True Higher-Order Modules, Separate Compilation, and Signature Calculi

George Kuan

University of Chicago

PL Seminar
March 6, 2009

Why Higher-Order Functors?

```
functor F() = struct ... end
functor G() =
struct
  structure M = F()
end

functor G'(functor F() : sig end) =
struct
  structure M = F()
end
```

Commentary on Standard ML: Separate compilation and abstraction over functors

Why Higher-Order Functors?

Indefinite references to functors

In action: simple and succinct extensions of functors (Biswas 95)

```
functor RedBlackSetFn(K:ORD_KEY) ...
functor ExtSetFn
  (functor SetFn(Ord:ORD_KEY): ORD_SET)
  (K:ORD_KEY) =
struct
  structure M = SetFn(K)
  open M
  (* Extensions to SetFn *)
  type ...
  val ...
end
```

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                          : sig type t end)
             (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

???

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                        : sig type t end)
              (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

structure M : **sig** **type** t **end**

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                         : sig type t end)
             (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

structure M : **sig** **type** t **end** Too conservative!

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                        : sig type t=X.t end)
             (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

???

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                          : sig type t=X.t end)
             (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

structure M : **sig** **type** t = int **end**

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                       : sig type t=X.t end)
             (M : sig type t end)
  = F(M)

functor Id(X : sig type t end)
  = struct type t = X.t end

structure M = Apply(functor F=Id)
                   (struct type t = int end)
```

structure M : **sig** **type** t = int **end** Too restrictive!

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                        : sig type t=X.t end)
             (M : sig type t end)
  = F(M)

functor K(X : sig type t end)
  = struct type t = int end

structure M = Apply(functor F=K)
                   (struct type t = int end)
```

???

Full Transparency in Higher-Order Functors

```
functor Apply(functor F(X: sig type t end)
                       : sig type t=X.t end)
             (M : sig type t end)
  = F(M)

functor K(X : sig type t end)
  = struct type t = int end

structure M = Apply(functor F=K)
                   (struct type t = int end)
```

Signatures of formal functor F and K don't match

Full Transparency in Higher-Order Functors

Definition (Type Action)

The way in which a functor computes its output types from its parameter types including generativity and actions of formal functor components

Definition (Full Transparency)

The propagation of all type actions in a functor through higher-order functor applications.

Definition (True Higher-Order Functors)

True higher-order functors respect the full transparency property.

Applicative Functors (Leroy 95)

Type equivalence = path equivalence
Notion of paths extended to application of functor to another path F(M).t
Need theory of equality of paths

```
Apply : functor(functor F(X: sig type t end)
                          : sig type t end)
               (structure M : sig type t end)
        : sig type t = F(M).t end
```

Shortcomings of Applicative Functors

Lacks generative semantics

```
functor SymbolTable() =
  struct type symbol = int ... end
  :> sig type symbol ... end
```

Shortcomings of Applicative Functors

Functor applications in paths must be A-normalized

```
signature T = sig type t end

functor:
  functor ApplyToInt(functor G(X:T):T) =
    G(struct type t = int end)

signature:
  functor ApplyToInt(functor G: (X:T):T) : T

structure R = ApplyToInt(functor G = Id)
```

val x : R.t = 42 int mismatch R.t = ApplyToInt(Id).t

Design Space

1. Applicative functors (OCaml)
2. Include both applicative and generative functors (Moscow ML, DCH)
3. ...

True Higher-Order Functors

Alternative: Fully transparent generative higher-order functors

Examples: Re-elaboration semantics (MacQueen and Tofte 94) and internal language semantic representation static lambda calculus (implicitly in SML/NJ)

Claim

1. True HO functor semantics is exactly what we want
2. Applicative functors are an "in-between" approximation

True Higher-Order Functors

Why are they much more difficult than the first-order case?

- First-order: hide abstract types - access by interface of functions
- Higher-order: hide type action - ???

Separate Compilation

Key Problem

True higher-order functors do not seem to be compatible with true separate compilation. The signature language cannot describe type action propagation adequately in functor signatures.

Definition of True Separate Compilation

Cardelli 97

True separate compilation is the ability to separately typecheck program fragments in the presence of a local environment (a set of explicit interfaces) such that the fragments can be safely linked together.

True Separate Compilation

Conjecture

True HO functors and true separate compilation are mutually exclusive

Reasoning: Intuitively, necessary signature and type language too complex...Should be fairly straightforward

Signature Calculi

The ML signature language is a simple interface language with support for signature extension (syntactic **include**), hierarchical nesting, where type clauses, and type sharing constraints.

But as Ramsey *et al.* 05 and Garcia *et al.* 05 noted, richer extensions would be useful.

SML/NJ Implementation of Signature Extension

```
signature S2 = sig
  include S0
  include S1
end
```

		S0 type	S0 eqtype	S0 datatype	S0 deftype
S1	type	✓	eqtype	✗	✗
S1	eqtype	type	✓	✗	✗
S1	datatype	✓	datatype	✗	✗
S1	deftype	✗	✗	✗	✗

SML/NJ signature elaboration compatible signature merging:
✓ can be merged, ✗ cannot be merged, otherwise indicates specs merge-able but indicated spec takes precedence

Ramsey *et al.* Signature Extension

```
signature S0 =
sig
  type t
  type u
  val x : t list
end
```

```
signature S1 =
sig
  type t
  type u
  val x : u list
end
```

```
S0 andalso S1 =
sig
  type t
  type u = t
  val x : t list
end
```

Signature Calculi (1)

Merge

```
signature SIG0 = sig eqtype t end
signature SIG1 = sig type t = int end
signature SIG =
sig
  include SIG0
  include SIG1
end
```

Signature Calculi (2)

Access inferred signatures

```
structure M =
struct
  type t = int
end
signature S = sign(M)
```

Signature Calculi (3)

Signature components in modules

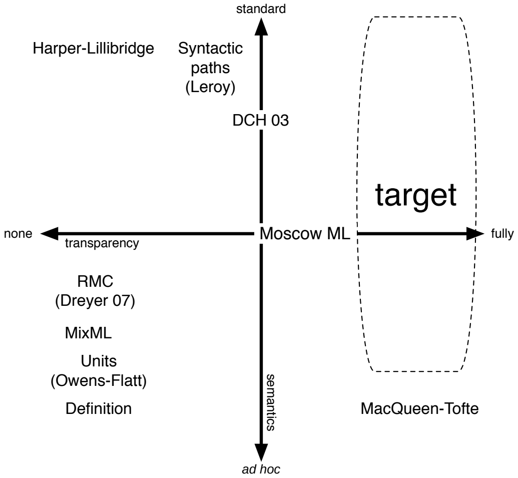
```
structure Control =
struct
  signature PRINT = sig ... end
  structure Print : PRINT = struct ... end
  ...
end
```

Signature Calculi (4)

Parameterized signatures

```
signature SIG0(M: sig type t end)
  = sig type t = M.t end
```

Related Work



Conclusion

1. Static and dynamic semantics for true HO modules based on SML/NJ and recent module system designs
2. Signature calculi with compatible merges and other signature manipulation elements
3. Towards a Successor ML

Thank You

Related Concepts

Type inference from first-class polymorphism

MLF, HMF, and FPH partially infer the types of higher-order functions. Can we do something similar with higher-order functors?

Automatic instantiation from type classes

Type classes dispatch the operator of the instance whose type matches the invocation without having to explicit instantiate the class each time. Can we do this with the module system under the same limited circumstances?

Applicative Functors

Leroy showed that there exists a type-preserving encoding of the strong sums calculus in the applicative functors calculus. This excludes generativity.

Potential Questions

1. What are the technical challenges to proving type soundness?
2. What kind of difficulties do signature components introduce?
3. Are there any potential improvements of the compiler's approach to signature matching and subtyping?
4. What are the main approaches for proving mutual exclusion of separate compilation and true higher-order functors