Modular Typechecking for Hierarchically Extensible Datatypes

Todd Millstein, Colin Bleckner, and Craig Chambers

(slides by Jason Reed)

September 22, 2004

Introduction

- **Extensibility**
- Functional Languages — **functional** extensibility
- Object Oriented Languages — **data** extensibility
- Goal is some sort of merger that allows both
- But we want to retain **modular** typechecking

Outline

1. Preliminaries
   - I. Extensibility in Functional Languages
   - II. Extensibility in OO Languages
   - III. Previous Work
2. EML
   - I. Motivating Examples
   - II. Basic Language Design
   - III. Other features (signature ascription)

Extensibility in Functional Languages

- Not often referred to as such by FPL programmers, usually taken for granted.
- Suppose we have a library that defines

```
datatype exp = App of exp * exp list
                (* f(e1, ... en) *)
              | Meth of string * arg list * exp * type
                (* rtn_type func(x1,...,xn) { ... } *) ...
```

- We can write in our client code

```
fun super_optimize (App(e, args)) = (* case for App *)
  | super_optimize (Meth (name, args, body, rtn_type))
      = (* case for Meth *)
            ...
```

Extensibility in Functional Languages 2

- Contrast this with the following pseudo-java code

```
abstract class Exp {...}
class App extends Exp { App(Exp e, List e) {...}...}
    // f(e_1, ... e_n)
class Meth extends Exp {
   Meth(String s, List e, Exp b, Type t) {...}
   ...
 }
    // rtn_type func(x_1,...,x_n) { ... }
```

- If this is in a library, can't write any new methods that case-analyze over `App` vs. `Meth`!

Extensibility in OO Languages

- However, suppose we want to add a new construct to the language of our compiler
- Easy in OO language
- Just define a new class

```
class IsHalting extends Exp {
    IsHalting(Exp e) {...}
    ...
}
    // ishalting(e)
```

- Override all methods that need overridden
- That's it!

Extensibility in OO Languages 2

- To a FPL hacker of the right persuasion this may seem kind of mysterious
- He/she sees a type in a library as given:

```
datatype exp = App of exp * exp list
             | Meth of string * arg list * exp * type
```

- Client can't just up and decide to add new possibilities

Previous Work

- **O'Caml** has an ML-style type system and an OO-style type system in the same language
- ...but `datatype` and `class` are different beasts
- **OML** has `objtype` which is a generalization of `datatype` and `class`
- ...but enforces a distinction between OO-extensible *methods* and FP-extensible *functions*.
- **ML**$_{\leq}$ unifies methods and functions
- ...but no pattern-matching, and no **modular** checking of extensible functions

Set Example

```
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)
  ...
```

Set Example 2

- Interject some quick comments before we finish:
- Syntax is quite close to ML, not so much to Java
- ML things: structure, struct, {records} and record types, pattern matching
- New things: abstract, class, extend, #.

Set Example 3

```
  ...
  extend fun add (i, s as CListSet {es=es,count=c}) =
    if (member i es) then s else CListSet(i::es,c+1)

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

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

Set Example 4

- What's going on? Simple class hierarchy:

$$Set \longleftarrow ListSet \longleftarrow CListSet$$

- and some functions `add` and `size` and `elems`.
- `size` more efficient for CListSet
- `elems` inherited by CListSet
- Ordinary OO stuff
- Typechecking takes place at resolution of **structures**; we only have one right now
- Note: 'owner' of `add` is 2nd arg

Functions in EML

- Somewhere define the “generic function”
- Elsewhere extend it
- Like ML pattern-matching cases
- **EXCEPT!** no notion of ‘first match’ — ‘best match’ instead
- ‘#’ Owner position — talk about later
- Single inheritance
- Possible errors: nonexhaustic match, ambiguous match
- Just like previous languages we’ve seen this class, prohibit multiple matches instead of fixing an order for ambiguity resolution

Functional Extensibility

```
structure UnionMod = struct
  fun union:(#Set * Set) ->
  extend fun union (s1, s2) = fold add s2 (elems s1)
  extend fun union (ListSet {es=e1},ListSet {es=es2}) =
    ListSet(merge(sort(e1), sort(e2)))
end
```

- New functionality in a separate structure
- Naïvely this looks okay from the point of view of exhaustivity and unambiguity

Data Extensibility

```
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
```

- New possibility for the type Set in a separate structure
- Looks like we've added a case for every function that needs new cases
- If we added `UnionMod` and `HashSetMod` it would be okay to call `union` on `HashSets`. Why?

Data Extensibility 2

```
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(ls@(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
```

Data Extensibility 3

- Here we see that we can reuse the representation type and change some of the methods
- `size` is still inherited
- A case to `union` is added
- `getMin` is added
- Again, everything **seems** to work out okay, no ambiguities or missing cases
- How can we be sure?

Type-Checking

- The paper talks a lot about **Implementation-side** Type Checking
- This is supposed to contrast with *Client-side* type-checking, where you make sure every use of the function is okay, instead of making sure the function cannot be misused.
- **Discussion Question:** Anybody's favorite language do the latter?
- How do we do ITC for EML?
- Naïve ITC ("just check all the dependencies") is unsound!
- At least without further restrictions

Challenge Case

Challenge Case

```
structure ShapeMod = struct
  abstract class Shape() of {}
  fun intersect:(#Shape * Shape) -> bool
end
```

Challenge Case

```
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
```

Challenge Case

```
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
```

Problems

- Naïve ITC says ok! **BUT**:
- `intersect(Circle{}, Shape{})` is ambiguous
- `intersect(Shape{}, Circle{})` is undefined
- `print(Circle{})` is undefined

Problems

- Naïve ITC says ok! **BUT**:
- `intersect(Circle{}, Shape{})` is ambiguous
- `intersect(Shape{}, Circle{})` is undefined
- `print(Circle{})` is undefined
- How to fix?

Problems

- Naïve ITC says ok! **BUT**:
- `intersect(Circle{}, Shape{})` is ambiguous
- `intersect(Shape{}, Circle{})` is undefined
- `print(Circle{})` is undefined
- How to fix?
- Make restrictions involving the **owner position**
- Owner can be any argument, possibly nested deeply
- Owner position of a function fixed by decl. of generic function
- The owner type must be a class
- Has some properties in common with OO notion of receiver

Restriction

- We say functions declared in the same module (i.e. structure) as their owner class are **internal**, all others **external**
- **Requirement**: external functions must have a **global default case**
- That is, a module that declares an external function must extend it with a case that covers all type-correct arguments
- This rules out `print` as we've defined it, because it only works for `Rect`s
- If we added a default case for `Shape`s, then it would be fine to pass a `Circle` to it

Restriction 2

- `intersection`'s still a problem, and it's an internal function
- Do we want to require global default cases for internal functions? **No**.
- Just local defaults, like in OO
- **Requirement**: for every module M containing a concrete subclass S of a class C that owns some internal function f, then M must have a **local default case** for f and S
- That is, M must extend f with a case that accepts anything of type S or a subclass of S at the owner position, and anything at all for every other position.
- In plain english, if you declare a subclass, you have to extend every function to deal with it at the owner position.

Restriction 2...

- This rules out `intersect` as we've defined it, because of `Rect`

```
...
  fun intersect:(#Shape * Shape) -> bool
...
  extend fun intersect(Shape _, Rect _) = ...
...
```

- We only consider `Rect` for the second argument!
- If we had put the owner position in the other spot, `Circle` would have failed

Restriction 3

- But suppose we added a `(Rect _, Shape _)` case to `RectMod`
- Ambiguity problem still there
- Say a function *case*'s owner is the type in the owner position
- **Requirement**: every function case must be defined in the same module as its owner, **or** the same module as the function declaration
- This rules out the `(Shape _, Rect _)` case being defined in `RectMod`
- This allows each function case to behave like an ML case (same module as function declaration) or an OO method (same module as its owner)

Caveats

- Take a moment to point out some less felicitous aspects of the language so far
- Can't as a client of `HashSetMod` and `UnionMod` write a special hashset-union
- Can't treat extensible functions as first class
- Have to give explicit types to functions
- Can't really simulate ML datatypes because of global default condition
- (but we'll fix this last problem in a moment)

Signatures

- The idea of modular type-checking is that you can check a module once and for all just knowing the **signature**s (i.e. interfaces) of all the modules it depends on
- So the implementation of other modules can change without harming it
- Up until now this has been implicit
- Just read off the (principal) signature from the structure
- But ML has a notion of *signature ascription*
- Expose only some things
- Not entirely unlike a class matching an interface in Java

Problem

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

Problem...

- If we pass a `Circle` to `print`, it will call `bad`, which is... bad.
- Suppose `print` were defined in a separate module
- Would then be an external function
- Would be required to have a global default case
- No problem!
- **Solution**: treat hidden functions as if they were in a separate module, for the purposes of enforcing restrictions

Other Forms of Ascription

- We see that we can omit some declarations (‘private methods’)
- Also can hide record fields (‘private fields’)
- Can ascribe a concrete class as abstract (analogue in OO languages? this does have an analogue in ML)
- Can ascribe a class as **sealed** (‘final’??)
- Sealing allows faithful encoding of ML datatypes by prohibiting further subtyping
- If an external function’s owner and all available subclasses are sealed, then the function need not have a global default, for no unexpected cases can arise

However

- Can't ascribe transitive superclass relationships
- Suppose C extends B extends A
- Can't ascribe C as extending A
- Could write cases for (A,A), (B,C), (C,B) in a module that only knows B extends A and C extends A.
- Ambiguous, (consider a (C,C) argument) but you only know that if you know C extends B!

Conclusion

- EML seems to be a nice step along some path of mixing functional and object-oriented programming ever closer to each other
- It doesn't necessarily come close to telling us how to Javify ML or MLify Java, or whateverify your whatever favorite language and paradigm
- Just for instance, it steps all over ML's philosophical toes in fundamental ways, if you ask the right people (who may be in the room at the moment)
- Nonetheless, I think it's an interesting exercise in finding a least upper bound of extensibility-power in some sense
- Maybe there's an altogether nicer upper bound?

Conclusion

- EML seems to be a nice step along some path of mixing functional and object-oriented programming ever closer to each other
- It doesn't necessarily come close to telling us how to Javify ML or MLify Java, or whateverify your whatever favorite language and paradigm
- Just for instance, it steps all over ML's philosophical toes in fundamental ways, if you ask the right people (who may be in the room at the moment)
- Nonetheless, I think it's an interesting exercise in finding a least upper bound of extensibility-power in some sense
- Maybe there's an altogether nicer upper bound?
- Questions, discussion?