Objective Caml:

the language and its type system

Part 3: the module system

Goals of modular programming

- Slice the program into pieces that can be type-checked and compiled independently.

  Make large programs compilable.

- Express the structure of the program and facilitate reuse of certain parts.

  Make large programs understandable.

ML modules

- The module system is a small typed functional language.
- It operates on collections of definitions (of values, types, ...) from the core ML language.
- Their types are collections of declarations and type specifications (types of values, declarations of types).
- Nested modules (as the only data structure).
- Functions (functors) and functor application.

Largely independent from the core language.

Base modules (structures)

Structures: collections of definitions.

`struct` *definition* ... *definition* `end`

Definitions can be any of the following:

definitions of values and functions	`let` $x = valexpr$
definitions of types	`type` $t = typexpr$
definitions of exceptions	`exception` E [`of` $type$]
definitions of classes	`class` $C = classexpr$
definitions of sub-modules	`module` $X = modulexpr$
definitions of module types	`module type` $S = modtype$

Naming a module

A module is named with the `module` binding.

```
module S = struct ... end
```

Example:

```
module S =
    struct
        type t = int
        let x = 0
        let f x = x+1
    end
```

Using a module

Components of a module are referenced with the dot notation *module*.*component*

```
... (S.f S.x : S.t) ...
```

Alternative: the directive `open` *module* allows to omit the prefix and the dot:

```
open S
... (f x : t) ...
```

Nested modules

A module can be a component of another module:

```
module T =
    struct
        module R = struct let x = 0 end
        let y = R.x + 1
    end
```

The dot notation and the `open` directive extend naturally:

```
T.R.x

open T.R
```

Types of base modules (signatures)

Signatures: collections of (type) specifications.

`sig` *specification* ... *specification* `end`

specification of values	`val` x : $valtype$
specification of types (abstract)	`type` t
specification of types (manifest)	`type` $t = type$
specification of exceptions	`exception` E [`of` $type$]
specification of classes	`class` C : $classtype$ `end`
specification of sub-modules	`module` X : $modtype$
specification of module types	`module type` S [$= modtype$]

Naming a module type: with the `module type` binding

```
module type SIG = sig ... end
```

Signature constraint

For each structure definition, the system infers a signature which is the most informative one: all components of the structure are declared, with their most precise type.

```
struct
    type t = int
    let f x = x
end
```

```
sig
    type t = int
    val f : 'a -> 'a
end
```

To hide components of a structure, or restrict their types, use a signature constraint (*structure* : *signature*).

The constraint checks that the structure satisfies the signature (all components specified in the signature must be defined in the structure with types that match those in the signature).

It returns the structure viewed with the restricted signature.

An example of signature constraint

```
module S =
  (struct
     type t = int
     let x = 1
     let y = x + 1
   end :
   sig
     type t
     val y : t
   end)
```

`S.x;;` *error "S.x is unbound"*

`S.y + 1;;` *error "S.y is not of type int"*

Multiple views of a module

```
module Stamp =
  struct
    type t = int
    let new () = ...
    let equal s1 s2 = (s1 = s2)
  end

module type ABSTRSTAMP =
  sig
    type t
    val equal: t -> t -> bool
  end

module AbstrStamp = (Stamp : ABSTRSTAMP)
```

Users of `AbstrStamp` can only test the equality of two stamps; those of `Stamp` can also create new stamp and use them as integers (e.g. for printing).

There is no code duplication between `AbstrStamp` and `Stamp`.

Modules and separate compilation

A compilation unit A comprises two files:

- An implementation file A`.ml`:
  a sequence of definitions
  like the inside of a **`struct...end`**

- An interface file A`.mli` (optional):
  a sequence of specifications
  like the inside of a **`sig...end`**

Another compilation unit B can reference A as if it were a structure, using the dot notation A`.x` or by doing `open` A.

Separate compilation of a program

Source files: `a.ml`, `a.mli`, `b.ml`

Compilation steps:

```
ocamlc -c a.mli              (compile the interface of A, create a.cmi)
ocamlc -c a.ml               (compile the implementation of A, create a.cmo)
ocamlc -c b.ml               (compile the implementation of B, create b.cmo)
ocamlc -o myprog a.cmo b.cmo    (final linking step)
```

The program is equivalent to the following monolithic source:

```
module A = (struct contents of a.ml end : sig contents of a.mli end)
module B = struct contents of b.ml end
```

The order of module definitions corresponds to the order of `.cmo` object files on the command line at linking-time.

Parameterized modules

A *fonctor* is a function from modules to modules:

`functor(`*S*`:` *signature*`) ->` *module*

The *module* (the functor body) is explicitly parameterized by the module parameter *S*. It references components of its parameter via the dot notation.

```
module T = functor(S : SIG) ->
    struct
        type u = S.t * S.t
        let y = S.g(S.x)
    end
```

Functor application

Components of `T` cannot be accessed directly. `T` must first be applied to an implementation of the signature `SIG`.

```
module T1 = T(S1)
module T2 = T(S2)
```

`T1`, `T2` can then be used as regular structures:

```
(T1.y : T2.u)
```

`T1` and `T2` share all their code.

Extended example: polynomials on a ring

We implement operations on polynomials over any ring, given as parameter.

The ring parameter is a structure matching the following signature:

```
module type RING =
  sig
    type t                (* type of the elements of the ring *)
    val zero: t
    val one: t
    val plus: t->t->t     (* sum *)
    val opp: t->t         (* opposite *)
    val mult: t->t->t     (* product *)
  end
```

Dense polynomials

```
module DensePoly = functor (Ring : RING) ->
  struct
    module R = Ring
    type t = R.t array                  (* the type of polynomials *)
    let zero = [| R.zero |]             (* the null polynomial *)
    let one = [| R.one |]               (* the polynomial 1 *)
    let mono c n =                      (* the monomial c.X^n *)
      let p = Array.create (n+1) R.zero in
      p.(n) <- c; p
    let coeff n p = p.(n)               (* return the coefficient of X^n in p *)
    let degree p =                      (* return the degree of p *)
      let d = ref (Array.length p - 1) in
      while !d >= 0 && p.(!d) = R.zero do decr d done;
      !d
    let plus p1 p2 =                    (* sum *)
      let r = Array.create (max (Array.length p1) (Array.length p2)) R.zero in
      for i = 0 to Array.length p1 - 1 do r.(i) <- p1.(i) done;
      for i = 0 to Array.length p2 - 1 do r.(i) <- R.plus r.(i) p2.(i) done;
      r
    let opp p = ...                     (* opposite *)
    let mult p1 p2 = ...                (* product *)
  end
```

Using the polynomial fonctor

Integer polynomials:

```
module Integers =
  struct
    type t = int
    let zero = 0
    let one = 1
    let plus x y = x + y
    let opp x = -x
    let mult x y = x * y
  end
module IntPoly = DensePoly(Integers)
```

Integer polynomials over two variables = polynomials of integer polynomials:

```
module Int2Poly = DensePoly(IntPoly)
```

This is correct because the result of `DensePoly` matches the signature `RING`.

Sparse polynomials

```
module SparsePoly = functor (Ring : RING) ->
  struct
    module R = Ring
    type t = (R.t * int) list            (* the type of polynomials *)
    let zero = []                        (* the null polynomial *)
    let one = [(R.one, 0)]               (* the polynomial 1 *)
    let mono c n = [(c, n)]              (* the monomial c.X^n *)
    let rec degree = function
        [] -> 0
      | [(coeff, d)] -> d
      | (coeff, d) :: rem -> degree rem
    let rec coeff n p =
      match p with
        [] -> R.zero
      | (c, d) :: rem ->
          if d = n then c else
          if d > n then R.zero else coeff n rem
    let plus p1 p2 =  ...                (* sum *)
    let opp p = ...                      (* opposite *)
    let mult p1 p2 = ...                 (* product *)
  end
```

Representation independence

`DensePoly` and `SparsePoly` implement the same interface and are essentially interchangeable.

```
module IntPoly = SparsePoly(Integers)
module Int2Poly = SparsePoly(IntPoly)
```

But not fully interchangeable: the representation type of polynomials remains apparent.

```
module P1 = DensePoly(Integers);;        (* P1.t = int array *)
P1.degree [| 1;2;3;4 |];;                (* - : int = 3 *)

module P2 = SparsePoly(Integers);;       (* P2.t = (int * int) list *)
P2.degree [| 1;2;3;4 |];;                (* type error *)
```

Representation hiding

We apply a signature constraint to the results of the functors `DensePoly` and `SparsePoly` to make the type `t` abstract.

First try:

```
module type POLYNOMIAL =
  sig
    module R : RING
    type t
    val zero: t                          val one: t
    val mono: R.t -> int -> t            val degree: t -> int
    val coeff: int -> t -> R.t           val opp: t -> t
    val plus: t -> t -> t                val mult: t -> t -> t
  end
module DensePoly = functor (Ring: RING) -> (struct ... end : POLYNOMIAL)
```

Problem: we have also made abstract the type `R.t` of coefficients, which renders the result of `DensePoly` useless:

```
module IntPoly = DensePoly(Integers);;
IntPoly.mono 15 2;;
```

This expression has type int but is used with type IntPoly.R.t

The right solution: use a manifest type to preserve the identity of the type `R.t` in the result:

```
module DensePoly = functor (Ring: RING) ->
  (struct ... end : POLYNOMIAL with type R.t = Ring.t)
```

Thus:

```
module IntPoly = DensePoly(Integers)
```

We have `IntPoly.t` abstract,
but `IntPoly.R.t` $=$ `Integers.t` $=$ `int` as desired.

The `with` notation

Allows to add type equalities to an existing signature.

The expression `POLYNOMIAL with type R.t = Ring.t` is an abbreviation for the signature

```
sig
  module R : sig type t = Ring.t
                 val zero: t   val one: t
                 val plus: t->t->t
                 val opp: t->t
                 val mult: t->t->t  end
  type t
  val zero: t
  val one: t
  val mono: R.t -> int -> t
  val plus: t -> t -> t
  val opp: t -> t
  val mult: t -> t -> t
  val degree: t -> int
  val coeff: int -> t -> R.t
end
```

Modules vs. polymorphism

Polynomials can also be expressed without functors, just with ML polymorphism:

```
type 'a polynomial
type 'a ring =
  { zero: 'a; one: 'a; plus: 'a->'a->'a; opp: 'a->'a; mult: 'a->'a->'a }
val zero: 'a ring -> 'a polynomial
val one: 'a ring -> 'a polynomial
val mono: 'a ring -> 'a -> int -> 'a polynomial
val plus: 'a ring -> 'a polynomial -> 'a polynomial -> 'a polynomial
val opp: 'a ring -> 'a polynomial -> 'a polynomial
val mult: 'a ring -> 'a polynomial -> 'a polynomial -> 'a polynomial
val degree: 'a ring -> 'a polynomial -> int
val coeff: 'a ring -> int -> 'a polynomial -> 'a
```

First problem: the ring operations must be passed as extra argument to each polynomial operation.

Functors = mass parameterization
Base language = individual parameterization

Second problem: the type of polynomials reflects only the type of the carrier of the ring, not of the ring as a whole.

```
let integers =
  { zero = 0; one = 1; plus = fun x y -> x + y; ...}
let z_over_3z =
  { zero = 0; one = 1; plus = fun x y -> (x + y) mod 3; ...}
let p1 = mono integers 15 2
let p2 = mono z_sur_3z 2 3
let p3 = mult integers p1 p2
```

It does not make sense to mix `p1` and `p2`, although they have the same type `int polynomial`.

The functorial presentation detects this error by static typing:

```
module Z_over_3Z =
  struct let zero = 0  let one = 1  let plus x y = (x+y) mod 3 ... end
module IntPoly = DensePoly(Integers)
module Z3ZPoly = DensePoly(Z_over_3Z)
```

The types `IntPoly.t` and `Z3ZPoly.t` are both abstract, hence incompatible:

```
Z3ZPoly.degree IntPoly.one
```

> *This expression has type IntPoly.t*
> *but is here used with type Z3ZPoly.t*

Core ML polymorphism and parameterized datatypes:

parameterized types:	type	$\longrightarrow$	type
functions:	value	$\longrightarrow$	value

ML modules:

functors:	types interpreted by values	$\longrightarrow$	types interpreted by values

Special case: $\mathbf{Z}/2\mathbf{Z}$

Another strength of the functor-based solution: we can provide direct implementations (without using the functor) for certain types.

Example: to implement $\mathbf{Z}/2\mathbf{Z}[X]$ efficiently, we give a structure `Poly_Z2Z` based on bit vectors, with the same interface as the result of the functor `DensePoly`:

```
module Poly_Z2Z : (POLYNOMIAL with type R.t = int) =
  struct
    module R = Z_over_2Z
    type t = bitvect
    let add = bitvect_xor
    let mult = bitvect_and
    let op = bitvect_neg
    ...
  end
```

`Poly_Z2Z` can replace `DensePoly(Z_over_2Z)` without requiring any changes in the remainder of the program.

Modules vs. classes

Some overlap in the case of abstract types modifiable in place and with only unary operations:

```
module Hashtbl:
  sig
    type ('a, 'b) t
    val empty: int -> ('a, 'b) t
    val add: ('a, 'b) t -> 'a -> 'b -> unit
    val find: ('a, 'b) t -> 'a -> 'b
  end

class ['a, 'b] hashtbl : int ->
  object
    method add: 'a -> 'b -> unit
    method find: 'a -> 'b
  end
```

Similarly, extension functors = simple cases of inheritance.

Strengths of modules:

- n-ary operations.
- Several related abstract types.
- Functors in their full generality.

Strengths of classes and objects:

- Subtyping between values.
- Late binding (open recursion).

A sociological perspective:

Modules and a.d.t.	Classes and objects
Protect against users	Trust users
Pessimistic mindset	Optimistic mindset
Educated terminology (signatures, structures, functors)	Common words (objects, classes, methods, to inherit)
Mathematical perspective (operations over abstract types)	Anthropomorphic perspective (send messages asking objects to do things)
Elitist discourse	Populist discourse
...	...

A calculus of modules

Modules:	$m ::= p$	named module
	$\mid$ `struct` $d_1; \ldots; d_n$ `end`	structure
	$\mid$ `functor` $(X : M) \rightarrow m$	functor
	$\mid m(m')$	functor application
	$\mid (m : M)$	signature constraint
Definitions:	$d ::=$ `let` $v = a$	value definition
	$\mid$ `type` $t = T$	type definition
	$\mid$ `module` $X = m$	module definition
Access paths:	$p ::= X$	module identifiers
	$\mid p.X$	sub-module component

The type algebra

Module types:	$M ::= \texttt{sig}\ S_1; \ldots; S_n\ \texttt{end}$	signature
	$\mid \texttt{functor}\ (X : M) \rightarrow M'$	functor type
Specifications:	$S ::= \texttt{val}\ v : \sigma$	value specification
	$\mid \texttt{type}\ t$	abstract type spec.
	$\mid \texttt{type}\ t = \tau$	manifest type spec.
	$\mid \texttt{module}\ X : M$	module specification

Extensions to the base language

Expressions:	$a ::= v$	value identifier
	$\mid p.v$	value component of a structure
	$\mid \ldots$	as before
Types:	$\tau ::= t$	named type
	$\mid p.t$	type component of a structure
	$\mid \ldots$	as before
Type scheme:	$\sigma ::= \forall \alpha_1 \ldots \alpha_n .\ \tau$	

Dynamic semantics

By translation to ML + records, erasing all type information:

$$
\begin{array}{rcl}
[\![X]\!] & = & X \\
[\![p.X]\!] & = & p.X \\
[\![\texttt{struct}\ d_1; \ldots; d_n\ \texttt{end}]\!] & = & \{[\![d_1]\!]; \ldots; [\![d_n]\!]\} \\
[\![\texttt{functor}\ (X : M) \to m]\!] & = & \texttt{fun}\ X \to [\![m]\!] \\
[\![m(m')]\!] & = & [\![m]\!]([\![m']\!]) \\
 & & \\
[\![\texttt{let}\ v = a]\!] & = & (v = a) \\
[\![\texttt{type}\ t = T]\!] & = & \text{nothing} \\
[\![\texttt{module}\ X = m]\!] & = & (X = [\![m]\!])
\end{array}
$$

Typing issues

- Equivalence between base language types induced by manifest types: if `X : sig type t = int end`, then `X.t` is equivalent to `int`.

- Subtyping between module types:
  by forgetting extra components
  by forgetting type equalities

- Dependent types:
  inside signatures

  ```
  sig type t; val x: t end
  ```

  between argument and result of a functor

  ```
  functor (X: sig type t end) -> sig type t = X.t end
  ```

Type equivalence

$$\frac{E = E_1;\ \texttt{type}\ t = \tau;\ E_2}{E \vdash t \approx \tau}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{type}\ t = \tau;\ S_2^*\ \texttt{end}}{E \vdash p.t \approx \tau\{z \leftarrow p.z \mid z \text{ bound in } S_1^*\}}$$

Plus reflexivity, symmetry, transitivity, congruence.

Example:

```
p : sig
      type t;
      type u;
      type v = t * u
    end
```

We do not have `p.v` $\approx$ `t * u` (this type is ill-formed).

We have `p.v` $\approx$ `(t * u)` $\{\mathrm{t} \leftarrow \mathrm{p.t}; \mathrm{u} \leftarrow \mathrm{p.u}\}$
that is, `p.v` $\approx$ `p.t * p.u`.

Typing the base language

As before, but take $\approx$ into account:

$$\frac{E = E_1;\ \texttt{val}\ v : \sigma;\ E_2 \qquad \tau \leq \sigma}{E \vdash v : \tau}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{val}\ v : \sigma : T;\ S_2^*\ \texttt{end} \qquad \tau \leq \sigma\{z \leftarrow p.z \mid z \text{ bound in } S_1^*\}}{E \vdash p.v : \tau}$$

$$\frac{E \vdash a : \tau \qquad E \vdash \tau \approx \tau'}{E \vdash a : \tau'}$$

Type inference: as before, but perform unification modulo $\approx$.

Subtyping between signatures

$$\frac{\rho : \{1 \ldots n\} \mapsto \{1 \ldots m\} \qquad E; S_1; \ldots; S_m \vdash S_{\rho(i)} <: S'_i \text{ for } i = 1, \ldots, n}{E \vdash (\texttt{sig } S_1; \ldots; S_m \texttt{ end}) <: (\texttt{sig } S'_1; \ldots; S'_n \texttt{ end})}$$

ρ is uniquely determined by the names of the components of the two signatures.

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{val } v : \tau) <: (\texttt{val } v : \tau')} \qquad E \vdash (\texttt{type } t) <: (\texttt{type } t)$$

$$E \vdash (\texttt{type } t = \tau) <: (\texttt{type } t) \qquad \frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type } t = \tau) <: (\texttt{type } t = \tau')}$$

$$\frac{E \vdash \tau \approx t}{E \vdash (\texttt{type } t) <: (\texttt{type } t = \tau)}$$

$$\frac{E \vdash M <: M'}{E \vdash (\texttt{module } X : M) <: (\texttt{module } X : M')}$$

Examples of subtyping between signatures

Removal of a component:

```
(sig type t = int; val v : t end) <: (sig type t = int end) <: (sig end)
```

Removal of a type equality

```
(sig type t = int; val v : t end) <: (sig type t; val v : t end)
```

Using a removed component to subtype the others:

```
(sig type t = int; val v : t end) <: (sig val v : int end)
```

Permutation of independent components:

```
(sig type t = int; val v : int end) <: (sig val v : int; type t = int end)
```

Another permutation:

```
(sig type t; type u = t end) <: (sig type u; type t = u end)
```

Examples of non-subtyping

A component is missing:

(sig type t = int end) $\not<:$ (sig type t = int val v : t end)

A component has the wrong name:

(sig type t = int val n : t end) $\not<:$ (sig type t = int val v : t end)

Bad definition of a manifest type:

(sig type t = int end) $\not<:$ (sig type t = bool end)

(sig type t end) $\not<:$ (sig type t = bool end)

Subtyping between functors

$$\frac{E \vdash P_2 <: P_1 \qquad E;\ \texttt{module}\ X : P_2 \vdash R_1 <: R_2}{E \vdash (\texttt{functor}\ (X : P_1) \to R_1) <: (\texttt{functor}\ (X : P_2) \to R_2)}$$

Contravariance in the argument types.

Since the types are dependent, $R_1 <: R_2$ is checked under the assumption $X : P_2$ (the most precise of the two parameter types).

$$\begin{array}{l}(\texttt{functor}\ (X : \texttt{sig type}\ t\ \texttt{end}) \to \texttt{sig type}\ t = X.t\ \texttt{end}) \\ \quad <: \ (\texttt{functor}\ (X : \texttt{sig type}\ t = \ \texttt{int end}) \to \texttt{sig type}\ t = \texttt{int end})\end{array}$$

Typing the module language

$$\frac{E = E_1;\ \texttt{module}\ X : M;\ E_2}{E \vdash X : M}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{module}\ X : M;\ S_2^*\ \texttt{end}}{E \vdash p.X : M\{z \leftarrow p.z \mid z \text{ bound in } S_1^*\}}$$

$$\frac{E \vdash d^* : S^*}{E \vdash (\texttt{struct}\ d^*\ \texttt{end}) : (\texttt{sig}\ S^*\ \texttt{end})} \qquad E \vdash \varepsilon : \varepsilon$$

$$\frac{E \vdash a : \tau \qquad E;\ \texttt{val}\ v : \tau \vdash d^* : S^*}{E \vdash (\texttt{let}\ v = a;\ d^*) : (\texttt{val}\ v : \tau;\ S^*)}$$

$$\frac{t \notin \mathrm{Dom}(E) \qquad E;\ \texttt{type}\ t = \tau \vdash d^* : S^*}{E \vdash (\texttt{type}\ t = \tau;\ d^*) : (\texttt{type}\ t = \tau;\ S^*)}$$

$$\frac{E \vdash m : M \qquad X \notin \mathrm{Dom}(E) \qquad E;\ \texttt{module}\ X : M \vdash d^* : S^*}{E \vdash (\texttt{module}\ X = m;\ d^*) : (\texttt{module}\ X : M;\ S^*)}$$

$$\frac{X \notin \mathsf{Dom}(E) \qquad E;\ X : M \vdash m : M'}{E \vdash (\texttt{functor}\ (X : M) \to m) : (\texttt{functor}\ (X : M) \to M')}$$

$$\frac{E \vdash m_1 : (\texttt{functor}\ (X : M) \to M') \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'\{X \leftarrow m_2\}}$$

$$\frac{E \vdash m : M}{E \vdash (m : M) : M} \qquad \frac{E \vdash m : M \qquad E \vdash M <: M'}{E \vdash m : M'} \qquad \frac{E \vdash p : M}{E \vdash p : (M/p)}$$

Typing simple structures

```
module S =
  struct
    type t = int
    let x = 0
    let f x = x + 1
  end
```

Type definitions are transparent by default:

```
S : sig type t = int  val x : int  val f: int -> int end
```

Another possible signature (equivalent):

```
S : sig type t = int  val x : t  val f: t -> t end
```

In both case, we can type `1 + S.f 0`.

To render `t` opaque, use a constraint:

```
module T = (S : sig type t  val x : t  val f : t -> t end)
```

```
T : sig type t val x : t val f : t -> t end
```

Then, `1 + T.f 0` is rejected.

Redefinitions of type and module names

The restrictions $t \notin \mathrm{Dom}(E)$ et $X \notin \mathrm{Dom}(E)$ avoid the following situation:

```
struct
  type t = int                    type t = int
  let x = (1 : t)                 val x : t
  type t = bool                   type t = bool
  let y = (true : t)              val y : t
  ... if x = y then ...           x and y have the same type t
end
```

Redefinitions can be allowed if we compare identifiers by binding locations, not just by names:

```
struct
  type t₁ = int                   type t₁ = int
  let x = (1 : t₁)                val x : t₁
  type t₂ = bool                  type t₂ = bool
  let y = (true : t₂)             val y : t₂
  ... if x = y then ...           x : t₁ and y : t₂
end
```

Eliminating functor types

```
module Identity :
  functor(X : sig type t end) → sig type t = X.t end

module E = struct type t = int end

module B = Identity(E)
```

The type of `B` is

$$\texttt{(sig type t = X.t end)}\{\texttt{X} \leftarrow \texttt{E}\} \quad = \quad \texttt{sig type t = E.t end}$$

From this type, we obtain $\texttt{B.t} \approx \texttt{E.t} \approx \texttt{int}$. This is how type equalities are propagated through functors.

What if a functor argument is not a path?

```
module B = Identity(struct type t = int end)
```

The application rule is not directly usable, because

`(sig type t = X.t end)`$\{$`X` $\leftarrow$ `struct type t = int end`$\}$

that is,

```
sig type t = (struct type t = int end).t end
```

is not a well-formed type.

Simple solution: always name functor arguments.

Eliminating dependencies via subtyping

Better solution: notice that

```
  functor(X : sig type t end) → sig type t = X.t end
<:
  functor(X : sig type t = int end) → sig type t = int end
```

Using the application rule with the premises

```
Identity : functor(X : sig type t = int end) → sig type t = int end
```

```
struct type t = int end : (sig type t = int end)
```

we obtain

```
Identity(struct type t = int end) : (sig type t = int end)
```

The type strengthening rule

```
p : sig ...; type t; ... end
```

The type `p.t` is unknown, thus a priori incompatible with any other type. However, it is equal to itself. The path `p` thus also has type:

```
p : sig ...; type t = p.t; ... end
```

More generally, if $E \vdash p : M$, then $E \vdash p : (M/p)$, where the type M/p is defined by:

$$
\begin{aligned}
(\texttt{sig } d_1; \ldots; d_n \texttt{ end})/p &= \texttt{sig } d_1/p; \ldots; d_n/p \texttt{ end} \\
(\texttt{val } v : \tau)/p &= \texttt{val } v : \tau \\
(\texttt{type } t)/p &= \texttt{type } t = p.t \\
(\texttt{type } t = \tau)/p &= \texttt{type } t = \tau \\
(\texttt{module } X : M)/p &= \texttt{module } X : (M/(p.X)) \\
(\texttt{functor}(X : M) \rightarrow M')/p &= \texttt{functor}(X : M) \rightarrow M'
\end{aligned}
$$

Applications of strengthening, (1)

To take a restricted view of a module:

```
E : sig type t; val x: t; val y: t -> int end
```

We want to hide the `y` component. If we do

```
module B = (E : sig type t; val x: t end)
```

we hide `y`, but `B.t` is a different type from `E.t`. Better:

```
module B = (E : sig type t = E.t; val x: t end)
```

This ensures `B.t` $\approx$ `E.t`. To check that the constraint is satisfied, we must apply the strengthening rule to the signature of `E`.

Applications of strengthening, (2)

To propagate types through functors:

```
module Identity = functor(X : sig type t end) → X
```

Without strengthening:

```
module Identity = functor(X : sig type t end) → sig type t end
```

We lose the equality between the type `t` of the argument and that of the result.

With strengthening of the type of `X`:

```
module Identity = functor(X : sig type t end) → sig type t = X.t end
```

Applications of strengthening, (3)

To check sharing constraints:

```
module F: functor (X: sig type t end) →
          functor (Y: sig type t = X.t end) → ...

module E: sig type t = B.t end
module B: sig type t end
```

To type-check `F(E)(B)`:

```
F(E): functor (Y: sig type t = E.t end) → ...
```

and to conclude, we must consider `B` with type

```
B: sig type t = B.t end
```

which is subtype of the expected type for `Y`.

Extensions

- Extend the class of access paths.
  Example: $p ::= x \mid p.x \mid p(p')$.
  But if we take $p ::= m$, we lose the ability to generate a new abstract type.

- Modules as first-class values.
  Suppress the stratification into base language / module language.
  Problem: type-checking becomes undecidable (like in $F_{<:}$).