Abstraction

Abstraction

- When faced with creating and maintaining a complex system, the interactions of different components can be simplified by hiding the details of each component's implementation from the rest of the system.
- Details of a component's *implementation* are hidden by protecting it with an *interface*.
- Abstraction is maintained by ensuring that the rest of the system is invariant to changes of implementation that do not affect the interface.

Modules: structures

```
module IntSet = struct

  type t = int list

  let empty = []

  let is_empty = function
    | [] -> true
    | _ -> false

  let equal_member (x : int) (y : int) =
    x = y

  let rec mem x = function
    | [] -> false
    | y :: rest ->
```

Modules: structures

```
        if (equal_member x y) then true
        else mem x rest

  let add x t =
    if (mem x t) then t
    else x :: t

  let rec remove x = function
    | [] -> []
    | y :: rest ->
        if (equal_member x y) then rest
        else y :: (remove x rest)

  let to_list t = t

end
```

Modules: structures

```
let one_two_three : IntSet.t =
  IntSet.add 1
    (IntSet.add 2
      (IntSet.add 3 IntSet.empty))
```

Modules: structures

```
open IntSet

let one_two_three : t =
  add 1 (add 2 (add 3 empty))
```

Modules: structures

```
let one_two_three : IntSet.t =
  IntSet.(add 1 (add 2 (add 3 empty)))
```

Modules: structures

```
module IntSetPlus = struct
  include IntSet

  let singleton x = add x empty
end
```

Modules: signatures

```
sig
  type t = int list
  val empty : 'a list
  val is_empty : 'a list -> bool
  val equal_member : int -> int -> bool
  val mem : int -> int list -> bool
  val add : int -> int list -> int list
  val remove : int -> int list -> int list
  val to_list : 'a -> 'a
end
```

Modules: signatures

```
module IntSet : sig
  type t = int list
  val empty : int list
  val is_empty : int list -> bool
  val mem : int -> int list -> bool
  val add : int -> int list -> int list
  val remove : int -> int list -> int list
  val to_list : int list -> int list
end = struct
  ...
end
```

Modules: signatures

```
module type IntSetS = sig
  type t = int list
  val empty : int list
  val is_empty : int list -> bool
  val mem : int -> int list -> bool
  val add : int -> int list -> int list
  val remove : int -> int list -> int list
  val to_list : int list -> int list
end

module IntSet : IntSetS = struct
  ...
end
```

Modules: abstract types

```
let print_set (s : IntSet.t) : unit =
  let rec loop = function
    | x :: xs ->
        print_int x;
        print_string " ";
        loop xs
    | [] -> ()
  in
    print_string "{ ";
    loop s;
    print_string "}"
```

Modules: abstract types

```
module type IntSetS : sig
  type t
  val empty : t
  val is_empty : t -> bool
  val mem : int -> t -> bool
  val add : int -> t -> t
  val remove : int -> t -> t
  val to_list : t -> int list
end

module IntSet : IntSetS = struct
  ...
end
```

Modules: abstract types

```
# let print_set (s : IntSet.t) : unit =
    let rec loop = function
      | x :: xs ->
          print_int x;
          print_string " ";
          loop xs
      | [] -> ()
    in
      print_string "{ ";
      loop s;
      print_string "}";;
```

```
Characters 172-173:
      loop s;
           ^
Error: This expression has type IntSet.t
       but an expression was expected of type
       int list
```

Existential types

```
NatSetImpl =
  λα::*.
      α
      × (α → Bool)
      × (Nat → α → Bool)
      × (Nat → α → α)
      × (Nat → α → α)
      × (α → List Nat)
```

```
empty = Λα::*.λs:NatSetImpl α.π₁ s
is_empty = Λα::*.λs:NatSetImpl α.π₂ s
mem = Λα::*.λs:NatSetImpl α.π₃ s
add = Λα::*.λs:NatSetImpl α.π₄ s
remove = Λα::*.λs:NatSetImpl α.π₅ s
to_list = Λα::*.λs:NatSetImpl α.π₆ s
```

Existential types

```
nat_set_package =
  pack List Nat,⟨
       nil [Nat],
        isempty [Nat],
        λn:Nat.fold [Nat] [Bool]
          (λx:Nat.λy:Bool.or y (equal_nat n x))
          false,
        cons [Nat],
        λn:Nat.fold [Nat] [List Nat]
          (λx:Nat.λl:List Nat
             if (equal_nat n x) [List Nat] l
               (cons [Nat] x l))
          (nil [Nat]),
        λl:List Nat.l ⟩
  as ∃α::*.NatSetImpl α
```

Existential types

```
open nat_set_package as NatSet, nat_set

one_two_three =
  (add [NatSet] nat_set) one
    ((add [NatSet] nat_set) two
      ((add [NatSet] nat_set) three
        (empty [NatSet] nat_set)))
```

Existential types

$$\frac{\Gamma \vdash M : A[\alpha := B] \qquad \Gamma \vdash \exists\alpha{::}K.A :: *}{\Gamma \vdash \text{pack } B, M \text{ as } \exists\alpha{::}K.A : \exists\alpha{::}K.A}\ \exists\text{-intro}$$

Existential types in OCaml

$$\Lambda\alpha{::}*.\lambda p{:}\mathsf{Bool}.\lambda x{:}\alpha.\lambda y{:}\alpha.$$

$$\quad \mathsf{if}\ p\ [\alpha]\ x\ y$$

$$\Lambda\alpha{::}*.\Lambda\beta{::}*.\lambda p{:}\mathsf{Bool}.\lambda x{:}\alpha.\lambda y{:}\beta.$$

$$\quad \mathsf{if}\ p\ [\exists\gamma.\gamma]$$

$$\qquad (\mathbf{pack}\ \alpha,\ x\ \mathbf{as}\ \exists\gamma.\gamma)$$

$$\qquad (\mathbf{pack}\ \beta,\ y\ \mathbf{as}\ \exists\gamma.\gamma)$$

Existential types in OCaml

```
      λp        .λx    .λy    .
if  p       x  y
```

```
             λp        .λx    .λy    .
if  p
          x
          y
```

Existential types in OCaml

```
fun p x y -> if p then x else y
```

$\forall\alpha::*.\ \ \texttt{Bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$

$\forall\alpha::*.\forall\beta::*.\ \ \texttt{Bool} \rightarrow \alpha \rightarrow \beta \rightarrow \exists\gamma::*.\gamma$

Existential types in OCaml

```
(* ∃α.α × (α → α) × (α → string)  *)
type t =
  E : 'a * ('a -> 'a)* ('a -> string) -> t

let ints =
 E(0, (fun x -> x + 1), string_of_int)

let floats =
 E(0.0, (fun x -> x +. 1.0), string_of_float)

let E(z, s, p) = ints in
  p (s (s z))
```

Parametricity

Parametricity

- Polymorphism allows a single piece of code to be instantiated with multiple types.
- Polymorphism is *parametric* when all of the instances behave *uniformly*.
- Where abstraction hides details about an implementation from the outside world, parametricity hides details about the outside world from an implementation.

Modules: functors

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

module type SetS = sig
  type t
  type elt
  val empty : t
  val is_empty : t -> bool
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val to_list : t -> elt list
end
```

Modules: functors

```
SetS with type elt = foo
```

expands to

```
sig
  type t
  type elt = foo
  val empty : t
  val is_empty : t -> bool
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val to_list : t -> elt list
end
```

Modules: functors

```
SetS with type elt := foo
```

expands to

```
sig
  type t
  val empty : t
  val is_empty : t -> bool
  val mem : foo -> t -> bool
  val add : foo -> t -> t
  val remove : foo -> t -> t
  val to_list : t -> foo list
end
```

Modules: functors

```
module Set (E : Eq)
  : SetS with type elt := E.t = struct

  type t = E.t list

  let empty = []

  let is_empty = function
    | [] -> true
    | _ -> false

  let rec mem x = function
    | [] -> false
    | y :: rest ->
        if (E.equal x y) then true
        else mem x rest
```

Modules: functors

```
  let add x t =
    if (mem x t) then t
    else x :: t

  let rec remove x = function
    | [] -> []
    | y :: rest ->
        if (E.equal x y) then rest
        else y :: (remove x rest)

  let to_list t = t

end
```

Modules: functors

```
module IntEq = struct
  type t = int
  let equal (x : int) (y : int) =
    x = y
end

module IntSet = Set(IntEq)
```

Universal types

```
SetImpl =
  λγ::*.λα::*.
      α
      × (α → Bool)
      × (γ → α → Bool)
      × (γ → α → α)
      × (γ → α → α)
      × (α → List γ)
```

```
empty = Λγ::*.Λα::*.λs:SetImpl γ α.π₁ s
is_empty = Λγ::*.Λα::*.λs:SetImpl γ α.π₂ s
mem = Λγ::*.Λα::*.λs:SetImpl γ α.π₃ s
add = Λγ::*.Λα::*.λs:SetImpl γ α.π₄ s
remove = Λγ::*.Λα::*.λs:SetImpl γ α.π₅ s
to_list = Λγ::*.Λα::*.λs:SetImpl γ α.π₆ s
```

Universal types

$$\mathsf{EqImpl} = \lambda\gamma :: *.\, \gamma \rightarrow \gamma \rightarrow \mathsf{Bool}$$

$$\mathsf{equal} = \Lambda\gamma :: *.\, \lambda \mathsf{s} : \mathsf{EqImpl}\ \gamma.\, \mathsf{s}$$

Universal types

```
set_package =
  Λγ::*. λeq:EqImpl γ.
    pack List γ,⟨
        nil [γ],
         isempty [γ],
         λn:γ.fold [γ] [Bool]
           (λx:γ.λy:Bool.or y (equal [γ] eq n x))
           false,
         cons [γ],
         λn:γ.fold [γ] [List γ]
           (λx:γ.λl:List γ.
              if (equal [γ] eq n x) [List γ] l
                 (cons [γ] x l))
           (nil [γ]),
         λl:List γ.l ⟩
    as ∃α::*.SetImpl γ α
```

Universal types

$$\frac{\Gamma \vdash M : \forall \alpha{::}K.A \qquad \Gamma \vdash B :: K}{\Gamma \vdash M\ [B] : A[\alpha := B]}\ \forall\text{-elim}$$

Universal types in OCaml

```
(* ∀α.α → α *)
let f x = x

(* (∀α.List α → Int) → Int *)
let g h = h [1; 2; 3] + h [1.0; 2.0; 3.0]
```

```
Characters 27-30:
  let g h = h [1; 2; 3] + h [1.0; 2.0; 3.0]
                             ^^^
Error: This expression has type float
       but an expression was expected of type int
```

Universal types in OCaml

```
Λα::*.λf:α → Int.λx:α.λy:α.
          plus (f x) (f y)
```

```
Λα::*.Λβ::*.λf:∀γ.γ → Int.λx:α.λy:β.
          plus (f [α] x) (f [β] y)
```

Universal types in OCaml

```
λf         .λx   .λy   .
   plus (f x) (f y)
```

```
      λf              .λx   .λy   .
   plus (f      x) (f      y)
```

Universal types in OCaml

```
fun f x y -> f x + f y
```

$$\forall\alpha::*.\ (\alpha \to \text{Int}) \to \alpha \to \alpha \to \text{Int}$$

$$\forall\alpha::*.\forall\beta::*.\ (\forall\gamma::*.\ \gamma \to \text{Int}) \to \alpha \to \beta \to \text{Int}$$

Universal types in OCaml

```
(* ∀α.List α → Int *)
type t = { h : 'a. 'a list -> int }

let len = {h = List.length}

(* (∀α.List α → Int) → Int *)
let g r = r.h [1; 2; 3] + r.h [1.0; 2.0; 3.0]
```

Higher-kinded types

$$f : \forall F{::}{*}{\to}{*}.\forall\alpha{::}{*}.\ F\,\alpha \to (F\,\alpha \to \alpha) \to \alpha$$

$$x : \mathsf{List}\ (\mathsf{Int} \times \mathsf{Int})$$

$$f\ x$$

Higher-kinded types

$$F\ \alpha \quad \sim \quad \texttt{List}(\texttt{Int} \times \texttt{Int})$$

$$F = \texttt{List} \qquad \alpha = \texttt{Int} \times \texttt{Int}$$

$$F = \Lambda\beta.\texttt{List}(\beta \times \beta) \qquad \alpha = \texttt{Int}$$

$$F = \Lambda\beta.\texttt{List}(\texttt{Int} \times \texttt{Int})$$

Lightweight higher-kinded types

A set $\mathbf{F}$ of functions such that:

$$\forall F, G \in \mathbf{F}. \quad F \neq G \quad \Rightarrow \quad \forall t. F(t) \neq G(t)$$

Lightweight higher-kinded types

```
type 'a t = ('a * 'a) list
```

Lightweight higher-kinded types

```
type lst = List
type opt = Option

type ('a, 'f) app =
  | Lst : 'a list -> ('a, lst) app
  | Opt : 'a option -> ('a, opt) app
```

('a, lst) app ≈ 'a list

('a, opt) app ≈ 'a option

Lightweight higher-kinded types

```
type 'f map = {
  map: 'a 'b. ('a -> 'b) ->
             ('a, 'f) app -> ('b, 'f) app;
}

let f : 'b map ->
             (int, 'b) app -> (string, 'b) app =
  fun m c ->
    m.map
      (fun x -> "Int: " ^ (string_of_int x))
       c
```

Lightweight higher-kinded types

```
let lmap : lst map =
  {map = fun f (Lst l) -> Lst (List.map f l)}

let l = f lmap (Lst [1; 2; 3])

let omap : opt map =
  {map = fun f (Opt o) -> Opt (Option.map f o)}

let o = f omap (Opt (Some 6))
```

Lightweight higher-kinded types

Generalised in the *Higher* library