Modules Matter Most

Robert Harper
Carnegie Mellon University

MacQueen Fest — Chicago May 2012

Thanks

. . . to the organizers for organizing this meeting.

. . . to Dave for being an inspiration to and influence on my career.

Dave's Hit Parade

- Kahn-MacQueen networks: laziness and concurrency.
- HOPE: model functional language.
- Ideal model: semantics of type inference.
- Typed control operators: enabler for CML.
- SML/NJ: making FP a practical reality.
- The Definition of Standard ML.

Most Importantly . . .

The ML Module System

- A great improvement on most of its successors.
- A standard against which other designs are compared.
- Not easily improved upon (and rather easily worsened).

Like the λ-calculus itself, a profound infuence on all that follows.

Key Ideas

The modules design featured a number of key ideas:

- Signatures (interfaces) and structures (basic modules).
- Fibered representation of families of modules.
- Dependent types for hierarchy and parameterization.
- Subtyping to mediate composition.
- Sharing constraints to express coherence conditions.
- Type classes for flexible reuse.
- Effect-sensitive abstraction (aka "generativity").

A sharply original, and therefore socially disruptive, design.

Type Classes

A type class is a (partially) interpreted type:

```
signature EQ = sig
    type t
    val eq : t × t → bool
end

signature MONAD = sig
  type α t
  val return : α → α t
  val bind : (α → β t) → α t → β t
end
```

Matching

Signatures serve as constraints, rather than prescriptions, by subtyping:

```
signature AST = sig
  datatype t = Var of symbol | Plus of t × t | ...
  val eq : t × t → bool
  val parse : string → t
  val format : t → string
 end
```

Every implementation of `AST` is also of class `EQ`, without special provision.

Abstraction

A "more abstract" signature:

```
signature AAST = sig
  type t
  val eq : t × t → bool
  val parse : string → t
  val format : t → string
 end
```

Every implementation of `AST` is also an implementation of `AAST`, without special provision.

Fibrations

A module with a type component is already a family of modules indexed by types!

```
signature EQNAT = EQ where type t=nat
```

```
signature EQAST = EQ where type t=Ast.t
```

There is no need for a separate concept of family; it is definable already from hierarchy and interpretation.

Dependencies

Hierarchies are Σ types:

```
signature DICT = sig
  structure Key : ORDERED
  type α t
  val lookup : α t → Key.t → α
  ...
end
```

Types are propagated from substructure to rest of signature.

Dependencies

Parameterized modules are Π types:

```
functor Dict ( K : ORDERED ) :
    DICT where type Key.t=K.t
```

Types are propagated from argument to result signature.

Coherence

Coherence is imposed *a posteriori*, rather than *a priori*.

```
functor Ring
  (G : GROUP and M : MONOID sharing G.t = M.t): RING
```

The alternative is to factor out the underlying set, but this quickly becomes unwieldy.

Generativity

Each instance of a functor gives rise to distinct types.

```
functor NewAst() : AST = ...

structure Ast1 = NewAst()

structure Ast2 = NewAst()
```

Generativity: the types `Ast1.t` and `Ast2.t` are distinct.

Theory of Modules

The module system is battle-tested and hugely successful.

- Standard ML.
- O'Caml, with a minor variation on Standard ML modules.

And it *really* shines when reduced to its type-theoretic essence.

Static Phase

Kinds:

$$K ::= \mathbf{T} \mid \mathbf{S}(A) \mid \sum_{u::K_1} K_2 \mid \prod_{u::K_1} K_2$$

Constructors:

$$A ::= \mathbf{bool} \mid \cdots \mid \langle A_1, A_2 \rangle \mid \lambda_{u::K} A \mid \cdots \mid \ldots$$

The kind $\mathbf{S}(A)$ classifies the types definitionally equivalent to A.

- Type definitions.
- Instances of families.
- Coherence constraints.

(Static and) Dynamic Phase

Signatures:

$$\sigma ::= [\![K]\!] \mid (\!| A |\!) \mid \{ \sigma \} \mid \sum_{X:\sigma_1} \sigma_2 \mid \prod_{X:\sigma_1} \sigma_2$$

Glossary:

- $[\![K]\!]$: a constructor of kind K.
- $(\!| A |\!)$: a pure value of type A.
- $\{ \sigma \}$: a computation with effects yielding a module of type σ.

(Static and) Dynamic Phase

Pure modules:

$$M ::= X \mid [\![A]\!] \mid (\!| e |\!) \mid E \upharpoonright \sigma \mid \dots$$

Impure modules:

$$E ::= \mathbf{ret}(M) \mid \mathbf{bind}(M; X.E) \mid \dots$$

Components of Modules

Kinds:

$$K ::= \cdots \mid M \cdot \mathbf{st}$$

Expressions:

$$e ::= \cdots \mid M \cdot \mathbf{dy}$$

Projections are limited to pure modules of signature $[\![K]\!]$ or $(\!| A |\!)$.

Generativity: sealed modules are of monadic type, $\{ \sigma \}$.

Polymorphism, via Modules

Polymorphic functions may be viewed as functors:

$$\forall \alpha.\alpha \to \alpha \approx \prod_{X:[\![\,\mathbf{Ty}\,]\!]} X \cdot \mathbf{st} \to X \cdot \mathbf{st}$$

Scales naturally to constrained quantification: separation:

$$\forall \alpha\, \mathtt{EQ}(\alpha) \Rightarrow \alpha \to \alpha \approx \prod_{X:\mathbf{EQ}} X \cdot \mathbf{st} \to X \cdot \mathbf{st},$$

where

$$\mathbf{EQ} \approx \sum_{Y:[\![\,\mathbf{Ty}\,]\!]} (\!|\, Y \cdot \mathbf{st} \times Y \cdot \mathbf{st} \to \mathbf{bool} \,|\!)$$

Canonical Structures

Haskell-style type classes amount to generating canonical structures of a given signature.

- Designate certain functors as eligible for backchaining.
- Derive canonical structure by composing eligible functors.

Haskell-style type classes are but a mode of use of modules (you were expecting me to say that).

Abstract Types

Existential types are definable using sums and monadic sealing:

$$\exists \alpha{::}K.A \approx \sum_{X:[\![K]\!]} (\!| A |\!)$$

Packages are sealed pairs:

$$\textbf{pack}\, A\, \textbf{with}\, e\, \textbf{as}\, \exists \alpha{::}K.B \approx \langle A, e \rangle \upharpoonright \exists \alpha{::}K.A$$

Monadic structure ensures generative abstraction:

$$\textbf{open}\, M\, \textbf{as}\, \alpha{::}K\, \textbf{with}\, x{:}A\, \mathbf{E} \approx \textbf{bind}(M; X.[X \cdot \textbf{st}, X \cdot \textbf{dy}/\alpha, x]A)$$

Generative Abstraction

Concrete data types in ML are generative.

- Each elaboration "generates" a new type.
- Type classes + data types give rise to abstract types.

Concrete types are actually abstract types.

- Signature provides hooks to the pattern compiler.
- Canonical implementation from recursive types.
- Sealed to ensure generativity, even though effect-free.

Data Signatures

A data signature is the signature of a datatype:

```
signature NAT = data
  type t
  con zero : t
  con succ : t -> t
end

data structure Nat : NAT
```

A data structure is the canonical implementation of a data signature.

Data Signatures

Type-theoretically, **NAT** is the signature

$$\sum_{T:[\![\,\mathbf{Ty}\,]\!]} (T\cdot\mathbf{st})\times(T\cdot\mathbf{st}\to T\cdot\mathbf{st})\times(\forall\alpha.\,T\cdot\mathbf{st}\to\alpha\to(T\cdot\mathbf{st}\to\alpha)\to\alpha)$$

The canonical implementation is a module of this type.

- Recursive type formed from the constructors.
- Functions to create values of this type.
- Case-analysis operation to interface to pattern compiler.

Higher-Order Modules?

The type theoretic formulation provides a natural, but unreasonably weak, concept of higher-order module.

- Applicative functors cannot properly accommodate effects.
- Generative functors are too pessimistic.

There is a fundamental conflict between separate compilation and higher-order modules that has yet to be resolved (but see MacQueen, et al).

Recursive Modules?

Recursive, or cross-referential, modules are perenially popular.

- Conventional wisdom carried over from rudimentary languages.
- Certain examples are appealing, but the general case is messy.

There is a fundamental conflict between recursive modules and separate compilation that has yet to be resolved (but see Dreyer, et al).

Modules Matter Most

Modules are central to language design:

- For the obvious purpose of supporting decomposition.
- For less obvious purposes such as type classes, data types, polymorphism.

Dave first called attention to the importance of developing types for modularity in his 1986 POPL paper.

Much has come from that one idea.

Thanks, Dave!