Modular Typechecking for Hierarchically Extensible Datatypes

Todd Millstein, Colin Bleckner, and Craig Chambers

September 22, 2004 (slides by Jason Reed)
Introduction

But we want to retain modular typechecking.

Goal is some sort of merger that allows both

Object-Oriented Languages — data extensibility

Functional Languages — functional extensibility

Extensibility
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

We can write in our client code

•

Suppose we have a library that defines

taken for granted.

Not often referred to as such by FPL programmers, usually

Suppose we have a library that defines

taken for granted.

Not often referred to as such by FPL programmers, usually

Extensibility in Functional Languages

Extensibility in Functional Languages
Extensibility in Functional Languages

Contrast this with the following pseudo-Java code:

```java
abstract class Exp {
    // ... ...
}

class Meth extends Exp {
    // ... ...
}

class App extends Exp {
    // ... ...
}

abstract class Exp {
    // ... ...
}
```

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

### Contrast

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

2. Extensibility in Functional Languages
Extensibility in OOLanguages

However, suppose we want to add a new construct to the language of our compiler.

Just define a new class

Easy in OO Languages

Language of our compiler

Extensibility in OO Languages

class IsHalting extends Exp

{...}

IsHalting(e) //

{...} {Exp e) 

Override all methods that need overridden

That's it!
To a FPL hacker of the right persuasion this may seem kind of mysterious.

Client can’t just up and decide to add new possibilities.

```haskell
datatype exp = App of exp * exp

Meth of string * list * exp * type

| data type exp = App of exp * exp * list
```

He/she sees a type in a library as given:

Extensibility in OO Languages 2
extensible functions

...but no pattern-matching, and no modular checking of

ML uniﬁes methods and functions.

and FP-extensible functions.

...but enforces a distinction between O-O-extensible methods

OML has ObjType which is a generalization of datatype and

class

...but datatype and class are different beasts

system in the same language

OCaml has an ML-style type system and an O-O-style type

Previous work
class CListSet(es:intlist, c:int) extends ListSet(es) of {count:int = c}

class ListSet(es:intlist) extends Set of {es:intlist = es}

class ListSet extends ListSet of {es:intlist = es}

abstract class Set extends Structure & Method = struct

Set Example
Set Example 2

New things: abstract, class, extend, #.

Pattern matching

ML things: struct, struct, records and record types.

Syntax is quite close to ML, not so much to Java

Interject some quick comments before we finish:
fun size: Set -> int
  extend fun size (ListSet {es=es}) = length es
  extend fun size (CListSet {es=es, count=c}) = c
  end

fun elems: Set -> int list
  extend fun elems (ListSet {es=es}) = es
  extend fun elems (CListSet {es=es, count=c}) = [c]
  end

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

Set Example 3
What's going on?

Simple class hierarchy:

- Set
- ListSet
- CListSet

And some functions: `add`, `size`, `elems`.

- `size` more efficient for `CListSet`
- `elems` inherited by `CListSet`
- Ordinary OO stuff

Note: `owner` of `add` is 2nd arg

Typcheck takes place at resolution of structures; we only have one right now

Set Example 4

What's going on? Simple class hierarchy:
resolution

multiple matches instead of fixing an order for ambiguity

Just like previous languages we've seen this class, prohibit

Possible errors: nonexhaustive match, ambiguous match

Single inheritance

owner position - talk about later

EXCEPT!
Like ML pattern-matching cases

Elsewhere extended it

Someplace define the "generic function"

Functions in EML
New functionality in a separate structure

and unambiguity

Naively this looks okay from the point of view of exhaustivity

Functionality Extensibility

end
Why?

If we added `union` and `HashSet` it would be okay to call `new_cases`

Looks like we've added a case for every function that needs

New possibility for the type `Set` in a separate structure

```
extend fun add(i, s: HashSet{ht=ht}) =
  if containsKey(i, ht) THEN s
  else HashSet{put(i, (), ht)}
end
```

```
extend fun size(HashSet{ht=ht}) = numEntries(ht)
else HashSet{put(1, (), ht)}
  if containsKey(1, ht) THEN s
  = (ht, (ht=ht))
end
```

```
extend fun (ht: HashSet{ht=ht}) = s
end
```

```
class HashSet{ht: (int, unit) hashtable}
structure HashSetMod = struct
```

Data Extensibility
class SListSet(es: intlist) extends ListSet(es) of {}

extend fun add(i, sas: 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 2
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?
The paper talks a lot about Implementation-side Type-Checking.

Type-Checking

Implementation-side Type-Checking

Discussion Question: Anybody's favorite Language to do the latter?

At least without further restrictions •

Naive ITC ("just check all the dependencies") is unsound!

How do we do ITC for EML?

Cheecking

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.

Anyone's favorite Language to do the latter?
Challenge Case
structureShapeMod=struct
abstract class Shape() of {}

fun intersect:(#Shape * Shape) -> bool
end

structureCircleMod=struct
class Circle() extends Shape() of {}
extend
fun intersect(Circle_, Shape_) =...
end

structureRectMod=struct
class Rect() extends Shape() of {}
extend
fun intersect(Shape_, Rect_) =...
fun print: Shape -> unit
extend
fun print(Rect_) =...
end

Challenge Case
Challenge Case

structure Shapemod = struct
  abstract class Shape() of
  fun intersect(Shape *, Shape *) = true
  end
structure RectMod = struct
  class Rect() extends Shape() of
    extend fun intersect(Rect *, Shape *) = true
    end
  end
end
Challenge Case

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

structure RectMod = struct
  class Rect() extends Shape() of
    extend fun intersect(Rect_,Shape_)=...
    fun print:Shape->unit
    extend fun print(Rect_)=...
    end
end

structure CircleMod = struct
  class Circle() extends Shape() of
    extend fun intersect(Circle_,Shape_)=...
    end
end
```

Problems

Nave IT says ok!

BUT:

\[ \text{intersect(Circle } fg, \text{Shape } fg) \text{ is ambiguous} \]

\[ \text{intersect(Shape } fg, \text{Circle } fg) \text{ is undefined} \]

\[ \text{print(Circle } fg) \text{ is undefined} \]

\[ \text{intersects(Shape } fg, \text{Circle } fg\} \text{ is undefined} \]

\[ \text{intersects(Circle } fg, \text{Shape } fg\} \text{ is ambiguous} \]

\[ \text{Nave IT says ok! BUT:} \]

Problems
Problems

Nave IT says ok!

BUT:

intersect(Circle_{fg}, Shape_{fg}) is ambiguous

intersect(Shape_{fg}, Circle_{fg}) is undefined

print(Circle_{fg}) is undefined

How to fix?

- Makerestrictionsinvolvingthe
  - Ownerposition
  - Ownercanbeanargument,possiblynesteddeeply
  - Ownerpositionofafixedbydecl.ofgenericfunction
  - Thownertypemustbeaclass
  - HassomepropertiesincommonwithOOnotionofreceiver

Naive ITC says ok? BUT:

Problems
Problems

NaveIT! says ok!

BUT:

\[
\text{intersect(Circle}_\text{fg}, \text{Shape}_\text{fg}) \text{ is ambiguous}
\]

\[
\text{intersect(Shape}_\text{fg}, \text{Circle}_\text{fg}) \text{ is undefined}
\]

\[
\text{print(Circle}_\text{fg}) \text{ is undefined}
\]

How to fix?

Make restrictions involving the owner position

Owner position of a function fixed by decl. of generic function

Owner can be any argument, possibly nested deeply

Has some properties in common with O0 notion of receiver

The owner type must be a class

Owner position of a function fixed by decl. of generic function

Owner position of a function fixed by decl. of generic function

Owner position of a function fixed by decl. of generic function

Hassome properties in common with OO notion of receiver

20-b
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.

This rules out `print` as we've defined it, because it only works for `Rects`.

If we added a default case for `Shapes`, then it would be fine to extend it with a case that covers all type-correct arguments. That is, a module that declares an external function must pass a `Circle` to it.

If we added a default case for `Shapes`, then it would be fine to extend it with a case that covers all type-correct arguments. That is, a module that declares an external function must pass a `Circle` to it.
every function to deal with it at the owner position.

In plain English, if you declare a subclass, you have to extend
at all for every other position.

type or a subclass of S at the owner position, and anything
that is, M must extend f with a case that accepts anything of
S and f must have a local default case for and S.

Requirement: For every module containing a concrete
subclass S of a class C that owns some internal function
f, then M must extend f with a case that accepts anything of
S at the owner position, and anything at all for every other position.

Just local defaults, like in OO.

No.

Do we want to require global default cases for internal
intersection is still a problem, and it’s an internal function

Restriction 2
Restriction 2...

This rules out intersect as we've defined it, because of rect

If we had put the owner position in the other spot, circle

We only consider rect for the second argument!

... extend fun intersect (shape : Shape, rect : Rect) = ...

... fun intersect : (#Shape * Shape) -> bool

... This rules out intersect as we've defined it, because of rect

Restriction 2...
But suppose we add a 

\((\text{Rect}, \text{Shape})\) case to 

\(\text{RectMod}\) 

\(\text{Mod}\) 

This allows each function case to behave like an ML case (same module as its owner). 

Requirement: every function case must be defined in the same module as its owner, or the module as the function definition. 

Ambiguity problem still there. 

\(\text{Restriction 3}\)
Caveats

- (but we'll fix this last problem in a moment)
- Can't treat extensible functions as first class
- Can't really simulate ML datatypes because of global default condition
- Can't treat extensible functions as first class
- Can't as a client of `HashSetMod` and `UnionMod` write a special

language so far
- Take a moment to point out some less felicitous aspects of the
The idea of modular type-checking and interfaces in Java:

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

Signatures
Problem
Problem...

If we pass a Circle to print, it will call bad, which is bad.

Suppose print were defined in a separate module.

Would be required to have a global default case.

Would then be an external function.

If we pass a Circle to print, it will call bad, which is... bad.

Solution: treat hidden functions as if they were in a separate module, for the purpose of enforcing restrictions.

No problem!
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 (‘final’ in OO)
- Can ascribe a class as sealed (‘final’ in ML)
- Sealing allows faithful encoding of ML datatypes by forbidding 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

Ambiguous (consider a (C, C) argument) but you only know that if you know C extends B.

Could write cases for (A, A), (B, C), (C, B) in a module that only knows B extends A and C extends A.

Suppose C extends B extends A.

Can't ascribe C as extending A.
Conclusion

Maybe there's an altogether nicer upper bound?

Least upper bound of extensibility-power in some sense

Nonetheless, I think it's an interesting exercise in finding a

the room at the moment

fundamental ways, if you ask the right people (who may be in

Just for instance, it steps all over ML's philosophical toes in

and paradigm

or ML!ify Java, or whatever your whatever favorite language

It doesn't necessarily come close to telling us how to Javaify ML

other

functional and object-oriented programming ever closer to each

EML seems to be a nice step along some path of mixing

Conclusions
Conclusion

EML seems to be an nicest step along some path of mixing functional and object-oriented programming ever closer to each other.

Some people might argue that EML is not a valid approach to mixing functional and object-oriented programming because it doesn't necessarily come close to telling us how to Javify ML or MLify Java, or whatever language is your favorite paradigm.

Nonetheless, I think it's an interesting exercise in finding a least upper bound of extensibility-power in some sense.

Just for instance, it steps all over ML's philosophical toes in the room at the moment.

I think it's an interesting exercise in finding a least upper bound of extensibility-power in some sense. If you ask the right people (who may be in fundamental ways, if you ask the right people) who may be in the right room at the moment.

Questions, discussion?

31-a