Motivation for Extensible ML (EML)

the functional and OO styles have different benefits

- pattern matching versus dynamic dispatch
- the "expression problem"...

An Interpreter in a Functional Language

```
datatype Expr = IntLit of int
              | IfExpr of {test:Expr, then:Expr,
                           else:Expr}
              | ...

fun evalExpr(IntLit i, env) = ...
  | evalExpr(IfExpr ..., env) = ...
  | ...

fun typecheckExpr(IntLit i, tenv) = ...
  | typecheckExpr(IfExpr ..., tenv) = ...
  | ...
```

An Interpreter in a Functional Language (cont.)

- How easy is it to add new passes to the interpreter (e.g. optimize, pretty-print)?
  - write a function for the new pass
  - no need to rewrite existing code or clients
  - no need to retypecheck/recompile existing code or clients
- How easy is it to add new kinds of expressions to the interpreter (e.g. function call, let expression)?
  - must modify the expression datatype in place
  - must add a function case handling the new expressions to *every* existing pass (i.e. function)
  - must retypecheck/recompile existing code

An Interpreter in a OO Language

```
abstract class Expr {
  abstract Value evalExpr(Env env);
  abstract Type typecheckExpr(TypeEnv tenv);
}

class IntLit extends Expr {
  Value evalExpr(Env env) { ... }
  Type typecheckExpr(TypeEnv tenv) { ... }
}

class IfExpr extends Expr {
  Value evalExpr(Env env) { ... }
  Type typecheckExpr(TypeEnv tenv) { ... }
}
```

An Interpreter in a OO Language (cont.)

- How easy is it to add new passes to the interpreter (e.g. optimize, pretty-print)?
  - must modify the `Expr` class to add the new method
  - must potentially modify *every* subclass of `Expr`
  - must retypecheck/recompile all of these classes
- How easy is it to add new kinds of expressions to the interpreter (e.g. function call, let expression)?
  - write a new subclass of `Expr` (or of some existing subclass), with overriding methods for the various passes
  - no need to modify existing code or clients
  - no need to retypecheck/recompile existing code or clients

Possible Solution: Workarounds

functional languages:

- create a wrapper datatype that has the new kinds of expressions

OO languages:

- create new subclasses to contain the new methods
- visitor pattern

Possible Solution: Language Combination

put both the functional and OO language constructs in a single language

- OCaml adds classes, methods to ML
- Pizza adds datatypes, pattern matching to Java

- it is difficult to know *a priori* which benefits will be important for a particular abstraction
- may require both kinds of expressiveness

Possible Solution: Cecil (and others)

- like other OO langs, can add new subclasses (children) modularly to existing classes (objects)
- multimethods are naturally declared "outside" of classes, so can also modularly add new methods to existing classes

- requires the whole program to be available to perform typechecking

Extensible ML (EML)

- generalize ML-style datatypes to be hierarchical and extensible
  - **class** construct subsumes datatypes
- generalize ML-style functions to be extensible
  - new cases added as new data variants are defined
- unify ML-style pattern matching with OO-style dynamic dispatch
- retain modular typechecking

EML Example

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

Datatypes as Classes

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

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

EML Example

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

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

EML Example

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

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

  ... // addElem, isEmpty, etc.
end
```

Adding New Subclasses

```
structure HashSetMod = struct

  class HashSet extends SetMod.Set
     of {ht:(int,unit) hashtable}

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

  ... // extend addElem, etc; inherit any others
end
```

Adding New Operations

```
  structure UnionMod = struct

    fun union:(Set * Set) → Set
    extend fun union (s1,s2) = …
    extend fun union (ListSet {es=e1},
                      ListSet {es=e2})
      = ListSet {es=merge(sort(e1),sort(e2))}
  end
```

Another Subclass

```
structure SortedListSetMod = struct

  class SListSet extends ListSet of {}
  extend fun union (SListSet {es=e1},
                    SListSet {es=e2})
    = SListSet {es=merge(e1,e2)}
  ... // extend addElem, etc

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

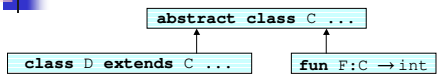
Modular Typechecking

- typecheck each module M given only the module *interfaces* I that M depends upon
  - M refers to a name bound in interface I
  - M depends upon I′ and I′ refers to a name bound in I
- an EML interface is a signature; a structure's *principal signature* contains
  - all class declarations
  - all function declarations
  - all function case headers, without the bodies

Implementation-side Typechecking (ITC)

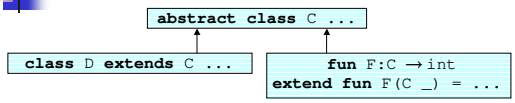
- functional languages
  - ensure each function is exhaustive and non-redundant
  - checked modularly: each function has access to all cases and type-correct arguments
- OO languages
  - ensure each class has a most-specific method for each supported operation
  - checked modularly: each class has access to the headers of all applicable methods
- EML
  - ensure each function is exhaustive and unambiguous
  - checked modularly?

New Functions on Existing Classes



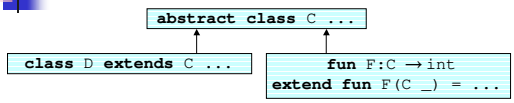
- problem: `F` may be incomplete for `D`

New Functions on Existing Classes

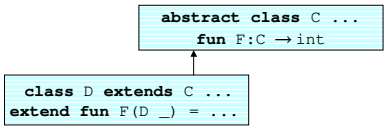


- problem: `F` may be incomplete for `D`
- solution: *external* `F` must contain a default case

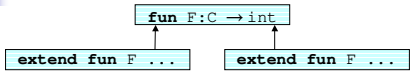
New Functions on Existing Classes



- problem: `F` may be incomplete for `D`
- solution: *external* `F` must contain a default case

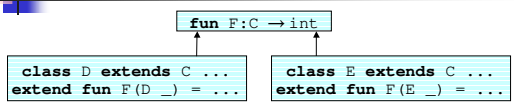


Ambiguous Function Cases



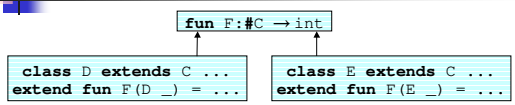
- problem: ambiguous function cases may not be accessible to each other modularly

Ambiguous Function Cases



- problem: ambiguous function cases may not be accessible to each other modularly
- OO idiom: add a new subclass of `C` and override some methods

Ambiguous Function Cases



- problem: ambiguous function cases may not be accessible to each other modularly
- OO idiom: add a new subclass of `C` and override some methods
- solution: add a new subclass of `C` and add cases to functions that `C` owns

Handling Advanced Features of ML

- parametric polymorphism
  - EML has a form consistent with ML
  - orthogonal to the issue of modular typechecking
- signature ascription
  - can hide EML declarations, instance variables
  - can seal a class or function
- functors
  - allow a form of mixin-style inheritance
  - more work needed to make EML functors practical