Lecture:
EML and Multimethods

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 9, 2004

1 Safety in EML

Last time we discussed two kinds of errors that an ill-formed EML program might produce at runtime. We'll refer to them as:

1. *Message not found*: a method call ('message') made by the program 'can't find a match,' i.e. can't find a method implementation corresponding to the arguments given.

2. *Message ambiguous*: a method call made by the program can't find a *unique* match, i.e. some method implementation cases match, but there is no most specific case.

We want to pick restrictions we can put on programs at compile-time so that these errors cannot occur at run-time. In the previous lecture, we simply ran a global analysis to see by brute force that every possible set of arguments that we might call a method with in fact determined a unique case. In a large program with a lot of classes and methods and method cases, this could be very expensive.

So what we'd like to do — and this is what makes EML interesting as compared to a naïve multimethod language — is do some compile-time checks to prevent these errors in a *modular* way. That is, we check the validity of a program one unit at a time. The unit of checking is going to be exactly the `module` $\cdots$ `end` blocks that we introduced already.

What it means to perform a 'modular' check on a module is that we *only depend on* which classes, methods, and method extentions have been declared in modules that the current module *statically depends* on, in the

following sense: we say the *interface* of a module is the set of classes, methods, and method exentions it declares, ignoring the actual implementation of the method extentions. A module M is then said to statically depend on another module M' if M's interface mentions a name (i.e. class name or method name) defined in the interface of M'. Also, if M_1 depends on M_2 which depends on $\cdots$ which depends on M_n, then we say that M_1 depends on M_n.

At the end of the day, when we check a module, we need only consider the current module and the interface of all the modules it depends on. This is similar in spirit to the encapsulation that Java and ML provide: you depend on a class's interface, or a structure's signature, but not the class's or the structure's implementation.

1.1 Completeness of Methods

We'll deal with *Message not found* first. Think about how languages like Java and ML deal with this issue.

In ML, it does appear at run-time as the error message `uncaught exception nonexhaustive match failure`, but the compiler also produces a warning message, `Warning: match nonexhaustive` when a function with missing cases is written. How the compiler knows that cases are actually missing involves the fact that, in an ML program, all of the cases of a function must appear as part of the function's definition. This fact means that the inherently local analysis the compiler does — that is, look at all of the cases given in the function definition and see if they're sufficient to cover any possible input — is correct.

In Java, the compiler can prevent a method from being invoked on objects that don't know how to implement it, because among other reasons, it knows what type the object has at run-time. If I try to downcast an `Object` to a `FloorWaxer` and call the method `waxFloor()` on it, the exception is raised at the downcast, not at the method call. Moreover, the compiler guarantees that any subclass of `FloorWaxer` must either inherit an implementation of or give its own implementation of all methods, e.g. `waxFloor()`. Because of this, if we ever have an expression with type `FloorWaxer`, we know it is safe to invoke `waxFloor()` on it, because it is statically guaranteed to be of some class that implements that method.

In this simplified version EML we will take a cue from Java and treat the first argument to each EML method as somewhat special.[1] This special

[1]The real EML allows the programmer to designate any of a tuple of arguments as this

treatment in no way changes our existing dispatch method: dispatch is still simultaneous and symmetric across all method arguments. We treat the first argument specially only for purposes of the compile-time restrictions we're about to introduce.

To prevent method calls from producing 'message not found' errors, informally we say the following: *every method must have at least as many implementations as an ML function or a Java method*. It may also have more.

The way we decide whether a method is 'more ML-like' or 'more Java-like' is based on the the first argument. If our code looks like

```
module
 class C of ...
 method m(C,D) : bool
 ...
end
```

then the method `m` occurs in the same module as the declaration of the class of its first argument. We call such methods **internal** methods. Since declaration of new methods in Java always occurs in the same file as the declaration of the class of their (implicit) first argument (i.e. `this`) we will treat internal methods similarly to OO-style methods.

> **Restriction 1**: *Internal Methods Require Local Defaults*. Suppose there is an internal method m with arguments $C, D_1, \ldots, D_n$. If C is abstract, then for any concrete subclass C' of C, there must be a *local default case for m and C'*, that is, a case `extend m(`$C', D_1, \ldots, D_n$`) = e` in the same module as the declaration of C'.

This restriction is checked whenever a concrete subclass of another class is declared. When this happens, we look to which methods are declared whose first argument is one of our superclasses. If we cannot inherit an implementation from a superclass, then we must write a local default case.

The other restriction applies to methods that are *not* declared in the same module as their first argument's class. These methods are called **external**. For these we need to impose a stronger condition.

---

singled-out 'owner' position.

Restriction 2: *External Methods Require Global Defaults*. Suppose there is an external method m with arguments $C, D_1, \ldots, D_n$. Then in the same module as m there must be a *global default case* for m, that is, a case `extend m(`$C, D_1, \ldots, D_n$`) = e`

Here we simply guarantee that the most general case of the method is covered, and therefore trivially no 'message not found' error can occur.

By fairly simple reasoning we can see that these two restrictions are sufficient for any well-typed program. For every method call that occurs at run-time, either the method is internal or external. If it is external, then somewhere there has been defined a global default, so we are covered. If it is internal, then since we assumed the program was well-typed, we know the argument given has a run-time tag that is a subclass C' of the first argument C of the method being called. This means that in some module, we declared the class C', as a (possibly indirect) subclass of C. At that time, by restriction 1, we must have checked that a local default case for C' existed in the same module. This local default case is sufficient to show that at least one applicable case matches our method call, QED.

Notice that although we require a sort of imitation of either OOP languages in requiring local defaults or FP in requiring that we are certain of exhaustiveness at the point of declaration of a method (in this case by requiring a global default) this imitation is only a minimum requirement: we can also define more specific cases in addition to the local or global defaults for either internal or external methods, and this is a strict improvement over what ML or Java offers us.

1.2 Nonambiguity of Methods

To prevent ambiguous message warnings, we similarly turn to familiar language paradigms for ideas.

The reason Java doesn't have ambiguity problems is that although it does OO-style dynamic dispatch, it's only *single* dispatch. The run-time dispatch that takes place during the call `x.method(y,z)` only depends on the run-time tag of `x`, not those of `y,z`. This is related to the fact that all method *implementations* are textually bundled up with the class of their first argument.[2] The only place you can override a method — and what overriding a method means is creating a new specialized behavior for when the

[2]Be careful not to confuse this with the idea in the definition of internal vs. external: that was about method *declarations* being bundled with the class of their first argument

reciever of a method call is some class C — is in the scope of the declaration of class C.

One reason that means already that ML can't have ambiguity problems is that its semantics for case analysis is different from EML's. In ML, if multiple cases match, then the earlier cases have precedence. However, even if ML had EML's semantics of preferring the most specific case regardless of order (and the potential for ambiguity that comes with it) it could still do a good job of warning about ambiguity and compile-time. This is because all cases of a case analysis have to occur all in the same place: in EML terms, all of the method implementations have to be in the same place as the method declaration.

We don't wish to impose either of these restrictions wholesale on the programmer, so we give them the choice extention-by-extention whether to make a particular case (a) OOP-style (keeping it with the class of its first argument) or (b) FP-style (keeping it with the method declaration).

> **Restriction 3**: *Nonambiguity Constraint*. Every method implementation `extend m(`$C, D_1, \ldots, D_n$`) = e` must occur either (a) in the same module as the declaration of C or (b) in the same module as the declaration of m

Why does this prevent ambiguous match errors at run-type? Here is a sketch of a proof. Suppose we have a method call `call m(`$e_1, \ldots, e_n$`)`, and e_1 evaluates to an object value $\{D : \cdots\}$ of class D. Suppose that two cases match this call, say, `extend m(`C'`, ... ) =` e' and `extend m(`C''`, ... ) =` e''. In particular we know D is a subclass of both C' and C''.

Recall that, when checking the validity of a module, we do have available all the information in the interfaces of all the modules that module depends on. This means that if either of these two extentions satisfied part (b) of the nonambiguity contraint above, then any potential ambiguity would have been detected by the compiler. For suppose without loss of generality the first case `extend m(`C'`, ... ) =` e' is declared in the same module as the method m itself. Then the second case, `extend m(`C''`, ... ) =` e'', since it mentions the method m, statically depends on that module. So when we check the module containing the second case, the first case will be visible when we do the ordinary ambiguity checks mentioned in last lecture.

If both extentions satisfy only part (a) of the constraint, however, we have to reason differently. In that case it may be that the two cases occur in different modules, say M' and M'', each of which statically depends on the module that declares m, but neither of M',M'' depends on the other. (If

it happens that one *does* depend on the other, we are already done, for the same reason as the last case: suppose it's M' that depends on M''. While checking M' we will 'see' M'' and therefore see both extentions and detect that they are ambiguous) But we must actually satisfy part (a) for each extention. This means that C' is declared in M' and C'' is declared in M''. Because M' and M'' are assumed not to depend on one another, this means neither of C' and C'' are subclasses of one another. Since we don't have multiple inheritance, this means the fact that D is a subclass of C' and C'' is a contradiction, QED.