The ML Module System

Donald Pinckney

"What is the ML module system? It is difficult to say."
— Derek Dreyer's PhD Thesis

0. Why modularity? Pre-ML work on modules

1. MacQueen's Module Proposal for SML
   → Examples & implementation, unclear semantics

2. Modelling modules using dependant type theory
   → Dependent sums vs. existential types

3. Translucent types
   → Novel, weird dependant types

Modularity

~ late 60s : "software crisis"

→ See Cameron's Contracts lecture

↓

Languages need features to help programmers build their systems from separate subsystems.

System F ... Smalltalk ...

Java Interfaces

Generics

Step 0. Liskov's CLU ~70s

ML

→ See Ryan's System-F lecture

Step 1. Modules for ML

...

0. Barbara Liskov's CLU Language
~1970s, CLU Reference Manual

→ Abstract data types via clusters.

Recurring Example: Set data structure

IntSet = cluster is empty, insert, member

rep = array[int] → this is hidden to clients!

empty = ...
insert = ...
member= ... if input < this[i] then... But implementation code knows the type
end IntSet

IntSet s := IntSet $ create()
IntSet $ insert (s, 12)

+ Polymorphism !

→ Parameterized Modules

```
Set = cluster[T:type] is empty, insert, member
    where T has compare: proctype(T,T) returns (order)
                    → with some required interface
        rep = array[T]

        empty = ....
        insert = ....
        member=... if T$compare(input, this[i]) =LESS then...
end Set


Set[int] s := Set[int] $ create()
Set[int] $insert(s, 12)
```

Questions on concept of Modules so-far?

ML as meta-language for LCF ~70s
theorem prover

↓

ML as stand-alone functional language ~80s

→ Uh-oh, we need a module system, to support "programming in the large"

Step 1. Modules for Standard ML, MacQueen 1984

3 components:

Ex 1

Signatures	Structures	Functors
↓	↓	↓
Abstract Spec.	Implementation	Functions on structures

Signature Syntax:

<SIGNATURE> ::=

sig

<type specs>

<value specs>

<structure specs>

end

Structure Syntax:

<STRUCT> :=

struct

<type impls>

<value impls>

<structure impls>

end

Ex 1: Int set!

```
signature INT_SET = sig
    type set
    val empty: set
    val insert: int * set -> set
    val member: int * set -> bool
end
```

```
structure IntSet : INT_SET = struct
    type set = int lis
    val empty = []   "private"
    val insert = λ(x,s). ...
    val member = λ(x,s).
        ... if x <= s[i] then ...
end
```

Using it:

TC: ✓

let s: IntSet.set = IntSet.empty in IntSet.insert (12, s)

✗ int list ≠ IntSet.set

TC: ✗

let s': int list = IntSet.empty in ...

Can we generalize the signature?

```
signature SET = sig
    type item
    type set
    val empty: set
    val insert: item * set -> set
    val member: item * set -> bool
end
```

Then implement:

```
structure IntSet' : SET = struct
    type item = int
    type set = item list        (or int list)
    val empty = []
    val insert = λ(x,s). ...            ✓ ( ⊢ x: item, ⊢ item = int )
    val member = λ(x,s). ... if x <= s[i] then ...
end
```

Using it: let s = IntSet'.empty in IntSet'.insert(12,s)

$\not\vdash$ IntSet'.item = int   ✗

Can we generalize the signature?

```
signature SET = sig
    type item
    type set
    val empty: set
    val insert: item * set -> set
    val member: item * set -> bool
end
```

Then implement:

```
structure IntSet': SET = struct
   type item is int            "public"
   type set = item list        (or int list)
   val empty = []
   val insert = λ(x,s). ...            ✓ ( ⊢ x: item, ⊢ item = int )
   val member = λ(x,s). ... if x <= s[i] then ...
end
```

Using it: let s = IntSet'.empty in IntSet'.insert(12,s) ✓

Functors: Parameterize a structure by a structure

```
functor (<formal params>) : <SIGNATURE> =
    <STRUCT>
```

a

```
functor  Set (Item: ? ) : SET  = struct
    type item is Item.item
    type set = item list
    val empty = []
    val insert = ...
    val member = λ(x,s). ... if Item.compare x S[i] ...
end
```

@ Potential structure <u>argument</u> for Set:

```
structure IntOrder : ______ = struct
    type item is int
    val compare : int * int -> bool = λ(x,y). ....
end

signature ORD = sig
    type item
    val compare: item * item -> bool
end

functor Set (Item: ORD) : SET  = struct
  type item is Item.item
  type set = item list
  val empty = []
  val insert = ...
  val member = λ(x,s). ... if Item.compare x S[i] ...
end
```

<u>Using it</u>: structure IntSet'' = Set(IntOrder)
let s=IntSet''.empty in IntSet''.insert (12, s) ✓
⊢ IntSet''.item = IntOrder.item = int

Main Features of MacQueen's Modules:

- Signature ✓
- Structures ✓
- Functors ✓

Additional Features:

- Type propagation (type item is int) ✓
- Nested structures
- Sharing

Ex 2

Ex 2 : Set Intersection

signature SPAIR = sig
    structure X : SET
    structure Y : SET
end

functor SPair (X' : SET, Y' : SET) : SPAIR = struct
    structure X = X' , Y = Y'
end

functor Intersect (P : SPAIR) : SET = struct

    type item is P.X.item
    type set = P.X.set * P.Y.set
    val empty = (P.X.empty, P.Y.empty)
    val insert = ...
    val member = $\lambda(m, (s_x, s_y))$.
        P.X.member m s_x and $\underline{\text{P.Y.member m } s_y}$
end

$\vdash m : P.X.item$

$\nvdash P.X.item = P.Y.item$

Ex 2: Set Intersection

```
signature SPAIR = sig
    structure X : SET
    structure Y : SET
end

functor SPair (X' : SET, Y' : SET) : SPAIR = struct
    structure X = X' , Y = Y'
end

functor Intersect (P : SPAIR) : SET = struct
    sharing P.X.item = P.Y.item
    type item is P.X.item
    type set = P.X.set * P.Y.set
    val empty = (P.X.empty, P.Y.empty)
    val insert = ...
    val member = λ(m, (s_x, s_y)).
          P.X.member m s_x and P.Y.member m s_y
end
```

$\vdash$ P.X.item = P.Y.item

Main Features of MacQueen's Modules:

- Signature
- Structures
- Functors

Signature, Structures: can include structure fields!

Additional Features:

Type propagation
(type item is int)

Sharing
(sharing P.X.item = P.Y.item)

Propagate <u>more</u> info

Step 2. <u>Abstract types have existential types</u>, Mitchell and Plotkin 1985

↓

<u>Using Dependent Types to Express Modular Structure</u>, MacQueen 1986

<u>Mitchell & Plotkin:</u>

- Use <u>existential types</u> for modules.
- Based on System-F

$$\frac{\Gamma \vdash M : \sigma[t \mapsto \tau]}{\Gamma \vdash (\underline{\text{pack}}\ \tau\ M\ \underline{\text{to}}\ \exists t.\sigma) : \exists t.\sigma} \text{(Intro)}$$

Implementation Type

Operations

$\text{IntSet} = \underline{\text{pack}}\ (\text{int list})\ ([\,], \lambda(x,s).\cdots, \lambda(x,s).\cdots)$
$\quad \underline{\text{to}}\ \exists S.\ S * (\text{int} * S \to S) * (\text{int} * S \to \text{bool})$

$\Gamma \vdash [\,] : \text{int list}$ ✓

$$\frac{\Gamma \vdash M : \exists t.\sigma \qquad \Gamma, x:\sigma \vdash N : \rho}{\Gamma \vdash \underline{\text{abstype}}\ s\ \underline{\text{with}}\ x\ \underline{\text{is}}\ M\ \underline{\text{in}}\ N : \rho} \quad (\exists \text{Elim})$$

Abstract type

Ops.

Existential to unpack

$\text{IntSet} : \exists S.\ S * (\text{int} * S \to S) * (\text{int} * S \to \text{bool})$

$\underline{\text{abstype}}\ s\ \underline{\text{with}}\ \text{ops}\ \underline{\text{is}}\ \text{IntSet}\ \underline{\text{in}}$

$\underline{\ldots}$ $\Gamma = \cdot, \text{ops} : s * (\text{int} * s \to s) * (\text{int} * s \to \text{bool})$

$(\text{snd}\ (\text{snd}\ \text{ops}))\ (3, \text{fst}\ \text{ops}) \qquad : \text{bool}$ ✓

Parameterized Modules? Use $\forall$!

$\text{Set} = \Lambda T.\ \underline{\text{pack}}\ (T\ \text{list})\ (\ldots)$
$\underline{\text{to}}\ \exists S.\ S * (T * S \to S) * (T * S \to \text{bool})$

$\text{Set} : \forall T.\ \exists S.\ S * (T * S \to S) * (T * S \to \text{bool})$

Problem: Too restrictive! (a)

$$\frac{\Gamma \vdash M : \exists t.\sigma \qquad \Gamma, x:\sigma \vdash N : \rho}{\Gamma \vdash \underline{\text{abstype}}\ s\ \underline{\text{with}}\ x\ \underline{\text{is}}\ M\ \underline{\text{in}}\ N : \rho}$$

$Ord = \exists T.\ T * T \to bool$

$\underline{\text{abstype}}\ s\ \underline{\text{with}}\ le\ \underline{\text{is}}\ (\underline{\text{pack}}\ int\ \lambda(x,y).\cdots\ \underline{\text{to}}\ Ord)\ \underline{\text{in}}$

$\underline{\cdots}$ $\Gamma = \circ, le : s * s \to bool$

$\downarrow$

$le\ 3\ 5$ $\times$ $\Gamma \nvdash s = int$

Parameterized Modules? Use $\forall$?

$Set : \forall T.\exists S.\ S * (T * S \to S) * (T * S \to bool)$

$Set = \Lambda T.\ \underline{\text{pack}}\ (T\ list)\ (\cdots)\ \underline{\text{to}}\ (\cdots)$

Problem: Can't parameterize by structs, only types. (b)

Using Dependent Types to Express Modular Structure,

MacQueen 1986

→ Argues for using full MLTT

ⓐ Existential types $\exists$ ⟶ Dependent sum Σ

ⓑ Universal quantification $\forall$ ⟶ Dependent function Π

ⓐ

$$\frac{\Gamma \vdash \tau : T \qquad \Gamma \vdash M : \sigma[t \mapsto \tau]}{\Gamma \vdash (\tau, M) : \sum_{(t:T)} \sigma} \quad (\text{Intro } \sigma)$$

→ MacQueen chooses $T = \text{Type}_1$

$$\frac{\Gamma \vdash P : \sum_{(t:T)} \sigma}{\Gamma \vdash \pi_2(P) : \sigma[t \mapsto \pi_1(P)]}$$

(∼ Elims)

$$\frac{}{\pi_1((\tau, M)) = \tau} \qquad \ldots$$

ⓐ $Ord = \sum_{T:Type} T * T \to bool$

let $le = \pi_2 ((int, \lambda(x,y). \cdots))$ in

$le\ 2\ 3$ ✓

$\vdash le : \pi_1(\cdots) \times \pi_1(\cdots) \to bool$

and $\pi_1(\cdots) = int$

a ✓

b

may depend on x

$$\frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A).\, b : \prod_{x:A} B} \quad \text{(Intro)}$$

$$\frac{\Gamma \vdash M : \prod_{x:A} B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B[x \mapsto N]} \quad \text{(Elim)}$$

$$\mathrm{Ord} = \sum_{T:\mathrm{Type}} T * T \to \mathrm{bool}$$

type depends on value

b ✓

$$\mathrm{Set} : \prod_{i:\mathrm{Ord}} \sum_{S:\mathrm{Type}} S * (\pi_1(i) * S \to S) * (\pi_1(i) * S \to \mathrm{bool})$$

$$\mathrm{Set} = \lambda(i{:}\mathrm{Ord}).\, (\pi_1(i)\ \mathrm{list}, ([\,], \lambda.\cdots \pi_2(i) \cdots))$$

$$\mathrm{Set}\ ((\mathrm{int}, \lambda(x,y).\cdots)) : \sum_{S:\mathrm{Type}} S * (\mathrm{int} * S \to S) * (\mathrm{int} * S \to \mathrm{bool})$$ ✓

Σ types

"transparent" encapsulation

Π types

parameterizing structures by:
- Types
- Values
- Structures

vs

$\exists$ types

"opaque" encapsulation

$\forall$ types

parameterize by: - Types

What's missing in the model?

? Type propogation: type t = int vs. type t is int

? Sharing: sharing P.X.item = P.Y.item

Dependent Types are Concerning:

Modules can't be "compiled away" ? Complex...

Quick Mention: Higher-order Modules and the Phase Distinction

Harper, Mitchell, Moggi 1990

→ Working on a formalization of ML

→ Develop a phase-distinction in their ML calculus, so modules can be compiled away.

What's missing in the model?

? Type propagation

? Sharing

Step 3. Translucent Types

<u>A Type-Theoretic Approach to Higher-Order Modules with Sharing</u>

Harper and Lillibridge

1994

<u>Manifest Types, modules, and separate compilation</u>

Leroy

<u>Type Theory:</u> $\exists$ Σ

opaque $\longleftrightarrow$ transparent

<u>SML:</u> type t = ... type t is ...

sharing

<u>Idea</u>: Take existential types, generalize by adding optional type equations

<u>Syntax</u>: $T ::= \cdots \mid \alpha \mid \{D_1, \ldots, D_n\} \mid V.b \mid \cdots$

dependencies

$$D ::= b \triangleright \alpha : \mathrm{Type} \mid b \triangleright \alpha : \mathrm{Type} = T \mid y \triangleright x : T$$

Public Names

Private names

Public type info.

dependencies

$$V ::= \cdots \mid \{B_1, \ldots, B_n\} \mid V.y \mid \cdots$$

$$B ::= b \triangleright \alpha = A \mid y \triangleright x = V$$

Public Names

Private names

Private Implementation

$$\Gamma ::= \cdot \mid \Gamma, \alpha : \mathrm{Type} \mid \Gamma, \alpha : \mathrm{Type} = T \mid \Gamma, x : T$$

Ex:

$$IntSet = \{$$
$$\quad item \triangleright I = int$$
$$\quad set \triangleright S = I \; list$$
$$\quad \vdots$$
$$\quad member \triangleright m = \lambda(x,s).\ \cdots\ x <= s[i]\ \cdots$$
$$\}$$

Type inference...

$$IntSet : \{$$
$$\quad item \triangleright I : Type = int$$
$$\quad set \triangleright S : Type = int \; list$$
$$\quad \vdots$$
$$\quad member \triangleright m : int * (int \; list) \to bool$$
$$\}$$

Fully transparent!

TC: ✓

$$IntSet.member\ (12,\ 3 :: IntSet.empty)$$

→ Fully transparent

Ex:

IntSet = {
  item ▷ I = int
  set ▷ S = I list
  ⋮
  member ▷ m = λ(x,s). ⋯ x <= s[i] ⋯
} : {   ← Forced coercion
  item ▷ I : Type = int
  set ▷ S : Type
  ⋮
  member ▷ m : int * (int list) → bool
}

↳ Subtyping!

Main idea: Use translucent sums to (possibly) propagate type equalities, then hide info via subtyping.

Translucent Sums

- Subtyping relationship on signatures
- Let user choose what type equations to propogate
- Functors as before, Π types

- Can also encode SML's sharing, by dependent order of sum fields.

What's missing in the model?

✓ Type propogation

✓ Sharing

✗ A Bunch of other dimensions I've hidden...

'84

MacQueen's SML Proposal

Mitchell + Plotkin: $\exists$ types '85

$\forall$ types

MacQueen: Σ types '86

Π types

Harper: Phase Distinction '90

Harper + Lillibridge / Leroy: '94

Translucent Types

Russo: '98

Non-dependent types

Leroy: '95

Applicative Functors

OCaml

<u>Great resource</u>: Derek Dreyer's PhD thesis

Ex 3: Symbol Tables
→ Derek Dreyer's Thesis

```
signature SYMBOL_TABLE = sig
    type symbol
    val string2symbol : string -> symbol
    val symbol2string : symbol -> string
end
```

```
functor SymbolTable () : SYMBOL_TABLE = struct
    type symbol = int
    val table = <allocate new hash table>   -> private
    val string2symbol = λstr.
        <lookup or insert str into table, return index>
    val symbol2string = λi. table[i]
end
```

Using it:

```
structure ST1 = SymbolTable()
structure ST2 = SymbolTable()
                ⊬ ST2.symbol = ST1.symbol
ST2.symbol2string(ST1.string2symbol("HoPL"))
```