Mixin' Up the ML Module System

Derek Dreyer and Andreas Rossberg

Max Planck Institute for Software Systems
Saarbrücken, Germany

ICFP 2008
Victoria, British Columbia
September 24, 2008

The ML Module System

Widely used feature of ML languages

Originally proposed by Dave MacQueen in 1984

- Developed further by Harper, Leroy, Lillibridge, Stone, Russo, *et al.*

Powerful support for:

- Namespace management
- Abstract data types
- Generic programming

Two Problems with the ML Module System

It is not sufficiently *expressive*.

Two Problems with the ML Module System

It is not sufficiently *expressive*.

It is overly *complex*.

Problem #1: Recursive Modules

One of the most requested extensions to ML.

Over 10 years of work on recursive modules

- Various problems solved, but a big one remains:

Problem #1: Recursive Modules

One of the most requested extensions to ML.

Over 10 years of work on recursive modules

- Various problems solved, but a big one remains:

Separate Compilation

Why is Separate Compilation Hard?

Signatures of mutually recursive modules `A` and `B` may be *recursively dependent*.

```
module A : sig
  type t
  val f : B.u -> A.t
end
and B : sig
  type u
  val g : A.t -> B.u
end
```

Why is Separate Compilation Hard?

Signatures of mutually recursive modules A and B may be *recursively dependent*.

```
module A : sig
  type t
  val f : B.u -> A.t
end
and B : sig
  type u
  val g : A.t -> B.u
end
```

Why is Separate Compilation Hard?

Signatures of mutually recursive modules `A` and `B` may be *recursively dependent*.

```
module A : sig
  type t
  val f : B.u -> A.t
end
and B : sig
  type u
  val g : A.t -> B.u
end
```

Why is Separate Compilation Hard?

ML's separate compilation mechanism is *the functor*.

```
functor Sep_A (X : SIG_B) :> SIG_A = ...
```

Why is Separate Compilation Hard?

ML's separate compilation mechanism is *the functor*.

```
functor Sep_A (X : SIG_B) :> SIG_A = ...
```

Problem: `SIG_B` depends on type components of `A`, which are not in scope.

Not obvious how to generalize functors to work in the recursive case.

Problem #2: Conceptual Complexity

We often present ML module system as just a (dependently-typed) λ-calculus at the module level:

- λ = Functors
- Records = Structures
- Record types = Signatures

But in reality…

The ML Module System in Reality

- Structure formation (`struct`)
- Structure inheritance (`open`)
- Signature formation (`sig`)
- Signature inheritance (`include`)
- Transparent type specifications (`type t =` *typ*)
- Opaque type specifications (`type t`)
- Value specifications (`val v :` *typ*)
- Signature refinement (`where type` / `with type`)
- Sharing constraints (`sharing type`)
- Signature bindings (`signature`)
- Functor abstraction (`functor`)
- Functor application (( ))
- Transparent signature ascription (:)
- Opaque signature ascription (:>)
- Local definitions (`let` / `local`)
- Recursive structures (`struct rec`)
- Recursively dependent signatures (`sig rec`)

Mixin Modules

Originally proposed by Bracha & Lindstrom (1992)

- Module = record with imports and exports.
- Two modules can be *merged*, with the exports of each one filling in the imports of the other.

Mixin Modules

Originally proposed by Bracha & Lindstrom (1992)

- Module = record with imports and exports.
- Two modules can be *merged*, with the exports of each one filling in the imports of the other.

Advantage of mixin modules:

- Mixin merging *is* recursive linking.

Mixin Modules

Originally proposed by Bracha & Lindstrom (1992)

- Module = record with imports and exports.
- Two modules can be *merged*, with the exports of each one filling in the imports of the other.

Advantage of mixin modules:

- Mixin merging *is* recursive linking.

Disadvantage of mixin modules:

- No type components, hence no type abstraction.

Combining Mixin Modules and ML-Style Modules

More recent descendants of mixin modules do include support for type components.

- *Units:* Flatt-Felleisen (PLDI'98), Owens-Flatt (ICFP'06)
- *Recursive DLLs:* Duggan (TOPLAS'02)
- *Scala:* Odersky et al. (OOPSLA'05, ECOOP'03)

Combining Mixin Modules and ML-Style Modules

More recent descendants of mixin modules do include support for type components.

- *Units:* Flatt-Felleisen (PLDI'98), Owens-Flatt (ICFP'06)
- *Recursive DLLs:* Duggan (TOPLAS'02)
- *Scala:* Odersky et al. (OOPSLA'05, ECOOP'03)

But they do not subsume the ML module system.

- Direct encodings of several key ML features are verbose and/or impossible.

Contribution of the Paper

Our attempt to synthesize ML modules and mixin modules: MixML

Very simple, minimalist design

Generalizes the ML module system

- Supports separately compilable recursive modules, in addition to all the old features of ML modules

Simplifies the ML module system

- Leverages mixin composition to give a unifying account of superficially distinct features of ML modules

MixML: The Basic Idea

MixML modules synthesize ML's structure and signature languages into one.

MixML: The Basic Idea

MixML modules synthesize ML's structure and signature languages into one.

Consequences:

- ML structures and signatures are endpoints on a spectrum of MixML modules.

MixML: The Basic Idea

MixML modules synthesize ML's structure and signature languages into one.

Consequences:

- ML structures and signatures are endpoints on a spectrum of MixML modules.
- Signatures and structures (and mixtures of both) are composed using *the exact same constructs*.

The MixML Module Language

$$
\begin{array}{rcll}
\mathit{mod} & ::= & X & \text{(variable)} \\
& \mid & \{\} & \text{(empty)} \\
& \mid & [\mathit{exp}] \quad \mid \quad [: \mathit{typ}] & \text{(term)} \\
& \mid & [\mathit{typ}] \quad \mid \quad [: \mathit{kind}] & \text{(type)} \\
& \mid & \{l = \mathit{mod}\} \quad \mid \quad \mathit{mod}.l & \text{(namespaces)} \\
& \mid & (X = \mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 & \text{(linking)} \\
& \mid & (X = \mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2 & \text{(sealing)} \\
& \mid & [\mathit{mod}] \quad \mid \quad \texttt{new}\ \mathit{mod} & \text{(units)}
\end{array}
$$

Some Useful Derived Forms

- Structure formation (`struct`)
- Structure inheritance (`open`)
- Signature formation (`sig`)
- Signature inheritance (`include`)
- Transparent type specifications (`type t =` *typ*)
- Opaque type specifications (`type t`)
- Value specifications (`val v :` *typ*)
- Signature refinement (`where type` / `with type`)
- Sharing constraints (`sharing type`)
- Signature bindings (`signature`)
- Functor abstraction (`functor`)
- Functor application (( ))
- Transparent signature ascription (:)
- Opaque signature ascription (:>)
- Local definitions (`let` / `local`)
- Recursive structures (`struct rec`)
- Recursively dependent signatures (`sig rec`)

ML Structure Example

We can encode the structure

```
struct
  type t = int
  val v = λx.x+3
end
```

as

```
{
  t = [int],
  v = [λx.x+3]
}
```

ML Signature Example

We can encode the signature

```
sig
  type t
  val v : t -> t
end
```

as

$$\left\{\begin{array}{l} \texttt{t = [:}\Omega\texttt{],} \\ \texttt{v = [:t -> t]} \end{array}\right\}$$

ML Signature Example

We can encode the transparent signature

```
sig
  type t = int
  val v : t -> t
end
```

as

```
{
  t = [int],
  v = [:t -> t]
}
```

The MixML Module Language

$$
\begin{array}{rcll}
mod & ::= & X & \text{(variable)} \\
& | & \{\} & \text{(empty)} \\
& | & [exp] \quad | \quad [:\, typ] & \text{(term)} \\
& | & [typ] \quad | \quad [:\, kind] & \text{(type)} \\
& | & \{l = mod\} \quad | \quad mod.l & \text{(namespaces)} \\
& | & (X = mod_1)\ \texttt{with}\ mod_2 & \text{(linking)} \\
& | & (X = mod_1)\ \texttt{seals}\ mod_2 & \text{(sealing)} \\
& | & [mod] \quad | \quad \texttt{new}\ mod & \text{(units)}
\end{array}
$$

Signature Refinement

sig `with type t = int`

Signature Refinement

`(X =` *sig*`) with {t = [int]}`

Signature Refinement

`(X = `*sig*`) with {t = [u]}`

Signature Refinement

(X = *sig*) with {t = [X.u]}

Recursive Modules

$$\texttt{rec}\,(\mathrm{X} : \mathit{sig})\,\mathit{mod}$$

Recursive Modules

$$\texttt{rec}\,(\mathrm{X} : \mathit{sig})\,\mathit{mod}$$

$$\stackrel{\text{def}}{=}$$

$$(\mathrm{X} = \mathit{sig})\ \texttt{with}\ \mathit{mod}$$

The MixML Module Language

$$
\begin{array}{rcll}
mod & ::= & X & \text{(variable)} \\
& \mid & \{\} & \text{(empty)} \\
& \mid & [exp] \quad \mid \quad [:typ] & \text{(term)} \\
& \mid & [typ] \quad \mid \quad [:kind] & \text{(type)} \\
& \mid & \{l = mod\} \quad \mid \quad mod.l & \text{(namespaces)} \\
& \mid & (X = mod_1)\ \texttt{with}\ mod_2 & \text{(linking)} \\
& \mid & (X = mod_1)\ \texttt{seals}\ mod_2 & \text{(sealing)} \\
& \mid & [mod] \quad \mid \quad \texttt{new}\ mod & \text{(units)}
\end{array}
$$

Separate Compilation via "Units"

We can break mutually recursive modules

$$(\mathrm{X} = sig)\ \texttt{with}\ \{\texttt{A} = mod_{\texttt{A}}, \texttt{B} = mod_{\texttt{B}}\}$$

into separately compiled *units*:

$$\texttt{U}_{\texttt{A}} = [(\mathrm{X} = sig)\ \texttt{with}\ \{\texttt{A} = mod_{\texttt{A}}\}]$$

$$\texttt{U}_{\texttt{B}} = [(\mathrm{X} = sig)\ \texttt{with}\ \{\texttt{B} = mod_{\texttt{B}}\}]$$

and link them later on by writing:

$$\texttt{new}\ \texttt{U}_{\texttt{A}}\ \texttt{with}\ \texttt{new}\ \texttt{U}_{\texttt{B}}$$

Improvements Over Previous Mixin Module Systems

Orthogonality

- No monolithic mixin construct (`import` Γ_i `export` Γ_e Ds).

Hierarchical composability (aka "deep mixing")

- Previous mixin modules only allow flat namespaces.

Unifying linking and binding: $(\mathrm{X} = mod_1)$ `with` mod_2

- Very useful, e.g. signature refinement, recursive modules.

"Double vision" problem

- Problem with interaction of recursion and type abstraction.
- We generalize (Dreyer 07) to handle "cross-eyed" version.

See the paper for…

- Tour of MixML by example
- Informal explanation of typing issues
- Full formalization
- Higher-order module extension
- Related work
- Future work
- Link to prototype implementation