Principles of Programming Languages

Lecture 08 Modules

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Gabriel Radanne
University of Freiburg, Germany
radanne@informatik.uni-freiburg.de

18 June 2018

In the previous episode: Modules

- Modules
  - Abstract data type packaged with operators.
  - Can import other (known) modules.
  - Can be parameterized with (unknown) modules.
- Parametric modules
  - Functions over existential types.

Example: Module with two Implementations

```
type PointWRT[PointRep] =
  {mkPoint: (Real × Real) → PointRep,
   x_coord: PointRep → Real,
   y_coord: PointRep → Real}

type Point = ∃ PointRep. PointWRT[PointRep]

value cartesianPointOps =
  {mkpoint = fun(x: Real, y: Real) (x,y),
   x_coord = fun(p: Real × Real) fst(p),
   y_coord = fun(p: Real × Real) snd(p)}

value cartesianPointPackage: Point =
  pack[PointRep = Real × Real  in PointWRT[PointRep]]
    (cartesianPointOps)

value polarPointPackage: Point =
  pack[PointRep = Real × Real in PointWRT[PointRep]]
    {mkpoint = fun(x: Real, y: Real)  ... ,
     x_coord = fun(p: Real × Real)    ... ,
     y_coord = fun(p: Real × Real)    ... }
```

Parametric Modules

```
type ExtendedPointWRT[PointRep] =
  PointWRT[PointRep] &
  {add: (PointRep × PointRep) → PointRep}

type ExtendedPoint =
  ∃ PointRep. ExtendedPointWRT[PointRep]

value extendPointPackage =
  fun(pointPackage: Point)
  open pointPackage as p[PointRep] in
    pack[PointRep' = PointRep in ExtendedPointWRT[PointRep']]
    p & {add = fun(a: PointRep, b: PointRep)
                   p.mkpoint(p.x_coord(a)+p.x_coord(b),
                             p.y_coord(a)+p.y_coord(b))}

value extendedCartesianPointPackage =
  extendPointPackage(cartesianPointPackage)
```

A Circle Package

```
type CircleWRT2[CircleRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkcircle: (PointRep × Real) → CircleRep,
   center: CircleRep → PointRep, ... }

type CircleWRT1[PointRep] =
   ∃ CircleRep. CircleWRT2[CircleRep, PointRep]

type Circle =
   ∃ PointRep. CircleWRT1[PointRep]

type CircleModule =
   ∀ PointRep.
   PointWRT[PointRep] → CircleWRT1[PointRep]

value circleModule: CircleModule =
    all[PointRep]
      fun(p: PointWRT[PointRep])
        pack[CircleRep = PointRep × Real
           in CircleWRT2[CircleRep,PointRep]]
       {pointPackage = p,
        mkcircle = fun(m: PointRep, r: Real)(m, r) ... }

value cartesianCirclePackage =
   open CartesianPointPackage as p[Rep] in
      pack[PointRep = Rep in CircleWRT1[PointRep]]
        circleModule[Rep](p)

open cartesianCirclePackage as c0[PointRep] in
open c0 as c[CircleRep] in
   ... c.mkcircle(c.pointPackage.mkpoint(3, 4), 5) ...
```

A Rectangle Package

```
type RectWRT2[RectRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkrect: (PointRep × PointRep) → RectRep, ... }

type RectWRT1[PointRep] =
   ∃ RectRep. RectWRT2[RectRep, PointRep]

type Rect =
   ∃ PointRep. RectWRT1[PointRep]

type RectModule = ∀ PointRep.
   PointWRT[PointRep] → RectWRT1[PointRep]

value rectModule: RectModule =
    all[PointRep]
      fun(p: PointWRT[PointRep])
        pack[PointRep' = PointRep
            in RectWRT1[PointRep']]
       {pointPackage = p,
        mkrect = fun(tl: PointRep, br: PointRep) ... }
```

A Figures Package

```
type FiguresWRT3[RectRep, CircleRep, PointRep] -
  {circlePackage: CircleWRT[CircleRep, PointRep],
   rectPackage: RectWRT[RectRep, PointRep],
   boundingRect: CircleRep → RectRep}

type FiguresWRT1[PointRep] =
   ∃ RectRep. ∃ CircleRep.
      FiguresWRT3[RectRep, CircleRep, PointRep]

type Figures =
   ∃ PointRep. FiguresWRT1[PointRep]

type FiguresModule = ∀ PointRep.
   PointWRT[PointRep] → FiguresWRT1[PointRep]

value figuresModule: FIguresModule =
    all[PointRep]
       fun(p: PointWRT[PointRep])
          pack[PointRep' = PointRep
             in FiguresWRT1[PointRep]]
          open circleModule[PointRep](p) as c[CircleRep] in
             open rectModule[PointRep](p) as r[RectRep] in
               {circlePackage = c, ... }
```

This is a *low-level* view of modules. Let's take a step back to more familiar grounds.

This is a *low-level* view of modules. Let's take a step back to more familiar grounds.
There are as many module systems as there are programming languages.

Modules in C?

No notion of modules in C or C++.
Namespacing is done by prefixes, imports are done with `include`.

```
#include <stdlib.h>

int package_function ( ... );
```

- No encapsulation
- Separate compilation doesn't really work
- Order/dependencies doesn't matter .... but you can make the preprocessor loop.

Modules in Java

- *packages* provides namespacing, imports and separate compilation
- *classes* provides encapsulation
- Rich class language (interfaces, inheritance, generics, . . .)
- No cycles!

```
import java.util.ArrayList;

public class Algo {
  public int thing;
  private int hidden;
  protected int local;

  public int do_thing ( ... ) {
    ...
  }
}
```

Modules in Javascript

Recent extension (Ecmascript 6).

```
import * as lib from 'lib';

export function dothing() {
  ...
}

export { foo as myFoo, bar } from 'src/other_module';
```

- Namespacing and imports
- Little bit of encapsulation (can hide functions)
- Can make cycles

Modules in Rust

Rust has a proper notion of modules

- Namespacing and imports
- Encapsulations for type and functions
- Rich manipulation of modules (can open and rebind them, pointed access, ...)

```
pub mod network {
  pub fn connect() { ... }
  fn internal_fun() { ... }
  pub mod server { ... }
}
```

```
use network;

fn main() {
    server::thing()
}
```

What we expect from a module system

- Structuring programs

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces

What we expect from a module system

- Structuring programs
    - Cut programs in smaller, easier to manage, pieces
    - Manage dependencies

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces
  - Manage dependencies
- Modularity: Make smaller pieces of program compose.

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces
  - Manage dependencies
- Modularity: Make smaller pieces of program compose.
- Encapsulation: Hide the implementation details of a module to the outside words.

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces
  - Manage dependencies
- Modularity: Make smaller pieces of program compose.
- Encapsulation: Hide the implementation details of a module to the outside words.

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces
  - Manage dependencies
- Modularity: Make smaller pieces of program compose.
- Encapsulation: Hide the implementation details of a module to the outside words.

Additional worthwhile goals:

- Genericity: Modules that are parameterized by other modules

What we expect from a module system

- Structuring programs
  - Cut programs in smaller, easier to manage, pieces
  - Manage dependencies
- Modularity: Make smaller pieces of program compose.
- Encapsulation: Hide the implementation details of a module to the outside words.

Additional worthwhile goals:

- Genericity: Modules that are parameterized by other modules
- Extended modularity: Separate compilation

History of Modular programming

- Early prototype as Algol Extensions in 1970

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.
  - Separate compilation

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.
  - Separate compilation
  - Encapsulation

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.
  - Separate compilation
  - Encapsulation
- The ML family of programming languages (Standard ML, OCaml).
  The most complete module language currently.

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.
  - Separate compilation
  - Encapsulation
- The ML family of programming languages (Standard ML, OCaml).
  The most complete module language currently.
- Popularization in many languages (but absence in many others ...)

History of Modular programming

- Early prototype as Algol Extensions in 1970
- Modula 1, 2 and 3 (1975-1980s): The first proper module language
  - Pointed notation for access `M.x`.
  - Separate compilation
  - Encapsulation
- **The ML family of programming languages (Standard ML, OCaml). The most complete module language currently.**
- Popularization in many languages (but absence in many others ...)

Plan

1. In ML languages
2. Formalization
   - Reduction rules
   - Type checking
3. Results on modules

Modules in ML

Demo!

Plan

Grammar of a language with modules

UNI FREIBURG

Module Expressions

$$
\begin{aligned}
M ::= {} & X_i \mid p.X && \text{(Variables)}\\
\mid {} & (M : \mathcal{M}) && \text{(Type constraint)}\\
\mid {} & M_1(M_2) && \text{(Functor application)}\\
\mid {} & \texttt{functor}(X_i : \mathcal{M})M && \text{(Functor)}\\
\mid {} & \texttt{struct } S \texttt{ end} && \text{(Structure)}
\end{aligned}
$$

Structure body

$$S ::= \varepsilon \mid D; S$$

Structure components

$$
\begin{aligned}
D ::= {} & \texttt{let } x_i = e && \text{(Values)}\\
\mid {} & \texttt{type } \mathrm{t_i} = \tau && \text{(Types)}\\
\mid {} & \texttt{module } X_i = M && \text{(Modules)}
\end{aligned}
$$

Programs

$$P ::= \texttt{prog } S \texttt{ end}$$

Grammar of a language with modules

Module types

$$\mathcal{M} ::= \texttt{sig}\ \mathcal{S}\ \texttt{end} \qquad \text{(Signature)}$$
$$\mid \texttt{functor}(X_i : \mathcal{M}_1)\mathcal{M}_2 \qquad \text{(Functor)}$$

Signature body

$$\mathcal{S} ::= \varepsilon \mid \mathcal{D}; \mathcal{S}$$

Signature components

$$\mathcal{D} ::= \texttt{val}\ x_i : \tau \qquad \text{(Values)}$$
$$\mid \texttt{type}\ \mathrm{t_i} = \tau \qquad \text{(Types)}$$
$$\mid \texttt{type}\ \mathrm{t_i} \qquad \text{(Abstract types)}$$
$$\mid \texttt{module}\ X_i : \mathcal{M} \qquad \text{(Modules)}$$

Values

Modules

$$V ::= {V_b}^* \quad \text{(Structure)}$$
$$\mid \ \texttt{functor}(\rho)(X_i : \mathcal{M})M \quad \text{(Closures)}$$

Bindings

$$V_b ::= \{x_i \mapsto v\} \quad \text{(Values)}$$
$$\mid \ \{X_i \mapsto V\} \quad \text{(Modules)}$$

Plan

Reduction rules – Declarations

ValDecl

$$\frac{e \xrightarrow{\rho} v \qquad S \xrightarrow{\rho + \{x_i \mapsto v\}} V_s}{(\texttt{let } x_i = e; S) \xrightarrow{\rho} \{x \mapsto v\} + V_s}$$

Reduction rules – Declarations

ValDecl

$$\frac{e \xrightarrow{\rho} v \qquad S \xrightarrow{\rho+\{x_i \mapsto v\}} V_s}{(\texttt{let } x_i = e; S) \xrightarrow{\rho} \{x \mapsto v\} + V_s}$$

TypeDecl

$$\frac{S \xrightarrow{\rho} V_s}{(\texttt{type } t_i = \tau; S) \xrightarrow{\rho} V_s}$$

ModuleDecl

$$\frac{M \xrightarrow{\rho} V \qquad S \xrightarrow{\rho+\{X_i \mapsto V\}} V_s}{(\texttt{module } X_i = M; S) \xrightarrow{\rho} \{X \mapsto V\} + V_s}$$

Reduction rules – Declarations

ValDecl

$$\frac{e \xRightarrow{\rho} v \qquad S \xRightarrow{\rho + \{\mathrm{x_i} \mapsto \mathrm{v}\}} V_s}{(\texttt{let } x_i = e; S) \xRightarrow{\rho} \{x \mapsto v\} + V_s}$$

TypeDecl

$$\frac{S \xRightarrow{\rho} V_s}{(\texttt{type } \mathrm{t_i} = \tau; S) \xRightarrow{\rho} V_s}$$

ModuleDecl

$$\frac{M \xRightarrow{\rho} V \qquad S \xRightarrow{\rho + \{\mathrm{X_i} \mapsto \mathrm{V}\}} V_s}{(\texttt{module } X_i = M; S) \xRightarrow{\rho} \{X \mapsto V\} + V_s}$$

Struct

$$\frac{S \xRightarrow{\rho} V_s}{(\texttt{struct } S \texttt{ end}) \xRightarrow{\rho} V_s}$$

EmptyStruct

$$\frac{}{\varepsilon \xRightarrow{\rho} \{\}}$$

Reduction rules – Variables and type constraints

ModVar

$$\frac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow} V}$$

QualModVar

$$\frac{p \overset{\rho}{\Longrightarrow} V' \qquad V'(X) = V}{p.X \overset{\rho}{\Longrightarrow} V}$$

Reduction rules – Variables and type constraints

ModVar
$$\frac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow} V}$$

QualModVar
$$\frac{p \overset{\rho}{\Longrightarrow} V' \qquad V'(X) = V}{p.X \overset{\rho}{\Longrightarrow} V}$$

ModConstr
$$\frac{M \overset{\rho}{\Longrightarrow} V}{(M : \mathcal{M}) \overset{\rho}{\Longrightarrow} V}$$

Reduction rules – Functors

ModClosure

$$\frac{}{\texttt{functor}(X : \mathcal{M})M \xRightarrow{\rho} \texttt{functor}(\rho)(X : \mathcal{M})M}$$

ModBeta

$$\frac{M \xRightarrow{\rho} \texttt{functor}(\rho')(X : \mathcal{M})M_f \qquad M' \xRightarrow{\rho} V' \qquad M_f \xRightarrow{\rho' + \{X \mapsto V'\}} V''}{M(M') \xRightarrow{\rho} V''}$$

Reduction rules – Programs

Program

$$\frac{S \xRightarrow{\rho} V_s}{\texttt{prog}\ S\ \texttt{end} \xRightarrow{\rho} V_s(\texttt{return})}$$

Example reduction

$$P \equiv \begin{pmatrix} \texttt{prog} \\ \quad \texttt{module } X = \texttt{struct let } a = 3 \texttt{ end} \\ \quad \texttt{let return} = X.a \\ \texttt{end} \end{pmatrix}$$

Example reduction

$$
\text{Program}\;\frac{
\text{ModuleDecl}\;\dfrac{
\text{Struct}\;\dfrac{\dfrac{\vdots}{\texttt{let}\ a = 3 \Rightarrow \{a \mapsto 3\}}}{\begin{pmatrix}\texttt{struct} \\ \quad \texttt{let}\ a = 3 \\ \texttt{end}\end{pmatrix} \Rightarrow \{a \mapsto 3\}}
\qquad
\dfrac{\dfrac{\text{ModVar}\;\dfrac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow} V \equiv \{a \mapsto 3\}} \qquad V(a) = 3}{X.a \xrightarrow{\{X \mapsto \{a \mapsto 3\}\}} 3}\;\text{QualModVar}}{\texttt{let return} = X.a \xrightarrow{\{X \mapsto \{a \mapsto 3\}\}} \{\texttt{return} \mapsto 3\}}\;\text{ValDecl}
}{
\begin{pmatrix}\texttt{module}\ X = \texttt{struct let}\ a = 3\ \texttt{end} \\ \texttt{let return} = X.a\end{pmatrix} \Rightarrow \{X \mapsto \{a \mapsto 3\}\} + \{\texttt{return} \mapsto 3\}
}\;\text{ModuleDecl}
}{P \Rightarrow 3}\;\text{Program}
$$

Plan

Type checking

Two relations:

$$\Gamma \blacktriangleright M : \mathcal{M} \ : \ \text{The module } M \text{ is of type } \mathcal{M} \text{ in } \Gamma.$$

$$\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}' \ : \ \text{The module type } \mathcal{M} \text{ is a subtype of } \mathcal{M}' \text{ in } \Gamma.$$

Type checking – Declarations

Declarations follow the same flow as the dynamic semantics:

$$\frac{\Gamma \triangleright e : \tau \qquad x_i \notin \text{BoundVars}(\Gamma) \qquad \Gamma; (\texttt{val}\ x_i : \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{let}\ x_i = e; s) : (\texttt{val}\ x_i : \tau; \mathcal{S})}$$

Type checking – Declarations

Declarations follow the same flow as the dynamic semantics:

$$\frac{\Gamma \triangleright e : \tau \qquad x_i \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\texttt{val}\ x_i : \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{let}\ x_i = e; s) : (\texttt{val}\ x_i : \tau; \mathcal{S})}$$

$$\frac{\mathrm{t_i} \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\texttt{type}\ \mathrm{t_i} = \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i} = \tau; s) : (\texttt{type}\ \mathrm{t_i} = \tau; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright M : \mathcal{M} \qquad X_i \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\texttt{module}\ X_i : \mathcal{M}) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{module}\ X_i = M; s) : (\texttt{module}\ X_i : \mathcal{M}; \mathcal{S})}$$

Type checking – Declarations

Declarations follow the same flow as the dynamic semantics:

$$\frac{\Gamma \triangleright e : \tau \qquad x_i \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\mathtt{val}\ x_i : \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\mathtt{let}\ x_i = e; s) : (\mathtt{val}\ x_i : \tau; \mathcal{S})}$$

$$\frac{\mathrm{t_i} \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\mathtt{type}\ \mathrm{t_i} = \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\mathtt{type}\ \mathrm{t_i} = \tau; s) : (\mathtt{type}\ \mathrm{t_i} = \tau; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright M : \mathcal{M} \qquad X_i \notin \mathrm{BoundVars}(\Gamma) \qquad \Gamma; (\mathtt{module}\ X_i : \mathcal{M}) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\mathtt{module}\ X_i = M; s) : (\mathtt{module}\ X_i : \mathcal{M}; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright \mathtt{struct}\ S\ \mathtt{end} : \mathtt{sig}\ \mathcal{S}\ \mathtt{end}} \qquad \frac{}{\Gamma \blacktriangleright \varepsilon : \varepsilon}$$

Type checking – Variables

Normal variables are typechecked as usual, but *qualified* variables are surprisingly complicated.

ModVar

$$\frac{(\texttt{module}\ X_i : \mathcal{M}) \in \Gamma}{\Gamma \blacktriangleright X_i : \mathcal{M}}$$

QualModVar

$$\frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{module}\ X_i : \mathcal{M}; \mathcal{S}_2\ \texttt{end})}{\Gamma \blacktriangleright p.X : \mathcal{M}[n_i \mapsto p.n \mid n_i \in \text{BoundVars}(\mathcal{S}_1)]}$$

Type checking – Variables – Example

An example of type checking for qualified accesses. Given the module X, we wish to typecheck $X.a$

$$X : \texttt{sig ... type } t; \texttt{ val } a : t; \texttt{ ... end}$$

Type checking – Variables – Example

An example of type checking for qualified accesses. Given the module X, we wish to typecheck $X.a$

$$X : \texttt{sig} \ \ldots \ \texttt{type}\ t;\ \texttt{val}\ a : t;\ \ldots\ \texttt{end}$$

We substitute t by X.t in the type of the a with the QualVar rule

$$\text{QualVar}\ \dfrac{\text{ModVar}\ \dfrac{(\texttt{module}\ X : \texttt{sig}\ \ldots\ \texttt{type}\ t;\ \texttt{val}\ a : t;\ \ldots\ \texttt{end}) \in \Gamma}{\Gamma \blacktriangleright X : \texttt{sig}\ \texttt{type}\ t;\ \texttt{val}\ a : t\ \texttt{end}}}{\Gamma \triangleright X.a : \mathrm{X.t}}\ \text{with}\ X.t = t[t \mapsto X.t]$$

Type checking – Interlude in type equalities

```
module Showable : sig
  type t
  val show : t → string
end
module Elt : sig
  type t = Showable.t
  val v : elt
end
```

```
module F
  (E : sig type t val v : t end)
  (S : sig type t = E.t val show : t → string end)
 = struct
  let s = (S.show E.v)
end
module X = F(Elt)(Showable)
```

Type checking – Strengthening

"Strengthening" add new type equalities to existing modules

Strength

$$\frac{\Gamma \blacktriangleright p : \mathcal{M}}{\Gamma \blacktriangleright p : \mathcal{M}/p}$$

Type checking – Strengthening

"Strengthening" add new type equalities to existing modules

Strength

$$\frac{\Gamma \blacktriangleright p : \mathcal{M}}{\Gamma \blacktriangleright p : \mathcal{M}/p}$$

$$\begin{aligned}
\varepsilon/p &= \varepsilon \\
(\texttt{sig}\ \mathcal{S}\ \texttt{end})/p &= \texttt{sig}\ \mathcal{S}/p\ \texttt{end} \\
(\texttt{module}\ X_i = \mathcal{M}; \mathcal{S})/p &= \texttt{module}\ X_i = \mathcal{M}/p; \mathcal{S}/p \\
(\texttt{type}\ \mathrm{t_i} = \tau; \mathcal{S})/p &= \texttt{type}\ \mathrm{t_i} = p.t; \mathcal{S}/p \\
(\texttt{type}\ \mathrm{t_i}; \mathcal{S})/p &= \texttt{type}\ \mathrm{t_i} = p.t; \mathcal{S}/p \\
(\texttt{val}\ x_i : \tau; \mathcal{S})/p &= \texttt{val}\ x_i : \tau; \mathcal{S}/p \\
(\texttt{functor}(X_i : \mathcal{M})\mathcal{M}')/p &= \texttt{functor}(X_i : \mathcal{M})(\mathcal{M}'/p(X_i))
\end{aligned}$$

Type checking – Functors

Functors are typechecked mostly like lambdas:

$$\frac{\Gamma \blacktriangleright M_1 : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}' \qquad \Gamma \blacktriangleright M_2 : \mathcal{M}}{\Gamma \blacktriangleright M_1(M_2) : \mathcal{M}'[X_i \mapsto M_2]}$$

$$\frac{X_i \notin \text{BoundVars}(\Gamma) \qquad \Gamma; (\texttt{module}\ X_i : \mathcal{M}) \blacktriangleright M : \mathcal{M}'}{\Gamma \blacktriangleright \texttt{functor}(X_i : \mathcal{M})M : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}'}$$

Type checking – Module inclusion?

We are not done!
These rules does not allow use to hide fields or to abstract types. We need additional rules for module inclusions:

```
module type S = sig
  type t
end

module X : S = struct
  type t = int
  let x = 3
  (* .. *)
end
```

Type checking – Module inclusion

Inclusion on type declaration can take many forms:

$$\frac{\Gamma \triangleright \tau_1 \approx \tau_2}{\Gamma \blacktriangleright (\texttt{type}\ t_i = \tau_1) <: (\texttt{type}\ t_i = \tau_2)} \qquad \frac{}{\Gamma \blacktriangleright (\texttt{type}\ t_i) <: (\texttt{type}\ t_i)}$$

$$\frac{\Gamma \triangleright t_i \approx \tau}{\Gamma \blacktriangleright (\texttt{type}\ t_i) <: (\texttt{type}\ t_i = \tau)} \qquad \frac{}{\Gamma \blacktriangleright (\texttt{type}\ t_i = \tau_1) <: (\texttt{type}\ t_i)}$$

Type checking – Module inclusion

This rules allows to take a subset of the fields, and reorder them:

SubStruct

$$\frac{\pi : [1; m] \to [1; n] \qquad \forall i \in [1; m],\ \Gamma ; \mathcal{D}_1 ; \ldots ; \mathcal{D}_n \blacktriangleright \mathcal{D}_{\pi(i)} <: \mathcal{D}'_i}{\Gamma \blacktriangleright (\texttt{sig}\ \mathcal{D}_1 ; \ldots ; \mathcal{D}_n\ \texttt{end}) <: (\texttt{sig}\ \mathcal{D}'_1 ; \ldots ; \mathcal{D}'_m\ \texttt{end})}$$

Plan

Results on modules

We of course want usual soundness results on modules, but we also want to *prove* encapsulation and modularity.
"encapsulation" and "modularity" are big words without formal meaning. Let's try to make them more precise

Modularity

One way to look at modularity: We can typecheck each module without knowing the *implementation* of rest of the program.

Modularity

One way to look at modularity: We can typecheck each module without knowing the *implementation* of rest of the program.

Theorem (Incremental Typechecking)

Given a list of module declarations that form a typed program, there exists an order such that each module can be typechecked with only knowledge of the type of the previous modules.
More formally, given a list of n declarations D_i and a signature $\mathcal{S}$ such that

$$\blacktriangleright (D_1; \ldots; D_n) : \mathcal{S}$$

then there exists n definitions $\mathcal{D}_i$ and a permutation π such that

$$\forall i < n,\ \mathcal{D}_1; \ldots; \mathcal{D}_i \blacktriangleright D_{i+1} : \mathcal{D}_{i+1} \qquad \blacktriangleright \mathcal{D}_{\pi(1)}; \ldots; \mathcal{D}_{\pi(n)} <: \mathcal{S}$$

Encapsulation

Encapsulation means that the if I provide two modules that have the same type, the outside should not be able to differentiate them.

Encapsulation

Encapsulation means that the if I provide two modules that have the same type, the outside should not be able to differentiate them.

Theorem (Representation Independence)

Let M_1 and M_2 be two closed module expressions and $\mathcal{M}$ be a module type. Assume that $\mathcal{M}$ is a principal type for M_1 and for M_2 in the empty environment. Then, for all program contexts $C[\,]$ the program $C[M_1]$ is well-typed if and only if $C[M_2]$ is, and if so, $[\![C[M_1]]\!] = [\![C[M_1]]\!]$.

Back to existential and universal types

Modules can be translated into existential packs:

```
type PointWRT[PointRep] =
  {mkPoint: (Real × Real) → PointRep,
   x_coord: PointRep → Real,
   y_coord: PointRep → Real}

type Point = ∃ PointRep. PointWRT[PointRep]

value cartesianPointOps =
  {mkpoint = fun(x: Real, y: Real) (x,y),
   x_coord = fun(p: Real × Real) fst(p),
   y_coord = fun(p: Real × Real) snd(p)}

value cartesianPointPackage: Point =
  pack[PointRep = Real × Real  in
  PointWRT[PointRep]] (cartesianPointOps)

value polarPointPackage: Point =
  pack[PointRep = Real × Real in
  PointWRT[PointRep]]
    {mkpoint = fun(x: Real, y: Real)  ... ,
     x_coord = fun(p: Real × Real)    ... ,
     y_coord = fun(p: Real × Real)    ... }
```

```
module type Point = sig
  type t
  val mkPoint : Real × Real → PointRep
  val x_coord: PointRep → Real
  val y_coord: PointRep → Real
end

module cartesianPoint : Point =
  let mkpoint = fun(x: Real, y: Real) (x,y)
  let x_coord = fun(p: Real × Real) fst(p)
  let y_coord = fun(p: Real × Real) snd(p)
end

module polarPoint : Point =
  let mkpoint = fun(x: Real, y: Real)  ...
  let x_coord = fun(p: Real × Real)  ...
  let y_coord = fun(p: Real × Real)  ...
end
```

Conclusion

We have seen that modules as existential and universal-packs can be better expressed using proper module constructs. This can account for encapsulation and modularity, and many additional features such as separate compilation.

We only saw an example of a module language from the ML family. There are many different module systems.