$

$\boxed{\text{owner}(Mt, Pat) = C}$

$$\frac{\text{owner}(Mt, Pat) = C}{\text{owner}(Mt, I\ \texttt{as}\ Pat) = C}\ \textsc{OwnerBindPat}$$

$$\frac{\text{owner}(Mt, Pat_i) = C}{\text{owner}(\tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k, (Pat_1, \ldots, Pat_k)) = C}\ \textsc{OwnerTupPat}$$

$$\frac{}{\text{owner}(\#Ct, C\{\overline{V} = \overline{Pat}\}) = C}\ \textsc{OwnerClassPat}$$

Fig. 19. Accessing the owner.

judgments concrete(C), $Pat \leq Pat'$, and $C \leq C'$ in Figures 10 and 11 are borrowed from the dynamic semantics.[8]

4.4 Type Soundness

MINI-EML's type system is *sound*: a well-typed MINI-EML program cannot incur type errors at run time. MINI-EML's type errors are defined implicitly by the *stuck* expressions, which are those expressions that are not values but cannot be further evaluated (because there is no applicable rule in the MINI-EML dynamic semantics). The *match nonexhaustive* and *match ambiguous* type errors are represented by the fact that function invocations lacking a most-specific applicable function case are stuck in the MINI-EML dynamic semantics. MINI-EML's soundness therefore validates the correctness of modular ITC, showing that it is sufficient to ensure that function-case lookup always succeeds at run time.

As is standard [Wright and Felleisen 1994], we prove type soundness via a *progress* theorem and a *type preservation* theorem. The progress theorem says that a well-typed expression can always take a step in the dynamic semantics:

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

[8]Technically, the version of rules CONCRETE and SUBEXT in the static semantics should access the signature table rather than the structure table. It is clear by inspection that the rules do not access anything from a structure that is not also available in its principal signature.

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

Fig. 20. Value declarations and modular ITC.

The type preservation theorem says that evaluation preserves well-typedness:

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

The proofs of these two theorems are provided in Appendix A. 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 function application:

LEMMA. *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)* $= (\rho, E)$.

Proving type preservation is relatively straightforward, as it is completely independent of modular ITC.

5. ML-STYLE MODULES

This section discusses the interaction of EML with an ML-style module system, including structures, signatures, and functors [MacQueen 1984; Milner et al. 1997]. We describe the problems that can arise for modular ITC in this context and sketch some possible solutions.

5.1 Structures

Thus far we have assumed that EML structures contain only a sequence of class, function, and function case declarations. EML structures should also accommodate the ordinary ML declarations. These include the ability to bind a name to a value, provide a synonym for a type, declare an exception, and declare an inner structure. The latter three kinds of declarations can be straightforwardly incorporated, but special care is needed to handle value declarations. Figure 20 shows an example of the problems that can occur. As presented so far, ITC on `BadMod` succeeds, because function `f` has an appropriate case for `C`. However, at run time a *match nonexhaustive* error occurs when the `val` declaration is executed, because `f`'s function case has not yet been declared.

There are several approaches to handling this problem. One solution would be to 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 Figure 20, this semantics would ensure that `f`'s function case is declared before `f` is invoked. An alternative approach would be to make the unit of modularity used in the ITC requirements more fine-grained than an entire structure, with `val` declarations forming the

```
signature ShapeSig = sig
  abstract class Shape() of {}
  fun bad:Shape → unit
  extend fun bad s
end

structure ShapeMod :> ShapeSig = struct
  abstract class Shape() of {}
  fun print:Shape → unit
  fun bad:Shape → unit
  extend fun bad s = print s
end

structure CircleMod = struct
  class Circle() extends Shape() of {}
end
```

Fig. 21. Signature ascription and modular ITC.

boundaries of 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 this approach.

5.2 Signature Ascription

By default, clients of an ML structure *Sn* access its components through the view provided by *Sn*'s principal signature. Information hiding in ML is achieved by explicitly ascribing a signature to a structure. Let metavariable *SigN* range over *signature names*. Clients of a structure

`structure` *Sn* `:>` *SigN* = `struct` ... `end`

may only access *Sn*'s components through the view provided by the signature *SigN*.[9] The ascribed signature may include less information than the structure's principal signature. For example, in ML an ascribed signature may omit specifications for some of the structure's declarations, making them inaccessible to clients.

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 21 shows a simple example of the problems that can occur. `ShapeMod` creates the abstract `Shape` class and two associated functions, `print` and `bad`. ITC in `ShapeMod` succeeds vacuously for both functions since `Shape` is abstract. Because `ShapeSig` is ascribed to `ShapeMod`, `print` is hidden from `ShapeMod`'s clients, who therefore cannot perform ITC on `print`. ITC succeeds for `CircleMod`, because `bad` has a

[9]Standard ML includes two kinds of ascription: *transparent* and *opaque*. Transparent ascription can be desugared into opaque ascription, so we focus exclusively on the latter form.

case handling `Circles`. If `bad` is ever invoked with a `Circle` instance, however, `print` will be invoked, causing a *match nonexhaustive* error.

The example is purposely similar to the `print` example in Figure 7. In that case, the global-default requirement ensures that the problem is modularly detected. Intuitively, the modular ITC problems of signature ascription can be solved in the same way: a set of declarations can be safely hidden if that set could have been declared in a separate structure that passes modular ITC [Millstein and Chambers 2002]. The `print` function in Figure 21 does not satisfy this condition. If `print` were declared in its own structure, the modular requirements would force the existence of a global default case for `print`, since `print` would now be an external function. If `print` had such a case, then the function (and that case) could be hidden via signature ascription, and the problem for `Circle` would be resolved.

Aside from hiding entire declarations, it is useful to hide properties of a declaration. Several properties of classes may be hidden. First, any subset of a class's instance variables may be hidden. This hiding does not preclude clients from creating instances of the class, because instances are created only by invoking constructors, and neither the class's constructor nor any constructor arguments have been 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. Second, a concrete class can be viewed as an abstract one, thereby disallowing clients from instantiating the class. The global- and local-default requirements ensure that clients will provide the appropriate default cases in the face of these supposedly abstract classes. Treating a concrete class as abstract could be useful, for example, to enforce the *singleton pattern* [Gamma et al. 1995], in which a class has a single instance.

Finally, a signature can declare a class C `sealed` [Shalit 1997], which hides C's extensibility: classes declared outside of C's structure may not directly subclass from C. This construct can be used to faithfully model ML-style (nonextensible) datatypes. For example, if `SetMod` in Figure 1 were ascribed to a signature that specified `Set`, `ListSet`, and `CListSet` as `sealed`, then other structures would be disallowed from declaring new (direct or indirect) subclasses of `Set`. In that case, sets form a *sealed hierarchy*. As discussed in Section 3.3, functions in ML can safely omit (local and global) default cases, because datatypes are nonextensible. EML similarly need not enforce the global- and local-default requirements for functions on sealed hierarchies. For example, if the set hierarchy were sealed, then `UnionMod` of Figure 2 could safely omit the first `union` case, which is the global default. Because there would never exist concrete subclasses of `Set` other than `ListSet` and `CListSet`, the second `union` case in `UnionMod` would be sufficient to ensure exhaustiveness.

Hiding inheritance properties of classes is more problematic. It would be useful for a signature to expose only a transitive, rather than the direct, superclass of a class. This ability would reduce the dependence of clients on a class's particular implementation. Unfortunately, this flexibility makes modular ITC unsound. For example, a client of two classes C and C' can write ambiguous

```
structure PointMod = struct
  abstract class Point() of {}
  fun print:Point → unit
end

signature APointSig = sig
  class APoint(x:int,y:int)
    extends Point() of {x:int,y:int}
  extend fun print(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 print
    (ColorPoint {x=x,y=y,color=color}) = ...
  fun getColor:ColorPoint → int
  extend fun getColor
    (ColorPoint {x=x,y=y,color=color}) = color
end
```

Fig. 22. Encoding mixins with EML functors.

function cases that appear to be unambiguous if the fact that C subclasses C' is hidden. It would similarly be useful to ascribe an ML-style type specification to a class declaration, possibly augmenting the specification with *partial revelations* [Nelson 1991] to reveal some of the class's underlying structure.

Finally, a function may be sealed by ascribing an ordinary ML-style value specification to the function and its cases. For example, `union` and its two cases in Figure 2 could be represented in a signature by the specification `val union : (Set * Set) → Set`. Clients can still invoke the sealed `union` function, but its extensibility is hidden (and the # mark is no longer necessary): clients may not add new cases to `union` and do not perform ITC on it. In this way, function sealing allows us to model ML-style (nonextensible) functions. Function sealing is allowed under the same circumstances that the function and its cases may be hidden, thereby ensuring that the sealed function is exhaustive and unambiguous.

5.3 Functors

Standard ML supports *functors*, which are structures parameterized by other structures. In the presence of EML's features, functors can provide a great deal of flexibility. Figure 22 illustrates some of the idioms that would be useful to express. The `PointMod` structure contains a `Point` base class with an associated `print` function. The `Colorize` functor implements a form of *mixin* [Bracha and Cook 1990; Findler and Flatt 1998; Flatt et al. 1998], which is a class parameterized by its superclass. The functor can be *instantiated* by applying it to any structure that conforms to the `APointSig` signature, thereby creating a colored version of that structure's `Point` subclass. An overriding case for the existing `print` function is given, in order to print colored points specially. The functor

```
structure APointMod = struct
  class APoint(x:int,y:int)
    extends Point() of {x:int = x, y:int = y}
  extend fun print(APoint {x=x,y=y}) = ...
end

functor Negate(M:APointSig) = struct
  fun negate:Point → Point
  extend fun negate(Point _) = ...
  extend fun negate(APointMod.APoint _) = ...
  extend fun negate(M.APoint _) = ...
end
```

Fig. 23. Three-valued modular ITC of functor bodies.

also introduces a new function for accessing the color of a colored point, with an associated case.

In ML, each functor body can be safely typechecked once, given only the signature of the argument structure. EML should similarly perform modular ITC once on a functor body, guaranteeing exhaustiveness and unambiguity of all relevant functions, no matter how the functor will be 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 only partial information is known about the relationship between `M.APoint` and other classes. While others have investigated the integration of functors (or related forms of parameterized modules) with classes (e.g. [Fisher and Reppy 1999; Ancona and Zucca 2001; McDirmid et al. 2001]), the interaction of functors with EML's generalization of OO and functional dispatching semantics has not been considered previously.

A possible approach to conservatively performing ITC in the presence of partial information in EML is to generalize the subclass relation in the static semantics to be *three-valued*, saying "don't know" when the partial class hierarchy information is inconclusive. The pattern specificity relation is then also generalized to be three-valued, making use of the generalized subclass relation. Last, modular ITC is modified to be conservative with respect to three-valued subclassing and pattern specificity. In exhaustiveness checking, a local default should be required for any class that *may* subclass an available function's owner. The generated (local or global) default pattern *must* be at least as specific as some function case's pattern. In ambiguity checking, all tests for pattern congruence and disjointness should succeed only if the given patterns *must* be congruent and disjoint, respectively. Similarly, the intersection of two patterns should result in an exact intersection pattern. If the two patterns are neither definitely disjoint nor definitely intersecting, then the associated cases are conservatively considered ambiguous.

As an example, consider three-valued ITC on `negate` in `Negate` of Figure 23. The first case is definitely a global default, so `negate` is exhaustive. The first two cases are found to be unambiguous as usual. The first and third cases are similarly found to be unambiguous: even though the identity of `M.APoint` is not

known, from `APointSig` in Figure 22 it is clear that `M.APoint` strictly subclasses `Point`, and this is enough information to compute the exact intersection of the two patterns. Finally, the second and third cases are found to be ambiguous: their patterns are neither definitely disjoint nor definitely intersecting. Therefore ITC on `negate` fails. Indeed, if `Negate` is ever instantiated with `APointMod`, the resulting `negate` function will be ambiguous for `APoint`.

The restrictions on signature ascription described earlier can severely limit the reusability of functors. For example, the `Colorize` and `Negate` functors can only be instantiated with a class `APoint` that is a direct subclass of `Point`, rather than an indirect (transitive) one. Also, `APoint`'s module must contain a `print` case with the pattern specified in `APointSig`. While relaxing these restrictions would increase functor reusability, it would also force ITC on functor bodies to become much more conservative, in order to account for the new expressiveness of clients. The relaxation could therefore reduce the overall benefit of functors by requiring the type system to disallow too many of them.

A pragmatic way to avoid the restrictions on signature ascription could be to move some of the burden of ITC to clients of the functor. In the limit, EML would perform modular ITC once per instantiation of the functor, on the structure resulting from the instantiation. At the point of instantiation, all the identities of classes and functions in the functor's argument would be known, so ordinary modular ITC as described in Section 3.3 would suffice. It is possible that in practice most of ITC could still be performed on the functor body in isolation, with only a few additional checks performed per instantiation. Such a scheme would provide early feedback about a functor's type correctness while still safely supporting the desired expressiveness.

6. RELATED WORK

In previous work we defined Dubious [Millstein and Chambers 1999, 2002], an expressive multimethod-based OO calculus, along with requirements for modular typechecking of Dubious programs. Dubious was then used as the foundation for our work on MultiJava [Clifton et al. 2000], an extension to Java that supports multimethods as well as the addition of new operations to existing classes. MultiJava illustrates how Dubious's expressiveness and modular typechecking requirements can underlie an extension to a traditional OO language, and EML does the same for a traditional functional language. In particular, in EML we have adapted Dubious's requirements to support modular typechecking of extensible datatypes and functions in an ML-like setting. EML also improves on Dubious's theoretical foundations in several ways. First, we have significantly simplified both the informal and formal presentations of the modularity requirements. Second, the notion of modularity in EML is stronger than that of Dubious: an EML module requires access to less of the program to soundly perform ITC than does a Dubious module. Finally, Dubious does not address modular typechecking in the face of pattern matching, parametric polymorphism, or ML-style modules.

OML [Reppy and Riecke 1996] and $ML_{\leq}$ [Bourdoncle and Merz 1997] were described earlier. Zenger and Odersky [2001] 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. 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 [Garrigue 2000]. As in Zenger's language, functions are not extensible, so existing clients must be modified as new variants with overriding function cases are introduced. Unlike EML, polymorphic variants preserve ML-style type inference.

The *mixin modules* of Duggan and Sourelis [1996] allow datatype and function declarations to be split across multiple modules, thereby providing a form of extensible datatypes and functions.[10] Mixin modules must be explicitly composed with one another and "closed" to form an ordinary ML module containing nonextensible datatypes and functions. Therefore, adding a new mixin module that extends an existing datatype or function requires modifying existing code to explicitly include the new mixin module in the composition. In contrast, EML datatypes and functions may be extended by new modules without modifying existing code. Mixin modules are also not hierarchical and do not support subtyping. Follow-on work [Duggan and Techaubol 2001] incorporates mixin modules into a traditional class-based OO language. A distinction is made between class-based and mixin-based objects; mixin-based objects do not support subtyping. Mixin-based objects support a form of type specialization for methods, which EML lacks.

Jiazzi [McDirmid et al. 2001] is an extension to Java based on *units* [Findler and Flatt 1998; Flatt and Felleisen 1998], a form of parameterized module with recursive linking. The authors show how to encode an *open class pattern* in Jiazzi, whereby a module imports a class and exports a version of that class modified to contain a new method or field. In this way, Jiazzi supports adding both new subclasses and new methods to existing classes. Similar to mixin modules, the final version of a class must be explicitly created by composing the relevant modules.

Work on *predicate dispatching* [Ernst et al. 1998] describes ITC for patterns that are more general than those in EML, including conjunctions, disjunctions, and negations of arbitrary predicates in the host language. An OO-style dispatching semantics is used, with predicate implication as the specificity relation among patterns, and functions are extensible. However, the ITC algorithm is nonmodular, requiring access to the entire program.

[10]Other proposals that also use the name "mixin modules" [Ancona and Zucca 2002; Hirschowitz and Leroy 2002] lack extensible datatypes and functions and are not directly related to our work.

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 longstanding 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. First, Section 2.5 discussed future work related to EML's polymorphic type system. Second, more work is needed to integrate EML with ML-style modules, particularly functors. Section 5 sketched some of the challenges and solutions, but additional study and experience are necessary to find practical requirements that balance expressiveness and modular typechecking. Third, EML's extensibility is currently second class: only statically known classes and functions may be extended. Allowing lexically nested classes and functions to be created and extended would enhance EML's expressiveness and would better integrate its constructs with the ML style. A form of linear types [Wadler 1990] could possibly be used to statically track classes and functions for the purpose of ITC. Finally, it would be useful to investigate practical compilation techniques for EML's extensible datatypes and functions. These techniques can likely build on efficient implementation strategies for multimethod and predicate dispatch [Chambers and Chen 1999].

APPENDIX

A. TYPE SOUNDNESS FOR MINI-EML

This appendix provides the proof of type soundness for MINI-EML. As discussed in Section 4.4, progress and type preservation theorems are proven. The first section below contains a proof of the progress theorem and the key technical lemmas that it depends upon. The second section does the same for the type preservation theorem. Both theorems also rely on several basic facts about MINI-EML, which are stated in the final section.

As in the formal dynamic and static semantics, the proof assumes a fixed structure table ST satisfying the two sanity conditions given in Section 4 and a fixed signature table $SigT$ containing the principal signatures of the structures in ST. Also as in the formal rules, it is assumed that S OK holds for each structure S in ST.

A.1 Progress

The progress theorem is straightforward except for the case when E is a function application. That case follows easily from lemma 1, which is given below.

THEOREM 1 (PROGRESS). *If* $\vdash E : \tau$ *and* E *is not a value, then there exists an* E' *such that* $E \longrightarrow E'$.

PROOF. By (strong) induction on the depth of the derivation of $\vdash E : \tau$. Case analysis of the last rule used in the derivation.

- Case T-ID. Then $E = I$ and $(I, \tau) \in \{\}$, so we have a contradiction. Therefore this rule could not be the last rule used in the derivation.
- Case T-NEW. Then $E = Ct(\overline{E})$ and $Ct = (\overline{\tau}\ Sn.Cn)$ and $\{\}; \bullet \vdash Ct(\overline{E})$ OK and concrete($Sn.Cn$). Therefore by Lemma 32, there exist $\overline{V_1}$ and $\overline{E_1}$ such that $\text{rep}(Ct(\overline{E})) = \{\overline{V_1} = \overline{E_1}\}$. Then by E-NEW we have $E \longrightarrow Ct\{\overline{V_1} = \overline{E_1}\}$.
- Case T-REP. Then $E = Ct\{V_1 = E_1, \ldots, V_k = E_k\}$ and for all $1 \leq i \leq k$ we have $\vdash E_i : \tau_i$ for some τ_i. We have two subcases:
  - —For all $1 \leq i \leq k$, E_i is a value. Then E is a value, contradicting our assumption.
  - —There exists some j such that $1 \leq j \leq k$ and E_j is not a value. Without loss of generality, let j be the smallest integer satisfying this condition, so for all $1 \leq q < j$ we have that E_q is a value. By induction, there exists an E'_j such that $E_j \longrightarrow E'_j$. Therefore by E-REP we have $Ct\{V_1 = E_1, \ldots, V_k = E_k\} \longrightarrow Ct\{V_1 = E_1, \ldots, V_{j-1} = E_{j-1}, V_j = E'_j, V_{j+1} = E_{j+1}, \ldots, V_k = E_k\}$.
- Case T-FUN. Then $E = \overline{\tau}\ Sn.Fn$. Then E is a value, contradicting our assumption.
- Case T-TUP. Then $E = (E_1, \ldots, E_k)$ and $\tau = \tau_1 * \cdots * \tau_k$ and for all $1 \leq i \leq k$ we have $\vdash E_i : \tau_i$. We have two subcases:
  - —For all $1 \leq i \leq k$, E_i is a value. Then E is a value, contradicting our assumption.
  - —There exists some j such that $1 \leq j \leq k$ and E_j is not a value. Without loss of generality, let j be the smallest integer satisfying this condition, so for all $1 \leq q < j$ we have that E_q is a value. By induction, there exists an E'_j such that $E_j \longrightarrow E'_j$. Therefore by E-TUP we have $(E_1, \ldots, E_k) \longrightarrow (E_1, \ldots, E_{j-1}, E'_j, E_{j+1}, \ldots, E_k)$.
- Case T-APP. Then $E = E_1\ E_2$ and $\vdash E_1 : \tau_2 \rightarrow \tau$ and $\vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. We have three subcases:
  - —E_1 is not a value. Then by induction, there exists an E'_1 such that $E_1 \longrightarrow E'_1$. Therefore by E-APP1 we have $E_1\ E_2 \longrightarrow E'_1\ E_2$.
  - —E_1 is a value, but E_2 is not a value. Then by induction, there exists an E'_2 such that $E_2 \longrightarrow E'_2$. Therefore by E-APP2 we have $E_1\ E_2 \longrightarrow E_1\ E'_2$.
  - —Both E_1 and E_2 are values. Since $\vdash E_1 : \tau_2 \rightarrow \tau$ and E_1 is a value, the last rule in the derivation of $\vdash E_1 : \tau_2 \rightarrow \tau$ must be T-FUN, so E_1 has the form Fv. Therefore by Lemma 1 we have that there exist ρ_0 and E_0 such that most-specific-case-for $(Fv, E_2) = (\rho_0, E_0)$. Let $\rho_0 = \{(\overline{I}, \overline{v})\}$. Then by E-APPRED we have $Fv\ E_2 \longrightarrow [\overline{I} \mapsto \overline{v}]E_0$. □

This lemma says that a most-specific applicable function case exists for each type-correct function application. It follows easily from Lemmas 2 and 7, which say that all functions are exhaustive and unambiguous, respectively.

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 $(Fv, v) = (\rho, E)$.*

PROOF. Let $Fv = (\overline{\tau}\ F)$. By Lemma 2, there exists some $Sn \in \mathrm{dom}(SigT)$, some $(\texttt{extend}\ \texttt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat) \in SigT(Sn)$, and some environment ρ_0 such that $\mathrm{match}(v, Pat) = \rho_0$. Then by Lemma 7 there exists some $Sn' \in \mathrm{dom}(SigT)$, some $(\texttt{extend}\ \texttt{fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn')$, and some ρ' such that $\mathrm{match}(v, Pat') = \rho'$ and $\forall Sn'' \in \mathrm{dom}(SigT).\forall(\texttt{extend}\ \texttt{fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'') \in SigT(Sn'').\forall\rho''.((\mathrm{match}(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

By the definition of $SigT$ we have that $Sn' \in \mathrm{dom}(ST)$ and there exists some E' such that $(\texttt{extend}\ \texttt{fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in ST(Sn')$ and $\forall Sn'' \in \mathrm{dom}(ST).\forall(\texttt{extend}\ \texttt{fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'' = E'') \in ST(Sn'').\forall\rho''.((\mathrm{match}(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

Since $\vdash Fv : \tau_2 \to \tau$, by T-FUN we have $F = Sn_0.Fn_0$ and $(\texttt{fun}\ \overline{\alpha_0}\ Fn_0 : Mt_0 \to \tau_0) \in SigT(Sn_0)$ and $|\overline{\alpha_0}| = |\overline{\tau}|$. Since $(\texttt{extend}\ \texttt{fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn')$, by CASEOK we have $|\overline{\alpha_1}| = |\overline{\alpha_0}|$. Therefore we have $|\overline{\alpha_1}| = |\overline{\tau}|$. Then by LOOKUP there exist ρ and E such that most-specific-case-for $((\overline{\tau}\ F), v) = (\rho, E)$. □

A.1.1 *Exhaustiveness.* These lemmas prove that all functions are exhaustive. They make use of the notion of the *owner* of a value v with respect to a marked type Mt. Intuitively, v's owner is the class in v located at the owner position as specified by Mt. It is defined via the following two inference rules:

$\boxed{\text{owner}\ (Mt, v) = C}$

$$\frac{\text{owner}(Mt, v_i) = C}{\text{owner}(\tau_1 * \ldots * \tau_{i-1} * Mt * \tau_{i+1} * \ldots * \tau_k(v_1, \ldots, v_k)) = C}\ \textsc{OwnertupVal}$$

$$\frac{}{\text{owner}(\#Ct, (\overline{\tau}C)\{\overline{V} = \overline{v}\}) = C}\ \textsc{OwnerInstance}$$

This is the main exhaustiveness lemma, which says that every type-correct argument value for a function has at least one applicable function case. Its proof follows from the global- and local-default requirements. If the function has a global default case, then the result follows from the definition of a global default. Otherwise, the function must be internal. Then there must be a local default case for each type-correct argument value's owner, and the result follows from the definition of a local default case.

LEMMA 2 (EXHAUSTIVENESS). *If* $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$ *and* $\vdash v : \tau_2'$ *and* $\tau_2' \leq \tau_2$, *then there exist some* $Sn \in dom(SigT)$, *some* $(\texttt{extend}\ \texttt{fun}_{\mathrm{Mn}}\ \overline{\alpha_1}\ F\ Pat) \in SigT(Sn)$, *and some environment* ρ *such that* $match(v, Pat) = \rho$.

PROOF. Since $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$, by T-FUN we have $F = Sn'.Fn$ and $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in SigT(Sn')$ and $|\overline{\alpha}| = |\overline{\tau}|$ and $\tau_2 \to \tau = [\overline{\alpha} \mapsto \overline{\tau}](\hat{M}t \to \tau_0)$. By the definition of $SigT$, also $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in ST(Sn')$. Let $ST(Sn') = (\texttt{structure}\ Sn' = \texttt{struct}\ \overline{D}\ \texttt{end})$. Then by STRUCTOK we have $\overline{Sn} \subseteq \mathrm{dom}(SigT)$ and $\overline{Sn} \vdash (\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0)$ OK in Sn', so by FUNOK we have that $\mathrm{owner}(Mt) = Sn''.Cn$. Then by Lemma 3 there exists some class C such that $\mathrm{owner}(Mt, v) = C$ and $\mathrm{concrete}(C)$ and $C \leq Sn''.Cn$. Also by FUNOK we have either $\overline{Sn} \vdash F$ has-gdefault or $Sn' = Sn''$. We consider these cases separately.

- Case $\overline{Sn} \vdash F$ has-gdefault. By GDEFAULT we have owner$(F) = C'$ and $\overline{Sn} \vdash F$ has-default-for C'. By OWNERFUN, C'=$Sn''.Cn$. Then by Lemma 4 there exists some $Sn \in \overline{Sn}$, some (`extend` $\texttt{fun}_{Mn}$ $\overline{\alpha_1}$ F Pat) $\in SigT(Sn)$, and some environment ρ such that match$(v, Pat) = \rho$. Since $\overline{Sn} \subseteq$ dom$(SigT)$, we have $Sn \in$ dom$(SigT)$ and the result is shown.
- Case $Sn' = Sn''$. Let $C = Sn_0.Cn_0$. Since concrete(C), by CONCRETE we have (`class` $\overline{\alpha_0}$ $Cn_0 \ldots$) $\in ST(Sn_0)$. Let $ST(Sn_0) =$ (`structure` $Sn_0 =$ `struct` $\overline{D_0}$ `end`). Then by STRUCTOK we have $\overline{Sn_0} \subseteq$ dom$(SigT)$ and $\overline{Sn_0} \vdash$ `class` $\overline{\alpha_0}$ $Cn_0 \ldots$ OK in Sn_0, so by CLASSOK we have concrete$(C) \Rightarrow \overline{Sn_0} \vdash$ funs-have-ldefault-for C. Since concrete(C) holds, we have $\overline{Sn_0} \vdash$ funs-have-ldefault-for C.

  Also by CLASSOK we have $\overline{Sn_0} \vdash C$ ITCTransUses. Since $C \leq Sn''.Cn$ and $Sn' = Sn''$, by Lemma 37 we have $Sn' \in \overline{Sn_0}$.

  Since $F = Sn'.Fn$ and $Sn' \in \overline{Sn_0}$, by FUNITCUSES we have $\overline{Sn_0} \vdash F$ ITCUses. Since (`fun` $\overline{\alpha}$ Fn : $Mt \rightarrow \tau_0$) $\in SigT(Sn')$ and owner$(Mt) = Sn'.Cn$, by OWNERFUN we have owner$(F) = Sn'.Cn$. Also, we showed above that $C \leq Sn'.Cn$. Therefore, since $\overline{Sn_0} \vdash$ funs-have-ldefault-for C, by LDEFAULT we have $\overline{Sn_0} \vdash F$ has-default-for C. By SUBREF $C \leq C$, so by Lemma 4 there exists some $Sn \in \overline{Sn_0}$, some (`extend` $\texttt{fun}_{Mn}$ $\overline{\alpha_1}$ $Sn.Fn$ Pat) $\in SigT(Sn)$, and some environment ρ such that match$(v, Pat) = \rho$. Since $\overline{Sn_0} \subseteq$ dom$(SigT)$, we have $Sn \in$ dom$(SigT)$, and the result is shown. □

This lemma says that a value conforming to a marked type has a well-defined owner (with respect to that marked type), which is a subclass of the marked type's owner.

LEMMA 3. *If* $\vdash v : \tau'$ *and* $\tau' \leq \tau$ *and* $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t$ *and owner(Mt)* $= C'$, *then there exists some class C such that owner(Mt, v)* $= C$ *and concrete(C) and* $C \leq C'$.

PROOF. By induction on the depth of the derivation of $\vdash v : \tau'$. Case analysis of the last rule used in the derivation.

- Case T-REP. Then v has the form $(\overline{\tau_0}\ C)\{\overline{V} = \overline{v}\}$ and $\tau' = (\overline{\tau_0}\ C)$ and concrete(C). Since $\tau' \leq \tau$, by Lemma 13 τ has the form $(\overline{\tau_1}\ C'')$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t$, $\hat{M}t$ has the form $(\overline{\tau_2}\ C'')$, and by the grammar for marked types Mt must be $\#(\overline{\tau_2}\ C'')$. Then by OWNERINSTANCE we have owner$(\#(\overline{\tau_2}\ C''), (\overline{\tau_0}\ C)\{\overline{V} = \overline{v}\}) = C$. We're given $\tau' \leq \tau$, so by lemma 15 we have $C \leq C''$. Since owner$(Mt) = C'$, by OWNERCLASS we have $C' = C''$, so $C \leq C'$.
- Case T-FUN. Then v has the form $(\overline{\tau_0}\ F)$ and τ' has the form $\tau_1 \rightarrow \tau_2$. Therefore by Lemma 18 τ has the form $\tau_1' \rightarrow \tau_2'$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t$, $\hat{M}t$ has the form $\tau_1'' \rightarrow \tau_2''$, but this contradicts the grammar of marked types. Therefore, T-FUN cannot be the last rule in the derivation.
- Case T-Tup: Then v has the form $(v_1, \ldots, v_k)$ and τ' has the form $\tau_1' * \cdots * \tau_k'$ and for all $1 \leq j \leq k$ we have $\vdash v_j : \tau_j'$. Therefore by Lemma 20 τ has the form $\tau_1 * \cdots * \tau_k$, where for all $1 \leq j \leq k$ we have $\tau_j' \leq \tau_j$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t$, $\hat{M}t$ has the form $\tau_1'' * \cdots * \tau_k''$, and by the grammar for marked types Mt is

$\tau''_1 * \cdots * \tau''_{i-1} * Mt_i * \tau''_{i+1} * \cdots * \tau''_k$, where $\hat{Mt}_i = \tau''_i$. We're given owner(Mt) = C', so by OwnerTup we have owner(Mt_i) = C'.

Therefore we have $\vdash v_i : \tau'_i$ and $\tau'_i \leq \tau_i$ and $\tau_i = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}_i$ and owner(Mt_i) = C', so by induction there exists C such that owner(Mt_i, v_i) = C and concrete(C) and $C \leq C'$. By OwnerTupVal we have owner($\tau''_1 * \cdots * \tau''_{i-1} * Mt_i * \tau''_{i+1} * \cdots * \tau''_k, (v_1, \ldots, v_k)$) = C, so the result follows. □

This lemma validates the has-default-for judgment: if a function F has a default for class C, then F has at least one applicable function case for each type-correct argument value whose owner is a subclass of C.

Lemma 4. *If* $\vdash v : \tau'_2$ *and* $\tau'_2 \leq \tau_2$ *and* $\tau_2 = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$ *and (*`fun` $\overline{\alpha}$ $Fn : Mt \rightarrow \tau_0$*)* $\in SigT(Sn)$ *and owner(*Mt, v*)* $= C_0$ *and* $C_0 \leq C$ *and* $\overline{Sn} \vdash Sn.Fn$ *has-default-for C, then there exists some* $Sn' \in \overline{Sn}$*, some (*`extend` $\texttt{fun}_{Mn}$ $\overline{\alpha_1}$ $Sn.Fn$ Pat*)* $\in SigT(Sn')$*, and some environment* ρ *such that match(*v, Pat*)* $= \rho$*.*

Proof. Since $\overline{Sn} \vdash Sn.Fn$ has-default-for C, by Default we have defaultPat(Mt, C, d) = Pat'. Therefore by Lemma 5 there exists ρ' such that match(v, Pat') = ρ'. Also by Default we have (`extend` $\texttt{fun}_{Mn}$ $\overline{\alpha_1}$ $Sn.Fn$ Pat) $\in SigT(Sn')$ and $Pat' \leq Pat$ and $Sn' \in \overline{Sn}$. By Lemma 25 there exists ρ such that match(v, Pat) = ρ, so the result follows. □

The next two lemmas validate the defaultPat judgment, by showing that the generated pattern is in fact a (global or local) default pattern: each type-correct argument value matches the generated pattern. The first lemma handles patterns generated with respect to marked types, and the second lemma handles patterns generated with respect to ordinary unmarked types.

Lemma 5. *If* $\vdash v : \tau'$ *and* $\tau' \leq \tau$ *and* $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$ *and owner(*Mt, v*)* $= C_0$ *and* $C_0 \leq C$ *and defaultPat(*Mt, C, d*)* $= Pat$*, then there exists* ρ *such that match(*v, Pat*)* $= \rho$*.*

Proof. By strong induction on the depth of the derivation of defaultPat(Mt, C, d) = Pat. Case analysis of the last rule in the derivation.

- Case DefZero. Then Pat has the form _, so by E-MatchWild we have match(v, _) = {}.
- Case DefOwnerClassType. Then Mt has the form #($\overline{\tau_1}\ C'$) and Pat has the form ($C\{\overline{V} = \overline{Pat}\}$) and repType($\overline{\tau_1}\ C$) = $\{\overline{V} : \overline{\tau'}\}$ and defaultPat($\overline{\tau'}, C, d-1$) = $\overline{Pat}$ and $d > 0$. By Lemma 23 we have repType($[\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ C$) = $[\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau'}\}$. Since owner(#($\overline{\tau_1}\ C'$), v) = C_0, by OwnerInstance we have that v is of the form ($\overline{\tau_0}\ C_0$)$\{\overline{V_2} = \overline{v_2}\}$.

  Since we're given that $\vdash v : \tau'$, by T-Rep we have that $\tau' = (\overline{\tau_0}\ C_0)$ and $\bullet \vdash (\overline{\tau_0}\ C_0)$ OK and repType($\overline{\tau_0}\ C_0$) = $\{\overline{V_2} : \overline{\tau_2}\}$ and $\vdash \overline{v_1} : \overline{\tau'_2}$ and $\overline{\tau'_2} \leq \overline{\tau_2}$. We're given that $\tau' \leq \tau$, so that means $(\overline{\tau_0}\ C_0) \leq ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ C')$, and by Lemma 14 we have $\overline{\tau_0} = [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}$. Since $C_0 \leq C$ and $\bullet \vdash (\overline{\tau_0}\ C_0)$ OK, by Lemma 16 we

have $(\overline{\tau_0}\ C_0) \leq (\overline{\tau_0}\ C)$. Therefore by Lemma 36 we have $\{\overline{V_2} : \overline{\tau_2}\} = \{\overline{V} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}, \overline{V_3} : \overline{\tau_3}\}$.

Therefore there is some prefix $\overline{v_4}$ of $\overline{v_2}$ and some prefix $\overline{\tau_4}$ of $\overline{\tau_2'}$ such that $\vdash \overline{v_4} : \overline{\tau_4}$ and $\overline{\tau_4} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}$ and defaultPat$(\overline{\tau'}, C, d-1) = \overline{Pat}$, so by Lemma 6, there exists $\overline{\rho}$ such that match$(\overline{v_4}, \overline{Pat}) = \bigcup \overline{\rho}$. Finally, we're given $C_0 \leq C$, so by E-MatchClass we have match$((\overline{\tau_0}\ C_0)\{\overline{V_2} = \overline{v_2}\}, (C\{\overline{V} = \overline{Pat}\})) = \bigcup \overline{\rho}$.

- Case DefTupType. Then Mt has the form $\tau_1 * \cdots * \tau_{i-1} * Mt_i * \tau_{i+1} * \cdots * \tau_k$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and for all $1 \leq j \leq k$ such that $j \neq i$ we have defaultPat$(\tau_j, C, d-1) = Pat_j$ and we have defaultPat$(Mt_i, C, d-1) = Pat_i$. Let $\tau_i = \hat{Mt}_i$. Since $\tau' \leq [\overline{\alpha} \mapsto \overline{\tau}](\tau_1 * \cdots * \tau_k)$, by Lemma 19 we have that τ' has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq j \leq k$ we have $\tau_j' \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_j$. Since $\vdash v : \tau'$, by T-Tup we have that v has the form $(v_1, \ldots, v_k)$ and for all $1 \leq j \leq k$ we have $\vdash v_j : \tau_j'$. Therefore by Lemma 6, for all $1 \leq j \leq k$ such that $j \neq i$ we have that there exists some ρ_j such that match$(v_j, Pat_j) = \rho_j$. We're given that owner$(Mt, v) = C_0$, so by OwnerTupVal we have owner$(Mt_i, v_i) = C_0$. Therefore by induction we have that there exists some ρ_i such that match$(v_i, Pat_i) = \rho_i$. Then by E-MatchTup we have match$(v, Pat) = \rho_1 \cup \cdots \cup \rho_k$. □

Lemma 6. *If $\vdash v : \tau'$ and $\tau' \leq \tau$ and $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau_0$ and defaultPat$(\tau_0, C_0, d) = Pat$, then there exists ρ such that match$(v, Pat) = \rho$.*

Proof. By strong induction on the depth of the derivation of defaultPat$(\tau_0, C_0, d) = Pat$. Case analysis of the last rule in the derivation.

- Case DefZero or DefTypeVar or DefFunType. Then Pat has the form _, so by E-MatchWild we have match$(v, _) = \{\}$.
- Case DefClassType. Then τ_0 has the form $(\overline{\tau_0}\ C)$ and Pat has the form $(C\{\overline{V} = \overline{Pat}\})$ and repType$(\overline{\tau_0}\ C) = \{\overline{V} : \overline{\tau'}\}$ and defaultPat$(\overline{\tau'}, C_0, d-1) = \overline{Pat}$ and $d > 0$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau_0$, by Lemma 23 we have repType$(\tau) = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau'}\}$. Further, $\tau = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_0}\ C) = ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}\ C)$. Since $\tau' \leq \tau$, by Lemma 12 τ' has the form $(\overline{\tau_1}\ C')$. Since $\vdash v : \tau'$, by T-Rep v has the form $(\overline{\tau_1}\ C')\{\overline{V_1} = \overline{v_1}\}$ and $\bullet \vdash (\overline{\tau_1}\ C')$ OK and repType$(\overline{\tau_1}\ C') = \{\overline{V_1} : \overline{\tau_1}\}$ and $\vdash \overline{v_1} : \overline{\tau_1'}$ and $\overline{\tau_1'} \leq \overline{\tau_1}$.

  Since $(\overline{\tau_1}\ C') \leq ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}\ C)$, by Lemma 15 we have $C' \leq C$. Further, by Lemma 36 we have that $\{\overline{V_1} : \overline{\tau_1}\} = \{\overline{V} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}, \overline{V_2} : \overline{\tau_2}\}$. Therefore there is some prefix $\overline{v_3}$ of $\overline{v_1}$ and some prefix $\overline{\tau_3}$ of $\overline{\tau_1'}$ such that $\vdash \overline{v_3} : \overline{\tau_3}$ and $\overline{\tau_3} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}$ and defaultPat$(\overline{\tau'}, C_0, d-1) = \overline{Pat}$. Therefore by induction, match$(\overline{v_3}, \overline{Pat}) = \overline{\rho}$. Therefore by E-MatchClass we have match$((\overline{\tau_1}\ C')\{\overline{V_1} = \overline{v_1}\}, (C\{\overline{V} = \overline{Pat}\})) = \bigcup \overline{\rho}$.
- Case DefTupType. Then τ_0 has the form $\tau_1 * \cdots * \tau_k$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and for all $1 \leq i \leq k$ we have defaultPat$(\tau_i, C_0, d-1) = Pat_i$ and $d > 0$. Since $\tau' \leq [\overline{\alpha} \mapsto \overline{\tau}](\tau_1 * \cdots * \tau_k)$, by Lemma 19 we have that τ' has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq i \leq k$ we have $\tau_i' \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_i$. Since $\vdash v : \tau'$, by T-Tup we have that v has the form $(v_1, \ldots, v_k)$ and for all $1 \leq i \leq k$ we have $\vdash v_i : \tau_i'$. Therefore by induction, for all $1 \leq i \leq k$ we have that there exists some ρ_i such that match$(v_i, Pat_i) = \rho_i$. Then by E-MatchTup we have match$(v, Pat) = \rho_1 \cup \cdots \cup \rho_k$. □

A.1.2 *Ambiguity.* The following lemma says that if a value has at least one applicable function case then it has a most-specific applicable case. The lemma thereby validates our modular ambiguity checking algorithm and requirement. The lemma is proven by induction on the number of applicable function cases that are not overridden by the given one. If there are none (other than the given case itself), then the given case is the most-specific applicable case. Otherwise, another applicable function case is not overridden by the given one. Then we show that, by modular ambiguity checking, there must exist a resolving case, and the result follows by induction on the resolving case.

LEMMA 7 (UNAMBIGUITY). *If* $\vdash v : \tau$ *and* $Sn \in dom(SigT)$ *and* (`extend` $\texttt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat$) $\in SigT(Sn)$ *and* $match(v, Pat) = \rho$, *then there exists some* $Sn' \in dom(SigT)$, *some* (`extend` $\texttt{fun}_{Mn}\ \overline{\alpha_1}\ F\ Pat'$) $\in SigT(Sn')$, *and some* ρ' *such that* $match(v, Pat') = \rho'$ *and* $\forall Sn'' \in dom(SigT).\forall$(`extend` $\texttt{fun}_{Mn}\ \overline{\alpha_2}\ F\ Pat''$) $\in SigT(Sn'')$. $\forall \rho''.((match(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

PROOF. By (strong) induction on the number of function cases of F that are applicable to v but are not overridden by $Sn.Mn$. These are the cases of the form (`extend` $\texttt{fun}_{Mn_0}\ \overline{\alpha_0}\ F\ Pat_0$) such that (`extend` $\texttt{fun}_{Mn_0}\ \overline{\alpha_0}\ F\ Pat_0$) $\in SigT(Sn_0)$ for some $Sn_0 \in \mathrm{dom}(SigT)$, and $\mathrm{match}(v, Pat_0) = \rho_0$ for some ρ_0, and $Pat \not< Pat_0$.

- Case there are zero function cases of F that are applicable to v but are not overridden by $Sn.Mn$. We're given that $Sn \in \mathrm{dom}(SigT)$ and (`extend` $\texttt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat$) $\in SigT(Sn)$ and $\mathrm{match}(v, Pat) = \rho$. Further, since it cannot both be the case that $Pat \leq Pat$ and $Pat \not\leq Pat$, we have $Pat \not< Pat$. Therefore, $Sn.Mn$ itself is applicable to v but is not overridden by $Sn.Mn$, so we have a contradiction.
- Case there is exactly one function case of F that is applicable to v but is not overridden by $Sn.Mn$. Then from the previous case we know that $Sn.Mn$ must itself be that function case. Therefore it follows that $\forall Sn'' \in \mathrm{dom}(SigT).\forall$(`extend` $\texttt{fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat''$) $\in SigT(Sn'').\forall \rho''.$ $((\mathrm{match}(v, Pat'') = \rho'' \wedge Sn.Mn \neq Sn''.Mn'') \Rightarrow Pat < Pat'')$. Then the result follows.
- There are $k > 1$ function cases of F that are applicable to v but are not overridden by $Sn.Mn$. Let (`extend` $\texttt{fun}_{Mn_1}\ \overline{\alpha_3}\ F\ Pat_1$) be one such function case, so (`extend` $\texttt{fun}_{Mn_1}\ \overline{\alpha_3}\ F\ Pat_1$) $\in SigT(Sn_1)$ for some $Sn_1 \in \mathrm{dom}(SigT)$, and $\mathrm{match}(v, Pat_1) = \rho_1$ for some ρ_1, and $Pat \not< Pat_1$. Since $k > 1$, at least one of the function cases satisfying the conditions is not $Sn.Mn$, so assume without loss of generality that $Sn.Mn \neq Sn_1.Mn_1$.

  By CASEOK we have $\mathrm{matchType}(\tau_0, Pat) = (\Gamma_0, \tau_0')$ and $\mathrm{matchType}(\tau_1, Pat_1) = (\Gamma_1, \tau_1')$. We're given that $\vdash v : \tau$. Finally, we saw above that $\mathrm{match}(v, Pat) = \rho$ and $\mathrm{match}(v, Pat_1) = \rho_1$. Therefore by Lemma 30 there exists some Pat_{int} such that $Pat \cap Pat_1 = Pat_{int}$. Further, by Lemma 8 we have $\mathrm{dom}(SigT) \vdash (Pat, Pat_1)$ unambiguous. Therefore by PAIRAMB we have that $Pat \not\cong Pat_1$ and there exists some $Sn_2 \in \mathrm{dom}(SigT)$ and some (`extend` $\texttt{fun}_{Mn_2}\ \overline{\alpha_4}\ F\ Pat_2$) $\in SigT(Sn_2)$ such that $Pat_{int} \cong Pat_2$. Since $\mathrm{match}(v, Pat) = \rho$ and $\mathrm{match}(v, Pat_1) = \rho_1$ and $Pat \cap Pat_1 = Pat_{int}$, by Lemma 31 there exists some ρ_{int} such that $\mathrm{match}(v, Pat_{int}) = \rho_{int}$. Then since $Pat_{int} \leq Pat_2$, by Lemma 25 there exists ρ_2 such that $\mathrm{match}(v, Pat_2) = \rho_2$.

So we have shown there exists some $Sn_2 \in \mathrm{dom}(SigT)$ and some (`extend` $\mathtt{fun}_{Mn_2}\ \overline{\alpha_4}\ F\ Pat_2) \in SigT(Sn_2)$ and some ρ_2 such that $\mathrm{match}(v, Pat_2) = \rho_2$. Let l be the number of function cases of F that are applicable to v but are not overridden by $Sn_2.Mn_2$. This case is finished by showing that $l < k$, so that the result follows by induction with respect to $Sn_2.Mn_2$.

First we show that $l \leq k$. Consider some $Sn_0 \in \mathrm{dom}(SigT)$, some (`extend` $\mathtt{fun}_{Mn_0}\ \overline{\alpha_0}\ F\ Pat_0) \in SigT(Sn_0)$, and some ρ_0 such that $\mathrm{match}(v, Pat_0) = \rho_0$ and $Pat_2 \not< Pat_0$. We will show that also $Pat \not< Pat_0$.

Since $Pat \cap Pat_1 = Pat_{int}$, by Lemma 28 we have that $Pat_{int} \leq Pat$ and $Pat_{int} \leq Pat_1$. Since $Pat_2 \leq Pat_{int}$, by Lemma 26 also $Pat_2 \leq Pat$ and $Pat_2 \leq Pat_1$. Since $Pat_2 \not< Pat_0$, either $Pat_2 \not\leq Pat_0$ or $Pat_0 \leq Pat_2$. We consider these cases in turn.

—Case $Pat_2 \not\leq Pat_0$. Suppose $Pat \leq Pat_0$. Since $Pat_2 \leq Pat$, by Lemma 26 we have $Pat_2 \leq Pat_0$, contradicting the assumption of this case. Therefore $Pat \not\leq Pat_0$, so also $Pat \not< Pat_0$.

—Case $Pat_0 \leq Pat_2$. We showed above that $Pat_2 \leq Pat$. Then by Lemma 26 $Pat_0 \leq Pat$, so $Pat \not< Pat_0$.

Therefore we have shown that every function case of F that is applicable to v and is not overridden by $Sn_2.Mn_2$ is also not overridden by $Sn.Mn$, so $l \leq k$. To finish the proof, we show that there exists a function case of F that is applicable to v and is not overridden by $Sn.Mn$ but *is* overridden by $Sn_2.Mn_2$. We showed in the first case above that $Sn.Mn$ is not overridden by itself. We will show that $Sn.Mn$ is overridden by $Sn_2.Mn_2$. We do this by proving that $Pat_2 < Pat$. We showed above that $Pat_2 \leq Pat$, so we simply need to prove that $Pat \not\leq Pat_2$. Suppose $Pat \leq Pat_2$. We're given that $Pat \not< Pat_1$ and $Pat \not\geq Pat_1$. Therefore, $Pat \not\leq Pat_1$. On the other hand, since $Pat \leq Pat_2$ and $Pat_2 \leq Pat_1$, by Lemma 26 we have $Pat \leq Pat_1$, and we have a contradiction. □

This lemma says that every pair of function cases belonging to the same function is unambiguous. If the two cases are both available during some structure's modular ITC, then the result follows from that structure's ambiguity checks. Otherwise, the modular ambiguity requirement ensures that the cases are disjoint, so they are also unambiguous.

LEMMA 8. *If (*`extend` $\mathtt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat$*)* $\in SigT(Sn)$ *and (*`extend` $\mathtt{fun}_{Mn}\ \overline{\alpha'}\ F\ Pat'$*)* $\in SigT(Sn')$ *and* $Sn.\mathrm{Mn} \neq Sn'.\mathrm{Mn}'$*, then* $dom(SigT) \vdash (Pat, Pat')$ *unambiguous.*

PROOF. Since (`extend` $\mathtt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat) \in SigT(Sn)$, by STRUCTOK we have $\overline{Sn} \subseteq \mathrm{dom}(SigT)$ and $\overline{Sn} \vdash$ (`extend` $\mathtt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E$) OK in Sn for some $\overline{Sn}$ and E, so by CASEOK we have $Sn; \overline{Sn} \vdash$ `extend` $\mathtt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat$ unambiguous. Similarly, $\overline{Sn'} \subseteq \mathrm{dom}(SigT)$ and $Sn'; \overline{Sn'} \vdash$ `extend` $\mathtt{fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat'$ unambiguous, for some $\overline{Sn'}$. There are several cases.

- Case $Sn' \in \overline{Sn}$. Since $Sn.Mn \neq Sn'.Mn'$ and $Sn; \overline{Sn} \vdash$ `extend` $\mathtt{fun}_{Mn}\ \overline{\alpha}\ F\ Pat$ unambiguous, by AMB we have $\overline{Sn} \vdash (pat, pat')$ unambiguous. Since $\overline{Sn} \subseteq \mathrm{dom}(SigT)$, by AMB also $\mathrm{dom}(SigT) \vdash (Pat, Pat')$ unambiguous.
- Case $Sn \in \overline{Sn}$. Symmetric to the above case.

- Case $Sn' \notin \overline{Sn}$ and $Sn \notin \overline{Sn'}$. Since $Sn; \overline{Sn} \vdash$ `extend fun`$_{Mn}$ $\overline{\alpha}\ F\ Pat$ unambiguous, by AMB we have $F = Sn_1.Fn$ and (`fun` $\overline{\alpha_1}\ Fn : Mt \rightarrow \tau) \in SigT(Sn_1)$ and $Sn = Sn_1 \vee \text{owner}(Mt, Pat) = Sn.Cn$. Similarly, $Sn' = Sn_1 \vee (\text{owner}(Mt, Pat') = Sn'.Cn'$. We have three sub-cases.
  - —Case $Sn' = Sn_1$. Since $\overline{Sn} \vdash$ (`extend fun`$_{Mn}$ $\overline{\alpha}\ F\ Pat = E$) OK in Sn, by CASEOK we have $\overline{Sn} \vdash Sn'.Fn$ ITCUses. Then by FUNITCUSES $Sn' \in \overline{Sn}$, so we have a contradiction.
  - —Case $Sn = Sn_1$. Symmetric to the above case.
  - —Case $\text{owner}(Mt, Pat) = Sn.Cn$ and $\text{owner}(Mt, Pat') = Sn'.Cn'$. First we show that Pat and Pat' are disjoint: there does not exist Pat_0 such that $Pat \cap Pat' = Pat_0$. Suppose not. Then by Lemma 29 either $Sn.Cn \leq Sn'.Cn'$ or $Sn'.Cn' \leq Sn.Cn$. There are two subcases.
    - —Case $Sn.Cn \leq Sn'.Cn'$. By STRUCTOK $\overline{Sn} \vdash$ (`<abstract> class` $\overline{\alpha_4}\ Cn \ldots$) OK in Sn, so by CLASSOK $\overline{Sn} \vdash Sn.Cn$ ITCTransUses. Since $Sn.Cn \leq Sn'.Cn'$, by Lemma 37 we have $Sn' \in \overline{Sn}$, which is a contradiction.
    - —Case $Sn'.Cn' \leq Sn.Cn$. Symmetric to the above case.

    So we have shown that there does not exist Pat_0 such that $Pat \cap Pat' = Pat_0$. Then by the contrapositive of Lemma 27, $Pat \not\leq Pat'$, so also $Pat \not\geq Pat'$. Then by PAIRAMB we have $\text{dom}(SigT) \vdash (Pat, Pat')$ unambiguous. □

A.2 Type Preservation

The type preservation theorem is straightforward except for the case when E is a function application. That case follows easily from Lemmas 9 and 10.

THEOREM 2 (TYPE PRESERVATION). *If $\vdash E : \tau$ and $E \longrightarrow E'$ then $\vdash E' : \tau'$, for some τ' such that $\tau' \leq \tau$.*

PROOF. By (strong) induction on the depth of the derivation of $E \longrightarrow E'$. Case analysis of the last rule used in the derivation.

- Case E-NEW. Then E has the form $Ct(\overline{E})$ and E' has the form $Ct\{\overline{V_0} = \overline{E_0}\}$ and $Ct = (\overline{\tau}\ C)$ and $\text{concrete}(C)$ and $\text{rep}(Ct(\overline{E})) = \{\overline{V_0} = \overline{E_0}\}$. Since $\vdash E : \tau$, by T-NEW we have $\tau = Ct$ and $\{\}; \bullet \vdash Ct(\overline{E})$ OK. Then by T-CONSTR we have $\bullet \vdash Ct$ OK. Therefore by Lemmas 33 and 34 there exists $\overline{\tau_0}$ such that $\text{repType}(Ct) = \{\overline{V_0} : \overline{\tau_0}\}$, and by Lemma 35 we have $\vdash \overline{E_0} : \overline{\tau_0'}$, where $\overline{\tau_0'} \leq \overline{\tau_0}$. Then by T-REP we have $\vdash Ct\{\overline{V_0} = \overline{E_0}\} : Ct$, and by SUBTREF we have $Ct \leq Ct$.
- Case E-REP. Then E has the form $Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0, \overline{V_1} = \overline{E_1}\}$ and E' has the form $Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0', \overline{V_1} = \overline{E_1}\}$ and $E_0 \longrightarrow E_0'$. Since $\vdash E : \tau$, by T-REP we have $\tau = Ct = (\overline{\tau}\ C)$ and $\text{concrete}(C)$ and $\bullet \vdash Ct$ OK and $\text{repType}(Ct) = \{\overline{V_0} : \overline{\tau_0}, V_0 : \tau_0, \overline{V_1} : \overline{\tau_1}\}$ and $\vdash \overline{v_0} : \overline{\tau_0'}$ and $\overline{\tau_0'} \leq \overline{\tau_0}$ and $\vdash E_0 : \tau_0'$ and $\tau_0' \leq \tau_0$ and $\vdash \overline{E_1} : \overline{\tau_1'}$ and $\overline{\tau_1'} \leq \overline{\tau_1}$. By induction we have $\vdash E_0' : \tau_0''$, for some τ_0'' such that $\tau_0'' \leq \tau_0'$. Therefore by SUBTTRANS $\tau_0'' \leq \tau_0$. Then by T-REP $\vdash Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0', \overline{V_1} = \overline{E_1}\} : Ct$, and by SUBTREF $Ct \leq Ct$.
- Case E-TUP. Then E has the form $(v_1, \ldots, v_{i-1}, E_i, \ldots, E_k)$ and E' has the form $(v_1, \ldots, v_{i-1}, E_i', E_{i+1}, \ldots, E_k)$ and $E_i \longrightarrow E_i'$, where $1 \leq i \leq k$. Since $\vdash E : \tau$, by T-TUP τ has the form $\tau_1 * \cdots * \tau_k$ and $\vdash v_j : \tau_j$ for all $1 \leq j < i$

and $\vdash E_j : \tau_j$ for all $i \leq j \leq k$. By induction we have $\vdash E'_i : \tau'_i$ for some τ'_i such that $\tau'_i \leq \tau_i$. Then by T-TUP we have $\vdash (v_1, \ldots, v_{i-1}, E'_i, E_{i+1}, \ldots, E_k) : \tau_1 * \cdots * \tau_{i-1} * \tau'_i * \tau_{i+1} * \cdots * \tau_k$. By SUBTREF we have $\tau_j \leq \tau_j$ for all $1 \leq j \leq k$, so by SUBTTUP we have $\tau_1 * \cdots * \tau_{i-1} * \tau'_i * \tau_{i+1} * \cdots * \tau_k \leq \tau_1 * \cdots * \tau_k$.

- Case E-APP1. Then E has the form $E_1\ E_2$ and E' has the form $E'_1\ E_2$ and $E_1 \longrightarrow E'_1$. Since $\vdash E : \tau$, by (T-App) we have $\vdash E_1 : \tau_2 \to \tau$ and $\vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. By induction $\vdash E'_1 : \tau'$, for some τ' such that $\tau' \leq \tau_2 \to \tau$. By Lemma 17 τ' has the form $\tau''_2 \to \tau''$, where $\tau_2 \leq \tau''_2$ and $\tau'' \leq \tau$. Therefore by SUBTTRANS we have $\tau'_2 \leq \tau''_2$, so by T-APP we have $\vdash E'_1\ E_2 : \tau''$, where $\tau'' \leq \tau$.
- Case E-APP2. Then E has the form $v_1\ E_2$ and E' has the form $v_1\ E'_2$ and $E_2 \longrightarrow E'_2$. Since $\vdash E : \tau$, by T-APP we have $\vdash v_1 : \tau_2 \to \tau$ and $\vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. By induction $\vdash E'_2 : \tau''_2$, for some τ''_2 such that $\tau''_2 \leq \tau'_2$. By SUBTTRANS we have $\tau''_2 \leq \tau_2$, so by T-APP we have $\vdash v_1\ E'_2 : \tau$ and by SUBTREF we have $\tau \leq \tau$.
- Case E-APPRED. Then $E = (\overline{\tau}\ F)\ v$ and $E' = [\overline{I_0} \mapsto \overline{v_0}]E_0$ and most-specific-case-for$((\overline{\tau}\ F), v) = (\{(\overline{I_0}, \overline{v_0})\}, E_0)$. Since $\vdash E : \tau$, by T-APP we have $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$ and $\vdash v : \tau'_2$ and $\tau'_2 \leq \tau_2$. Then by T-FUN we have and $F = Sn.Fn$ and $\tau_2 \to \tau = [\overline{\alpha} \mapsto \overline{\tau}](\hat{M}t \to \tau_0)$ and $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in SigT(Sn)$ and $\bullet \vdash \overline{\tau}$ OK. By LOOKUP we have $E_0 = [\overline{\alpha_0} \mapsto \overline{\tau}]E'_0$ and $(\texttt{extend fun}_{Mn}\ \overline{\alpha_0}\ F\ Pat = E'_0) \in ST(Sn')$ and $\text{match}(v, Pat) = \{(\overline{I_0}, \overline{v_0})\}$. Then by CASEOK we have $\text{matchType}([\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{M}t, Pat) = (\Gamma, \tau'')$ and $\Gamma; \overline{\alpha_0} \vdash E'_0 : \tau'_0$ and $\tau'_0 \leq [\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$.

  By Lemma 24 we have $\text{matchType}([\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{M}t, Pat) = ([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'')$. By FUNOK we have $\overline{\alpha} \vdash \hat{M}t$ OK, so by Lemma 11 all type variables in $\hat{M}t$ are in $\overline{\alpha}$. Therefore $[\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{M}t$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t = \tau_2$, so we have $\text{matchType}(\tau_2, Pat) = ([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'')$. Then by Lemma 10 we have $\tau'_2 \leq [\overline{\alpha_0} \mapsto \overline{\tau}]\tau''$ and $\text{dom}([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma) = \text{dom}(\{(\overline{I_0}, \overline{v_0})\})$ and for each $(I_x, \tau_x) \in [\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma$, there exists $(I_x, v_x) \in \{(\overline{I_0}, \overline{v_0})\}$ such that $\vdash v_x : \tau'_x$, where $\tau'_x \leq \tau_x$.

  By Lemma 22 we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma; \bullet \vdash [\overline{\alpha_0} \mapsto \overline{\tau}]E'_0 : [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'_0$. Then by Lemma 9 $\vdash [\overline{I_0} \mapsto \overline{v_0}][\overline{\alpha_0} \mapsto \overline{\tau}]E'_0 : \tau_{sub}$ and $\tau_{sub} \leq [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'_0$. By Lemma 21 we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\tau'_0 \leq [\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$. By FUNOK we have $\overline{\alpha} \vdash \tau_0$ OK, so by Lemma 11 all type variables in τ_0 are in $\overline{\alpha}$. Therefore $[\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}]\tau_0 = \tau$, so we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\tau'_0 \leq \tau$. Then by SUBTTRANS we have $\tau_{sub} \leq \tau$. Therefore we have shown $\vdash E' : \tau_{sub}$ and $\tau_{sub} \leq \tau$. □

This is a standard lemma showing that type-correct substitution preserves the well-typedness of an expression.

LEMMA 9 (SUBSTITUTION). *If* $\Gamma; \overline{\alpha_0} \vdash E : \tau$ *and* $\Gamma = \{(\overline{I_0}, \overline{\tau_0})\}$ *and* $\Gamma_0; \overline{\alpha_0} \vdash \overline{E_0} : \overline{\tau_0}'$ *and* $\overline{\tau'_0} \leq \overline{\tau_0}$, *then* $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau'$, *for some* τ' *such that* $\tau' \leq \tau$.

PROOF. By (strong) induction on the depth of the derivation of $\Gamma; \overline{\alpha_0} \vdash E : \tau$. Case analysis of the last rule used in the derivation.

- Case T-ID. Then $E = I$ and $(I, \tau) \in \Gamma$, so $I = I_j$ and $\tau = \tau_j$ for some $1 \leq j \leq k$, where $\overline{I_0} = I_1, \ldots, I_k$ and $\overline{\tau_0} = \tau_1, \ldots, \tau_k$ and $\overline{E_0} = E_1, \ldots, E_k$. Therefore $[\overline{I_0} \mapsto \overline{E_0}]E = E_j$. Since we're given that $\Gamma_0; \overline{\alpha_0} \vdash E_j : \tau'_j$ and $\tau'_j \leq \tau_j$, the result is shown.

- Case T-NEW. Then $E = Ct(\overline{E})$ and $\tau = Ct$ and $\Gamma; \overline{\alpha_0} \vdash Ct(\overline{E})$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete($Sn.Cn$). Then by T-CONSTR we have $\overline{\alpha_0} \vdash Ct$ OK and (<abstract> class $\overline{\alpha_1}\ Cn(\overline{I} : \overline{\tau}) \ldots) \in SigT(Sn)$ and $\Gamma; \overline{\alpha_0} \vdash \overline{E} : \overline{\tau'}$ and $\overline{\tau'} \leq [\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau}$. Since $[\overline{I_0} \mapsto \overline{E_0}]Ct = Ct$ and $[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn = Sn.Cn$, we have $\overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]Ct$ OK and concrete($[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn$). By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]\overline{E} : \overline{\tau''}$ and $\overline{\tau''} \leq \overline{\tau'}$. Then by SUBTTRANS we have $\overline{\tau''} \leq [\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau}$. Therefore by T-CONSTR we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E$ OK, so by T-NEW we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$. By SUBTREF we have $\tau \leq \tau$, so the result is shown.
- Case T-REP. Then $E = Ct\{\overline{V} = \overline{E}\}$ and $\tau = Ct$ and $\overline{\alpha_0} \vdash Ct$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete($Sn.Cn$) and repType(Ct) $= \{\overline{V} : \overline{\tau}\}$ and $\Gamma; \overline{\alpha_0} \vdash \overline{E} : \overline{\tau'}$ and $\overline{\tau'} \leq \overline{\tau}$. Since $[\overline{I_0} \mapsto \overline{E_0}]Ct = Ct$ and $[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn = Sn.Cn$, we have $\overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]Ct$ OK and concrete($[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn$) and repType($[\overline{I_0} \mapsto \overline{E_0}]Ct$) $= \{\overline{V} : \overline{\tau}\}$. By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]\overline{E} : \overline{\tau''}$ and $\overline{\tau''} \leq \overline{\tau'}$. Then by SUBTTRANS we have $\overline{\tau''} \leq \overline{\tau}$, so by T-Rep we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$. By SUBTREF we have $\tau \leq \tau$, so the result is shown.
- Case T-FUN. Then since Γ is not used in T-FUN and $\Gamma; \overline{\alpha_0} \vdash E : \tau$, also $\Gamma_0; \overline{\alpha_0} \vdash E : \tau$. Further, we have $E = Fv$, so $[\overline{I_0} \mapsto \overline{E_0}]E = E$. Therefore $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$, and by SUBTREF $\tau \leq \tau$, so the result is shown.
- Case T-TUP. Then $E = (E_1, \ldots, E_k)$ and $\tau = \tau_1 * \cdots * \tau_k$ and for all $1 \leq j \leq k$ we have $\Gamma; \overline{\alpha_0} \vdash E_j : \tau_j$. Then by induction, for all $1 \leq j \leq k$ we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_j : \tau'_j$ and $\tau'_j \leq \tau_j$. Then by T-TUP we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}](E_1, \ldots, E_k) : \tau'_1 * \cdots * \tau'_k$. Finally, by SUBTTUP we have $\tau'_1 * \cdots * \tau'_k \leq \tau_1 * \cdots * \tau_k$.
- Case T-APP. Then $E = E_1\ E_2$ and $\Gamma; \overline{\alpha_0} \vdash E_1 : \tau_2 \to \tau$ and $\Gamma; \overline{\alpha_0} \vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_1 : \tau_0$ and $\tau_0 \leq \tau_2 \to \tau$. Also by induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_2 : \tau''_2$ and $\tau''_2 \leq \tau'_2$. Then by SUBTTRANS we have $\tau''_2 \leq \tau_2$. By Lemma 17 τ_0 has the form $\tau_{arg} \to \tau_{res}$, where $\tau_2 \leq \tau_{arg}$ and $\tau_{res} \leq \tau$. Therefore by SUBTTRANS we have $\tau''_2 \leq \tau_{arg}$. Therefore by T-APP we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}](E'_1\ E'_2) : \tau_{res}$. We saw above that $\tau_{res} \leq \tau$, so the result is shown. □

This lemma relates the results of pattern matching to its static approximation. In particular, the type of any value matching a pattern is a subtype of the pattern's type, and similarly for values bound to identifiers in the pattern. The proof is straightforward.

LEMMA 10. *If* $\vdash v : \tau''$ *and* $\tau'' \leq \tau$ *and* $match(v, Pat) = \rho$ *and* $matchType(\tau, Pat) = (\Gamma, \tau')$, *then (1)* $\tau'' \leq \tau'$; *and (2)* $dom(\Gamma) = dom(\rho)$ *and (3) for each* $(I_0, \tau_0) \in \Gamma$, *there exists* $(I_0, v_0) \in \rho$ *such that* $\vdash v_0 : \tau'_0$, *for some* τ'_0 *such that* $\tau'_0 \leq \tau_0$.

PROOF. By (strong) induction on the length of the derivation of match $(v, Pat) = \rho$. Case analysis of the last rule used in the derivation:

- Case E-MATCHWILD. Then Pat has the form _ and $\rho = \{\}$. By T-MATCHWILD we have $\Gamma = \{\}$ and $\tau' = \tau$. Therefore, conditions 1 and 2 are shown, and condition 3 holds vacuously.

- Case E-MatchBind. Then Pat has the form I as Pat' and $\rho = \rho' \cup \{(I, v)\}$ and $\text{match}(v, Pat') = \rho'$. By T-MatchBind we have $\Gamma = \Gamma' \cup \{(I, \tau')\}$ and $\text{matchType}(\tau, Pat') = (\Gamma', \tau')$. By induction we have $\tau'' \leq \tau'$ and $\text{dom}(\Gamma') = \text{dom}(\rho')$ and for each $(I_0, \tau_0) \in \Gamma'$, there exists $(I_0, v_0) \in \rho'$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore, we have $\tau'' \leq \tau'$ and $\text{dom}(\Gamma' \cup \{(I, \tau')\}) = \text{dom}(\rho' \cup \{(I, v)\})$ and for each $(I_0, \tau_0) \in \Gamma' \cup \{(I, \tau')\}$, there exists $(I_0, v_0) \in \rho' \cup \{(I, v)\}$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$.
- Case E-MatchTup. Then $v = (v_1, \ldots, v_k)$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and $\rho = \rho_1 \cup \cdots \cup \rho_k$ and for all $1 \leq i \leq k$ we have $\text{match}(v_i, Pat_i) = \rho_i$. By T-MatchTup we have $\tau = \tau_1 * \cdots * \tau_k$ and $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_k$ and $\tau' = \tau_1' \cdots * \tau_k'$ and for all $1 \leq i \leq k$ we have $\text{match}(\tau_i, Pat_i) = (\Gamma_i, \tau_i')$.

  Since $\vdash v : \tau''$, by T-Tup we have that $\tau'' = \tau_1'' * \cdots * \tau_k''$ and for all $1 \leq i \leq k$ we have $\vdash v_i : \tau_i''$. Since we're given that $\tau'' \leq \tau$, by Lemma 19 we have $\tau_i'' \leq \tau_i$ for all $1 \leq i \leq k$. Then by induction, for all $1 \leq i \leq k$ we have $\tau_i'' \leq \tau_i'$. Then by SubTTup we have $\tau_1'' * \cdots * \tau_k'' \leq \tau_1' * \ldots * \tau_k'$, proving condition 1. Also by induction, for all $1 \leq i \leq k$ we have $\text{dom}(\Gamma_i) = \text{dom}(\rho_i)$ and for each $(I_0, \tau_0) \in \Gamma_i$, there exists $(I_0, v_0) \in \rho_i$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore conditions 2 and 3 follow.
- Case E-MatchClass. Then $v = ((\overline{\tau}\ C)\{\overline{V_1} = \overline{v_1}, \overline{V_2} = \overline{v_2}\})$ and Pat has the form $(C'\{\overline{V_1} = \overline{Pat_1})$ and $C \leq C'$ and $\rho = \bigcup \overline{\rho_1}$ and $\text{match}(\overline{v_1}, \overline{Pat_1}) = \overline{\rho_1}$. By T-MatchClass we have $\tau = (\overline{\tau'}\ C'')$ and $\tau' = (\overline{\tau'}\ C')$ and $\Gamma = \bigcup \overline{\Gamma_1}$ and $C' \leq C''$ and $\text{repType}(\overline{\tau'}\ C') = \{\overline{V_1} : \overline{\tau_1}\}$ and $\text{matchType}(\overline{\tau_1}, \overline{Pat_1}) = (\overline{\Gamma_1}, \overline{\tau_1'})$.

  Since $\vdash v : \tau''$ and $v = ((\overline{\tau}\ C)\{\overline{V_1} = \overline{v_1}, \overline{V_2} = \overline{v_2}\})$, by T-Rep we have that $\tau'' = (\overline{\tau}\ C)$ and $\bullet \vdash (\overline{\tau}\ C)$ OK and and $\text{repType}(\overline{\tau}\ C) = \{\overline{V_1} : \overline{\tau_1''}, \overline{V_2} : \overline{\tau_2''}\}$ and $\vdash \overline{v_1} : \overline{\tau_1'''}$ and $\overline{\tau_1'''} \leq \overline{\tau_1''}$. Since $\tau'' \leq \tau$, we have $(\overline{\tau}\ C) \leq (\overline{\tau'}\ C'')$, so by Lemma 14 we have $\overline{\tau} = \overline{\tau'}$. Since $C \leq C'$ and $\bullet \vdash (\overline{\tau}\ C)$ OK, by Lemma 16 we have $(\overline{\tau}\ C) \leq (\overline{\tau}\ C')$, and since $\overline{\tau} = \overline{\tau'}$, condition 1 is shown. By Lemma 36 we have $\overline{\tau_1''} = \overline{\tau_1}$. Therefore $\vdash \overline{v_1} : \overline{\tau_1'''}$ and $\overline{\tau_1'''} \leq \overline{\tau_1}$ and $\text{match}(\overline{v_1}, \overline{Pat_1}) = \overline{\rho_1}$ and $\text{matchType}(\overline{\tau_1}, \overline{Pat_1}) = (\overline{\Gamma_1}, \overline{\tau_1'})$, so by induction it follows that $\text{dom}(\bigcup \overline{\Gamma_1}) = \text{dom}(\bigcup \overline{\rho_1})$ and for each $(I_0, \tau_0) \in \bigcup \overline{\Gamma_1}$, there exists $(I_0, v_0) \in \bigcup \overline{\rho_1}$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore, conditions 2 and 3 are shown. □

A.3 Basic Lemmas

These lemmas state some simple and intuitive properties about Eml's semantics. The proofs are all straightforward and can be found in the first author's dissertation [Millstein 2003].

A.3.1 *Type Well-formedness*

Lemma 11. *If $\overline{\alpha} \vdash \tau$ OK, then all type variables in τ are in $\overline{\alpha}$.*

A.3.2 *Subclassing and Subtyping*

Lemma 12. *If $\tau \leq (\overline{\tau}\ C)$, then τ has the form $(\overline{\tau_1}\ C')$.*

Lemma 13. *If $(\overline{\tau}\ C) \leq \tau$, then τ has the form $(\overline{\tau_1}\ C')$.*

Lemma 14. *If $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$, then $\overline{\tau} = \overline{\tau_1}$.*

LEMMA 15. *If* $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$ *then* $C \leq C'$.

LEMMA 16. *If* $C_1 \leq C_2$ *and* $\overline{\alpha} \vdash (\overline{\tau}\ C_1)$ *OK then (1)* $(\overline{\tau}\ C_1) \leq (\overline{\tau}\ C_2)$*; and (2)* $\overline{\alpha} \vdash (\overline{\tau}\ C_2)$ *OK.*

LEMMA 17. *If* $\tau \leq \tau_1 \rightarrow \tau_2$*, then* τ *has the form* $\tau_1' \rightarrow \tau_2'$*, where* $\tau_1 \leq \tau_1'$ *and* $\tau_2' \leq \tau_2$.

LEMMA 18. *If* $\tau_1 \rightarrow \tau_2 \leq \tau$*, then* τ *has the form* $\tau_1' \rightarrow \tau_2'$.

LEMMA 19. *If* $\tau \leq \tau_1 * \cdots * \tau_k$*, then* τ *has the form* $\tau_1' * \cdots * \tau_k'$*, where for all* $1 \leq i \leq k$ *we have* $\tau_i' \leq \tau_i$.

LEMMA 20. *If* $\tau_1 * \cdots * \tau_k \leq \tau$*, then* τ *has the form* $\tau_1' * \cdots * \tau_k'$*, where for all* $1 \leq i \leq k$ *we have* $\tau_i \leq \tau_i'$.

A.3.3 *Type Substitution*

LEMMA 21. *If* $\tau \leq \tau'$ *and* $|\overline{\alpha}| = |\overline{\tau}|$*, then* $[\overline{\alpha} \mapsto \overline{\tau}]\tau \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau'$.

LEMMA 22. *If* $\Gamma; \overline{\alpha} \vdash E : \tau$ *and* $|\overline{\alpha}| = |\overline{\tau}|$ *and* $\overline{\alpha_0} \vdash \overline{\tau}$ *OK, then* $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E : [\overline{\alpha} \mapsto \overline{\tau}]\tau$.

LEMMA 23. *If* $repType(Ct) = \{\overline{V} : \overline{\tau}\}$ *and* $|\overline{\alpha}| = |\overline{\tau}|$*, then* $repType([\overline{\alpha} \mapsto \overline{\tau}]Ct) = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau}\}$.

LEMMA 24. *If* $matchType(\tau, Pat) = (\Gamma, \tau')$ *and* $|\overline{\alpha}| = |\overline{\tau}|$*, then* $matchType([\overline{\alpha} \mapsto \overline{\tau}]\tau, Pat) = ([\overline{\alpha} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha} \mapsto \overline{\tau}]\tau')$.

A.3.4 *Pattern Matching, Specificity, and Intersection*

LEMMA 25. *If* $match(v, Pat) = \rho$ *and* $Pat \leq Pat'$*, then there exists* ρ' *such that* $match(v, Pat') = \rho'$.

LEMMA 26. *If* $Pat \leq Pat'$ *and* $Pat' \leq Pat''$ *then* $Pat \leq Pat''$.

LEMMA 27. *If* $Pat \leq Pat'$ *then there exists some* Pat_0 *such that* $Pat \cap Pat' = Pat_0$.

LEMMA 28. *If* $Pat \cap Pat' = Pat_0$ *then* $Pat_0 \leq Pat$ *and* $Pat_0 \leq Pat'$.

LEMMA 29. *If* $owner(Mt, Pat') = C'$ *and* $owner(Mt, Pat'') = C''$ *and* $Pat' \cap Pat'' = Pat$*, then either* $C' \leq C''$ *or* $C'' \leq C'$.

LEMMA 30. *If* $\vdash v : \tau$ *and* $match(v, Pat') = \rho'$ *and* $match(v, Pat'') = \rho''$ *and* $matchType(\tau', Pat') = (\Gamma', \tau_0')$ *and* $matchType(\tau'', Pat'') = (\Gamma'', \tau_0'')$*, then there exists some* Pat *such that* $Pat' \cap Pat'' = Pat$.

LEMMA 31. *If* $match(v, Pat') = \rho'$ *and* $match(v, Pat'') = \rho''$ *and* $Pat' \cap Pat'' = Pat$*, then there exists some* ρ *such that* $match(v, Pat) = \rho$.

A.3.5 *Representations and Representation Types*

LEMMA 32. *If* $\Gamma; \overline{\alpha} \vdash Ct(\overline{E_0})$ *OK then there exist* $\overline{V}$ *and* $\overline{E}$ *such that* $rep(Ct(\overline{E_0})) = \{\overline{V} = \overline{E}\}$.

LEMMA 33. *If* $\overline{\alpha} \vdash Ct$ *OK then there exist* $\overline{V_0}$ *and* $\overline{\tau_0}$ *such that repType(Ct)* $= \{\overline{V_0} : \overline{\tau_0}\}$ *and* $\overline{\alpha} \vdash \overline{\tau_0}$ *OK.*

LEMMA 34. *If rep(Ct*$(\overline{E})$*)* $= \{\overline{V_1} = \overline{E_1}\}$ *and repType(Ct)* $= \{\overline{V_2} : \overline{\tau_2}\}$ *then* $\overline{V_1} = \overline{V_2}$.

LEMMA 35. *If* $\Gamma_0; \overline{\alpha_0} \vdash Ct(\overline{E})$ *OK and rep(Ct*$(\overline{E})$*)* $= \{\overline{V_0} = \overline{E_0}\}$ *and repType(Ct)* $= \{\overline{V_0} : \overline{\tau_0}\}$*, then* $\Gamma_0; \overline{\alpha_0} \vdash \overline{E_0} : \overline{\tau_0'}$*, for some* $\overline{\tau_0'}$ *such that* $\overline{\tau_0'} \leq \overline{\tau_0}$.

LEMMA 36. *If* $\bullet \vdash Ct$ *OK and* $Ct \leq Ct'$ *then repType(Ct) has the form* $\{\overline{V_1} : \overline{\tau_1}, \overline{V_2} : \overline{\tau_2}\}$*, where repType(Ct*$'$*)* $= \{\overline{V_1} : \overline{\tau_1}\}$.

A.3.6 *Module Dependency Relation*

LEMMA 37. *If* $\overline{Sn} \vdash C$ *ITCTransUses and* $C \leq C'$*, then* $\overline{Sn} \vdash C'$ *ITCTransUses.*

ACKNOWLEDGMENTS

Thanks to Jonathan Aldrich, Sorin Lerner, and Vass Litvinov for helpful comments on the paper.

REFERENCES

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2002. A formal framework for Java separate compilation. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 2374, Malaga, Spain (June). Springer-Verlag.

ANCONA, D. AND ZUCCA, E. 2001. True modules for Java-like languages. In J. L. Knudsen, Ed. *ECOOP 2001–Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 2072. Springer.

ANCONA, D. AND ZUCCA, E. 2002. A calculus of module systems. *J. Funct. Prog. 12*, 2, 91–132 (March).

ARNOLD, K., GOSLING, J., AND HOLMES, D. 2000. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition.

BONNIOT, D. 2002. 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).

BOURDONCLE, F. AND MERZ, S. 1997. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 302–315, Paris, France, 15–17 (Jan).

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, 303–311.

CARDELLI, L. 1988. A semantics of multiple inheritance. *Information and Computation 76*, 2/3, 138–164 (Feb).

CARDELLI, L. 1997. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, 15–17 (Jan.).

CARDELLI, L. and WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4, 471–522 (Dec.).

CHAMBERS, C. AND CHEN, W. 1999. Efficient multiple and predicate dispatching. In L. Meissner, Ed. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, 238–255, N.Y., Nov. 1–5. ACM Press.

CHAMBERS, C. AND LEAVENS, G. T. 1995. Typechecking and modules for multimethods. *ACM Trans. Prog. Lang. Sys. 17*, 6, 805–843 (Nov.).

CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, 130–145 (Oct.).

COOK, W. R. 1991. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June*, LNCS 489, 151–178. Springer-Verlag, New York, NY.

DROSSOPOULOU, S., EISENBACH, S., and WRAGG, D. 1999. A fragment calculus—towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*, 147–156.

DUGGAN, D. AND SOURELIS, C. 1996. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, 262–273, Philadelphia, Pennsylvania (May).

DUGGAN, D. AND TECHAUBOL, C.-C. 2001. Modular mixin-based inheritance for application frameworks. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, 223–240. ACM Press.

ERNST, M., KAPLAN, C., AND CHAMBERS, C. 1998. Predicate dispatching: A unified theory of dispatch. In E. Jul, Ed. *ECOOP '98–Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 1445, 186–211. Springer.

FINDLER, R. B. AND FLATT, M. 1998. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, 34(1) of *ACM SIGPLAN Notices*, 94–104. ACM (June).

FISHER, K. AND REPPY, J. 1999. The design of a class mechanism for MOBY. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, 37–49, Atlanta, Georgia (May 1–4).

FISHER, K. AND REPPY, J. 2000. Extending Moby with inheritance-based subtyping. In *14th European Conference on Object-Oriented Programming*, LNCS 1850, 83–107 (June).

FLATT, M. AND FELLEISEN, M. 1998. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, 236–248, Montreal, Canada (17–19 June).

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, 171–183, New York, NY.

FUH, Y.-C. C. AND MISHRA, P. 1990. Type inference with subtypes. *Theoretical Computer Science 73*, 2, 155–175 (June).

GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts.

GARRIGUE, J. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass.

HIRSCHOWITZ, T. AND LEROY, X. 2002. Mixin modules in a call-by-value setting. In D. Le Métayer, Ed, *Programming Languages and Systems, ESOP'2002*, volume 2305 of Lecture Notes in Computer Science, 6–20. Springer-Verlag.

HOANG, M. AND MITCHELL, J. C. 1995. 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.*, 176–185, New York, NY (Jan.) ACM.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*, 3, 396–450, May.

KAHRS, S., SANNELLA, D., AND TARLECKI, A. 1997. The definition of extended ML: A gentle introduction. *Theoretical Computer Science 173*, 2, 445–484, 28(Feb.).

KRISHNAMURTHI, S., FELLEISEN, M., AND FRIEDMAN, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, Ed. *ECOOP'98–Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, Lecture Notes in Computer Science, vol. 1445, 91–113. Springer-Verlag (July).

MacQueen, D. 1984. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 198–207. ACM (Aug.).

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

Millstein, T. 2003. *Reconciling Software Extensibility with Modular Program Reasoning*. Ph.D. dissertation, Department of Computer Science & Engineering, University of Washington.

Millstein, T., Bleckner, C., and Chambers, C. 2002. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, volume 37(9) of *ACM SIGPLAN Notices*, 110–122, New York, NY (Sept.) ACM.

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

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

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

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

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

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

Pierce, B. C. and Turner, D. N. 2000. Local type inference. *ACM Trans. Prog. Lang. Syst. 22*, 1, 1–44 (Jan.).

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

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

Reynolds, J. C. 1978. 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*, 309–317. Springer-Verlag, New York, NY.

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

Wadler, P. 1990. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 7–359. North Holland (Apr.).

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115*, 1, 38–94, 15 (Nov.).

Zenger, M., and Odersky, M. 2001. Extensible algebraic datatypes with defaults. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*. ACM (September) 3–5.

Received August 2002; revised September 2003; accepted February 2004