

Abstraction

January 2018

$\Gamma \vdash M : ?$

- ▶ 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.

Abstraction in OCaml

Modules

```
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 ->
      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
```

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



```
open IntSet
```

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

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

```
module IntSetPlus = struct
  include IntSet

  let singleton x = add x empty
end
```

```
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
```

```
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
```

```
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
```

```
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 "}"
```

```
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
```



```

# 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

Abstraction in System F...

Existential types

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

$$\frac{\Gamma \vdash M : \exists \alpha :: K. A \quad \Gamma, \alpha :: K, x : A \vdash M' : B}{\Gamma \vdash \text{open } M \text{ as } \alpha, x \text{ in } M' : B} \exists\text{-elim}$$

NatSetImpl $\alpha =$

α
 $\times (\alpha \rightarrow \text{Bool})$
 $\times (\text{Nat} \rightarrow \alpha \rightarrow \text{Bool})$
 $\times (\text{Nat} \rightarrow \alpha \rightarrow \alpha)$
 $\times (\text{Nat} \rightarrow \alpha \rightarrow \alpha)$
 $\times (\alpha \rightarrow \text{List Nat})$

$\text{empty} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_1 \ s$
 $\text{is_empty} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_2 \ s$
 $\text{mem} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_3 \ s$
 $\text{add} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_4 \ s$
 $\text{remove} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_5 \ s$
 $\text{to_list} = \Lambda \alpha :: *. \lambda s : \text{NatSetImpl } \alpha . \pi_6 \ s$

```

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

```

```
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)))
```

NatSetImpl $\alpha =$

α

× ($\alpha \rightarrow \text{Bool}$)

× ($\text{Nat} \rightarrow \alpha \rightarrow \text{Bool}$)

× ($\text{Nat} \rightarrow \alpha \rightarrow \alpha$)

× ($\text{Nat} \rightarrow \alpha \rightarrow \alpha$)

× ($\alpha \rightarrow \text{List Nat}$)

NatSetImpl =

$\lambda\alpha::*.$

α

$\times (\alpha \rightarrow \text{Bool})$

$\times (\text{Nat} \rightarrow \alpha \rightarrow \text{Bool})$

$\times (\text{Nat} \rightarrow \alpha \rightarrow \alpha)$

$\times (\text{Nat} \rightarrow \alpha \rightarrow \alpha)$

$\times (\alpha \rightarrow \text{List Nat})$

Invariants

Abstraction has further implications beyond the ability to replace one implementation with another:

Abstraction allows us to preserve invariants on types.

```
module Positive : sig
  type t
  val zero : t
  val succ : t -> t
  val to_int : t -> int
end = struct
  type t = int
  let zero = 0
  let succ x = x + 1
  let to_int x = x
end
```

The ability for types to represent invariants beyond their particular data representation fundamentally changes the notion of what a type is:

- ▶ In a language without abstraction (e.g. the simply typed lambda calculus) types only represent particular data representations.
- ▶ In a language with abstraction (e.g. System F) types can represent arbitrary invariants on values.

Existential types in OCaml

```
Λα::*.λp:Bool.λx:α.λy:α.  
  if p [α] x y
```

```
Λα::*.Λβ::*.λp:Bool.λx:α.λy:β.  
  if p [∃γ.γ]  
    (pack α, x as ∃γ.γ)  
    (pack β, y as ∃γ.γ)
```

```
 $\Lambda\alpha::*. \lambda p:\text{Bool}. \lambda x:\alpha. \lambda y:\alpha.$   
  if p [α] x y
```

```
 $\Lambda\alpha::*. \Lambda\beta::*. \lambda p:\text{Bool}. \lambda x:\alpha. \lambda y:\beta.$   
  if p [∃γ.γ]  
    (pack α, x as ∃γ.γ)  
    (pack β, y as ∃γ.γ)
```



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

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

```
(*  $\exists \alpha. \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{string})$  *)  
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))
```

The meaning of types (continued)

Just as abstraction allows types to represent more than just a particular data representation, higher-kinded abstraction allows types to represent an even wider set of concepts:

- ▶ Base-kinded abstraction restricts types to directly representing invariants on values, with each type corresponding to particular set of values.
- ▶ Higher-kinded abstraction allows types to represent more general concepts without a direct correspondence to values – to be continued after we see System $F\omega$...

Relational abstraction

We can give a precise description of abstraction using relations between types.

We define relations between types

$$\rho ::= (x : A, y : B). \phi[x, y]$$

where A and B are System F types, and $\phi[x, y]$ is a logical formula involving x and y .

Logical connectives:

$$\phi ::= \phi \wedge \psi \quad | \quad \phi \vee \psi \quad | \quad \phi \Rightarrow \psi$$

Universal quantifications:

$$\phi ::= \forall x : A. \phi \quad | \quad \forall \alpha. \phi \quad | \quad \forall R \subset A \times B. \phi$$

Existential quantifications:

$$\phi ::= \exists x : A. \phi \quad | \quad \exists \alpha. \phi \quad | \quad \exists R \subset A \times B. \phi$$

Relations:

$$\phi ::= R(t, u)$$

Term equality:

$$\phi ::= (t =_A u)$$

```
type t

val empty : t

val is_empty : t -> bool

val mem : t -> int -> bool

val add : t -> int -> t

val if_empty : t -> 'a -> 'a -> 'a
```



```
type tlist = int list

let emptylist = []

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

let rec memlist x = function
  | [] -> false
  | y :: rest ->
      if x = y then true
      else memlist x rest

let addlist x t =
  if (memlist x t) then t
  else x :: t
```

```
let if_emptylist t x y =  
  match t with  
  | [] -> x  
  | _ -> y
```

```
type ttree =  
  | Empty  
  | Node of ttree * int * ttree  
  
let emptytree = Empty  
  
let is_emptytree = function  
  | Empty -> true  
  | _ -> false  
  
let rec memtree x = function  
  | Empty -> false  
  | Node(l, y, r) ->  
    if x = y then true  
    else if x < y then memtree x l  
    else memtree x r
```

```
let rec addtree x t =  
  match t with  
  | Empty -> Node(Empty, x, Empty)  
  | Node(l, y, r) as t ->  
    if x = y then t  
    else if x < y then Node(addtree x l, y, r)  
    else Node(l, y, addtree x r)  
  
let if_emptytree t x y =  
  match t with  
  | Empty -> x  
  | _ -> y
```

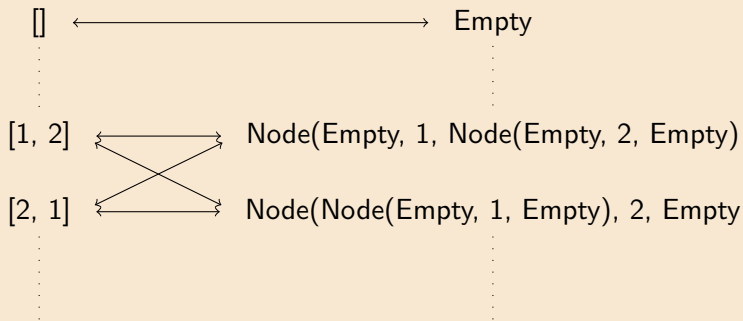
```
type t_list = int list ~
      type t_tree =
        | Empty
        | Node of t_tree * int * t_tree
```

```
type t_list = int list ~ type t_tree =  
                          | Empty  
                          | Node of t_tree * int * t_tree
```

[] ←————→ Empty

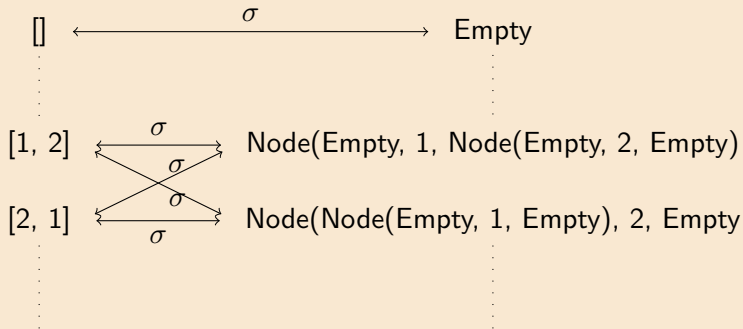
Relations between types

```
type t_list = int list ~ type t_tree =  
                          | Empty  
                          | Node of t_tree * int * t_tree
```



Relations between types

```
type t_list = int list ~ type t_tree =  
                          | Empty  
                          | Node of t_tree * int * t_tree
```




```
let emptylist = []      ~      let emptytree = Empty
```

`let emptylist = []` \sim `let emptytree = Empty`

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

```
let is_emptylist = function
| [] -> true
| _ -> false           ~           let is_emptytree = function
| Empty -> true
| _ -> false
```

```

let is_emptylist = function
  | [] -> true
  | _ -> false
    ~
    let is_emptytree = function
      | Empty -> true
      | _ -> false
  
```

$$\forall x : t_{list}. \forall y : t_{tree}. \\ \sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$$

```

let rec memlist x = function
| [] -> false
| y :: rest ->
    if x = y then true
    else memlist x rest

```

~

```

let rec memtree x = function
| Empty -> false
| Node(l, y, r) ->
    if x = y then true
    else if x < y then memtree x l
    else memtree x r

```

```

let rec memlist x = function
| [] -> false
| y :: rest ->
    if x = y then true
    else memlist x rest

```

```

~
let rec memtree x = function
| Empty -> false
| Node(l, y, r) ->
    if x = y then true
    else if x < y then memtree x l
    else memtree x r

```

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow (\text{mem}_{list} x i = \text{mem}_{tree} y j)$

```

let addlist x t =
  if (memlist x t) then t
  else x :: t

```

```

let rec addtree x t =
  match t with
  | Empty -> Node(Empty, x, Empty)
  | Node(l, y, r) as t ->
    if x = y then t
    else if x < y then
      Node(addtree x l, y, r)
    else
      Node(l, y, addtree x r)

```

~

```
let addlist x t =
  if (memlist x t) then t
  else x :: t
```

```
let rec addtree x t =
  match t with
  | Empty → Node(Empty, x, Empty)
  | Node(l, y, r) as t →
    if x = y then t
    else if x < y then
      Node(addtree x l, y, r)
    else
      Node(l, y, addtree x r)
```

~

$$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$$

$$\sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$$


```
let if_emptylist t x y =  
  match t with  
  | [] -> x  
  | _ -> y
```

~

```
let if_emptytree t x y =  
  match t with  
  | Empty -> x  
  | _ -> y
```

```

let if_emptylist t x y =
  match t with
  | [] -> x
  | _ -> y
  
```

```

~
let if_emptytree t x y =
  match t with
  | Empty -> x
  | _ -> y
  
```

$\forall \gamma. \forall \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow (a = c) \Rightarrow (b = d) \Rightarrow$

$(\text{if_empty}_{list} \ x \ a \ b = \text{if_empty}_{tree} \ y \ c \ d)$

Given $t : \tau_{list}$ and $s : \tau_{tree}$ such that $\sigma(t, s)$:

$if_empty_{list} \ t \ 5 \ 6 \quad \sim \quad if_empty_{tree} \ s \ 5 \ 6$

Given $t : \tau_{list}$ and $s : \tau_{tree}$ such that $\sigma(t, s)$:

$$\text{if_empty}_{list} \ t \ 5 \ 6 \quad \sim \quad \text{if_empty}_{tree} \ s \ 5 \ 6$$

$$\text{if_empty}_{list} \ t \ t \ (\text{add}_{list} \ t \ 1)$$

$$\sim$$

$$\text{if_empty}_{tree} \ s \ s \ (\text{add}_{tree} \ s \ 1)$$

Given $t : \tau_{list}$ and $s : \tau_{tree}$ such that $\sigma(t, s)$:

$$\text{if_empty}_{list} \ t \ 5 \ 6 \quad \sim \quad \text{if_empty}_{tree} \ s \ 5 \ 6$$

$$\text{if_empty}_{list} \ t \ t \ (\text{add}_{list} \ t \ 1)$$

$$\sim$$

$$\text{if_empty}_{tree} \ s \ s \ (\text{add}_{tree} \ s \ 1)$$

$$\text{if_empty}_{list} \ t \ \text{mem}_{list} \ \text{mem}_{list}$$

$$\sim$$

$$\text{if_empty}_{tree} \ t \ \text{mem}_{tree} \ \text{mem}_{tree}$$

```
let if_emptylist t x y =
  match t with
  | [] -> x
  | _ -> y
```

```

~
let if_emptytree t x y =
  match t with
  | Empty -> x
  | _ -> y
```

$\forall \gamma. \forall \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow (a = c) \Rightarrow (b = d) \Rightarrow$

$(\text{if_empty}_{list} \ x \ a \ b = \text{if_empty}_{tree} \ y \ c \ d)$

```

let if_emptylist t x y =
  match t with
  | [] -> x
  | _ -> y
  
```

```

~
let if_emptytree t x y =
  match t with
  | Empty -> x
  | _ -> y
  
```

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} \ x \ a \ b, \text{if_empty}_{tree} \ y \ c \ d)$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t → bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t → int → bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t → int → t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t → 'a → 'a → 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i=j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Given:

- ▶ type T with free variables $\vec{\alpha} = \alpha_1, \dots, \alpha_n$
- ▶ relations $\vec{\rho} = \rho_1 \subset A_1 \times B_1, \dots, \rho_n \subset A_n \times B_n$

We define the relation:

$$\llbracket T \rrbracket[\vec{\rho}] \subset T[\vec{A}] \times T[\vec{B}]$$

If T is α_i then

$$\llbracket T \rrbracket [\vec{\rho}] = \rho_i$$

If T is $T' \times T''$ then

$$\begin{aligned} \llbracket T \rrbracket[\vec{\rho}] &= (x : T[\vec{A}], y : T[\vec{B}]). \\ &\quad \llbracket T' \rrbracket[\vec{\rho}](fst(x), fst(y)) \\ &\quad \wedge \llbracket T'' \rrbracket[\vec{\rho}](snd(x), snd(y)) \end{aligned}$$

If T is $T' + T''$ then

$$\llbracket T \rrbracket[\vec{\rho}] = (x : T[\vec{A}], y : T[\vec{B}]).$$

$$\exists u' : T'[\vec{A}]. \exists v' : T'[\vec{B}].$$

$$x = \text{inl}(u') \wedge y = \text{inl}(v')$$

$$\wedge \llbracket T' \rrbracket[\vec{\rho}](u', v')$$

\vee

$$\exists u'' : T''[\vec{A}]. \exists v'' : T''[\vec{B}].$$

$$x = \text{inr}(u'') \wedge y = \text{inr}(v'')$$

$$\wedge \llbracket T'' \rrbracket[\vec{\rho}](u'', v'')$$

If T is $T' \rightarrow T''$ then

$$\llbracket T \rrbracket[\vec{\rho}] = (f : T[\vec{A}], g : T[\vec{B}]).$$

$$\forall u : T'[\vec{A}]. \forall v : T'[\vec{B}].$$

$$\llbracket T' \rrbracket[\vec{\rho}](u, v) \Rightarrow \llbracket T'' \rrbracket[\vec{\rho}](f u, g v)$$

If T is $\forall\beta.T'$ then

$$\begin{aligned} \llbracket T \rrbracket[\vec{\rho}] &= (x : T[\vec{A}], y : T[\vec{B}]). \\ &\quad \forall\gamma. \forall\delta. \forall\rho' \subset \gamma \times \delta. \\ &\quad \llbracket T' \rrbracket[\vec{\rho}, \rho'](x[\gamma], y[\delta]) \end{aligned}$$

If T is $\exists\beta.T'$ then

$$\begin{aligned}
 \llbracket T \rrbracket[\vec{\rho}] &= (x : T[\vec{A}], y : T[\vec{B}]). \\
 &\quad \exists\gamma. \exists\delta. \exists\rho' \subset \gamma \times \delta. \\
 &\quad \exists u : T'[\vec{A}, \gamma]. \exists v : T'[\vec{B}, \delta]. \\
 &\quad x = \text{pack } \gamma, u \text{ as } T[\vec{A}] \\
 &\quad \wedge y = \text{pack } \delta, v \text{ as } T[\vec{B}] \\
 &\quad \wedge \llbracket T' \rrbracket[\vec{\rho}, \rho'](u, v)
 \end{aligned}$$

Relations between values

val empty:

t

$\sigma(\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t → bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t → int → bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t → int → t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t → 'a → 'a → 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$[[\alpha]][\sigma](\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\forall x : t_{list}. \forall y : t_{tree}.$

$\sigma(x, y) \Rightarrow (\text{is_empty}_{list} x = \text{is_empty}_{tree} y)$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\llbracket \alpha \rrbracket [\sigma](\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\llbracket \alpha \rightarrow \gamma \rrbracket [\sigma, =_{\text{Bool}}](\text{is_empty}_{list}, \text{is_empty}_{tree})$

val mem:

t -> int -> bool

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$(\text{mem}_{list} x i = \text{mem}_{tree} y j)$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\llbracket \alpha \rrbracket [\sigma] (\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\llbracket \alpha \rightarrow \gamma \rrbracket [\sigma, =_{\text{Bool}}] (\text{is_empty}_{list}, \text{is_empty}_{tree})$

val mem:

t -> int -> bool

$\llbracket \alpha \rightarrow \beta \rightarrow \gamma \rrbracket [\sigma, =_{\text{Int}}, =_{\text{Bool}}] (\text{mem}_{list}, \text{mem}_{tree})$

val add:

t -> int -> t

$\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}.$

$\sigma(x, y) \Rightarrow (i = j) \Rightarrow$

$\sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\llbracket \alpha \rrbracket [\sigma] (\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\llbracket \alpha \rightarrow \gamma \rrbracket [\sigma, =_{\text{Bool}}] (\text{is_empty}_{list}, \text{is_empty}_{tree})$

val mem:

t -> int -> bool

$\llbracket \alpha \rightarrow \beta \rightarrow \gamma \rrbracket [\sigma, =_{\text{Int}}, =_{\text{Bool}}] (\text{mem}_{list}, \text{mem}_{tree})$

val add:

t -> int -> t

$\llbracket \alpha \rightarrow \beta \rightarrow \alpha \rrbracket [\sigma, =_{\text{Int}}] (\text{add}_{list}, \text{add}_{tree})$

val if_empty:

t -> 'a -> 'a -> 'a

$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$

$\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$

$\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$

$\rho(\text{if_empty}_{list} x a b, \text{if_empty}_{tree} y c d)$

Relations between values

val empty:

t

$\llbracket \alpha \rrbracket [\sigma] (\text{empty}_{list}, \text{empty}_{tree})$

val is_empty:

t -> bool

$\llbracket \alpha \rightarrow \gamma \rrbracket [\sigma, =_{\text{Bool}}] (\text{is_empty}_{list}, \text{is_empty}_{tree})$

val mem:

t -> int -> bool

$\llbracket \alpha \rightarrow \beta \rightarrow \gamma \rrbracket [\sigma, =_{\text{Int}}, =_{\text{Bool}}] (\text{mem}_{list}, \text{mem}_{tree})$

val add:

t -> int -> t

$\llbracket \alpha \rightarrow \beta \rightarrow \alpha \rrbracket [\sigma, =_{\text{Int}}] (\text{add}_{list}, \text{add}_{tree})$

val if_empty:

t -> 'a -> 'a -> 'a

$\llbracket \forall \delta. \alpha \rightarrow \delta \rightarrow \delta \rightarrow \delta \rrbracket [\sigma] (\text{if_empty}_{list}, \text{if_empty}_{tree})$

$\llbracket \alpha$

$\times (\alpha \rightarrow \gamma)$

$\times (\alpha \rightarrow \beta \rightarrow \gamma)$

$\times (\alpha \rightarrow \beta \rightarrow \alpha)$

$\times (\forall \delta. \alpha \rightarrow \delta \rightarrow \delta \rightarrow \delta) \rrbracket [\sigma, =_{\text{Int}}, =_{\text{Bool}}](\text{set}_{\text{list}}, \text{set}_{\text{tree}})$

Given a type T with free variables $\alpha, \beta_1, \dots, \beta_n$:

$$\forall B_1. \dots \forall B_n. \forall x : (\exists \alpha. T[\alpha, B_1, \dots, B_n]). \forall y : (\exists \alpha. T[\alpha, B_1, \dots, B_n]).$$

$$\exists \gamma. \exists \delta. \exists \sigma \subset \gamma \times \delta.$$

$$\exists u : T[\gamma, B_1, \dots, B_n]. \exists v : T[\delta, B_1, \dots, B_n].$$

$$x = y \quad \Leftrightarrow \quad \begin{aligned} &x = \text{pack } \gamma, u \text{ as } \exists \alpha. T[\alpha, B_1, \dots, B_n] \\ &\wedge y = \text{pack } \delta, v \text{ as } \exists \alpha. T[\alpha, B_1, \dots, B_n] \\ &\wedge \llbracket T \rrbracket[\sigma, =_{B_1}, \dots, =_{B_n}](u, v) \end{aligned}$$

If there is a relation between the implementation types of two values with existential type, and their implementations behave the same with respect to this relation, then the two values are equal.

Invariants

Represent an invariant $\phi[x]$ on a type γ as a relation $\rho \subset \gamma \times \gamma$:

$$\rho(x : \gamma, y : \gamma) = (x = y) \wedge \phi[x]$$

Given a type T with free variable α :

$$\forall f : (\forall \alpha. T[\alpha] \rightarrow \alpha).$$

$$\forall \gamma. \forall \rho \subset \gamma \times \gamma. \forall x : T[\gamma].$$

$$\llbracket T \rrbracket[\rho](x, x) \Rightarrow \rho(f[\gamma] x, f[\gamma] x)$$

Note that:

$$\begin{aligned} & \text{open (pack } \gamma, u \text{ as } \exists \alpha. T[\alpha]) \text{ as } x, \alpha \text{ in } t \\ & \quad = \\ & (\Lambda \alpha. \lambda x : T[\alpha]. t)[\gamma] u \end{aligned}$$

So:

$$\forall \rho \subset \gamma \times \gamma. \quad \llbracket T \rrbracket[\rho](u, u) \Rightarrow \rho \left(\begin{array}{l} \text{open (pack } \gamma, u \text{ as } \exists \alpha. T[\alpha]) \text{ as } x, \alpha \text{ in } t, \\ \text{open (pack } \gamma, u \text{ as } \exists \alpha. T[\alpha]) \text{ as } x, \alpha \text{ in } t \end{array} \right)$$

Identity extension

Given a type T with free variables $\alpha_1, \dots, \alpha_n$:

$$\forall A_1 \dots \forall A_n. \forall x : T[A_1, \dots, A_n]. \forall y : T[A_1, \dots, A_n]. \\ (x =_{T[A_1, \dots, A_n]} y) \Leftrightarrow \llbracket T \rrbracket [=_{A_1}, \dots, =_{A_n}](x, y)$$

System $F\omega$

Higher-kinded abstraction in OCaml

Phantom types

```
module File : sig
  type t
  val open_readwrite : string -> t
  val open_readonly : string -> t
  val read : t -> string
  val write : t -> string -> unit
end = struct
  type t = int
  let open_readwrite filename = ...
  let open_readonly filename = ...
  let read f = ...
  let write f s = ...
end
```

```
# let f = File.open_readonly "foo" in
  File.write f "bar";;
```

Exception: Invalid_argument "write: file is read-only".

```
module File : sig
  type t
  val open_readwrite : string -> t
  val open_readonly : string -> t
  val read : t -> string
  val write : t -> string -> unit
end = struct
  type t = int
  let open_readwrite filename = ...
  let open_readonly filename = ...
  let read f = ...
  let write f s = ...
end
```



```
module File : sig
  type readonly
  type readwrite
  type 'a t
  val open_readwrite : string -> readwrite t
  val open_readonly : string -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit
end = struct
  type readonly
  type readwrite
  type 'a t = int
  let open_readwrite filename = ...
  let open_readonly filename = ...
  let read f = ...
  let write f s = ...
end
```

```
# let f = File.open_readonly "foo" in
  File.write f "bar";;
```

Characters 51-52:

```
File.write f "bar";;
           ^
```

```
Error: This expression has type File.readonly File.t
but an expression was expected of type
File.readwrite File.t
Type File.readonly is not compatible with type
File.readwrite
```

Example: lightweight static capabilities

Example: lightweight static capabilities

```
module Array : sig
  type 'a t
  val length : 'a t -> int
  val set : 'a t -> int -> 'a -> unit
  val get : 'a t -> int -> 'a
end
```

Example: lightweight static capabilities

```
let search cmp arr v =  
  let rec look low high =  
    if high < low then None  
    else begin  
      let mid = (high + low)/2 in  
      let x = Array.get arr mid in  
      let res = cmp v x in  
        if res = 0 then Some mid  
        else if res < 0 then look low (mid - 1)  
        else look (mid + 1) high  
    end  
  in  
  look 0 (Array.length arr)
```

Example: lightweight static capabilities

```
# let arr = [|'a';'b';'c';'d'|];;  
val arr : char array = [|'a'; 'b'; 'c'; 'd'|]
```

Example: lightweight static capabilities

```
# let arr = [|'a';'b';'c';'d'|];;  
  val arr : char array = [|'a'; 'b'; 'c'; 'd'|]  
  
# let test1 = search compare arr 'c';;  
  val test1 : int option = Some 2
```

Example: lightweight static capabilities

```
# let arr = [|'a';'b';'c';'d'|];;  
  val arr : char array = [|'a'; 'b'; 'c'; 'd'|]  
  
# let test1 = search compare arr 'c';;  
val test1 : int option = Some 2  
  
# let test2 = search compare arr 'a';;  
val test2 : int option = Some 0
```


Example: lightweight static capabilities

```
# let arr = [|'a';'b';'c';'d'|];;  
  val arr : char array = [|'a'; 'b'; 'c'; 'd'|]  
  
# let test1 = search compare arr 'c';;  
val test1 : int option = Some 2  
  
# let test2 = search compare arr 'a';;  
val test2 : int option = Some 0  
  
# let test3 = search compare arr 'x';;  
Exception: Invalid_argument "index out of bounds".
```

Example: lightweight static capabilities

```
let search cmp arr v =  
  let rec look low high =  
    if high < low then None  
    else begin  
      let mid = (high + low)/2 in  
      let x = Array.get arr mid in  
      let res = cmp v x in  
      if res = 0 then Some mid  
      else if res < 0 then look low (mid - 1)  
      else look (mid + 1) high  
    end  
  in  
  look 0 (Array.length arr)
```

Example: lightweight static capabilities

```
let search cmp arr v =  
  let rec look low high =  
    if high < low then None  
    else begin  
      let mid = (high + low)/2 in  
      let x = Array.get arr mid in  
      let res = cmp v x in  
      if res = 0 then Some mid  
      else if res < 0 then look low (mid - 1)  
      else look (mid + 1) high  
    end  
  in  
  look 0 ((Array.length arr) - 1)
```

Example: lightweight static capabilities

```
module Array : sig
  type 'a t
  val length : 'a t -> int
  val set : 'a t -> int -> 'a -> unit
  val get : 'a t -> int -> 'a
end
```

Example: lightweight static capabilities

```
module BArray : sig
  type ('s,'a) t
  type 's index

  val last : ('s, 'a) t -> 's index
  val set  : ('s,'a) t -> 's index -> 'a -> unit
  val get  : ('s,'a) t -> 's index -> 'a
end
```

Example: lightweight static capabilities

```
type 'a brand =  
  | Brand : ('s, 'a) t -> 'a brand  
  | Empty : 'a brand  
  
val brand : 'a array -> 'a brand
```

Example: lightweight static capabilities

```
# let Brand x = brand [| 'a'; 'b'; 'c'; 'd' |] in
  let Brand y = brand [| 'a'; 'b' |] in
    get y (last x);;
```

Characters 96-104:

```
  get y (last x);;
      ~~~~~
```

Error: This expression has type s#1 BArray.index
but an expression was expected of type s#2 BArray.index
Type s#1 is not compatible with type s#2

Example: lightweight static capabilities

```
val zero : 's index
val last : ('s, 'a) t -> 's index

val index : ('s, 'a) t -> int -> 's index option
val position : 's index -> int

val middle : 's index -> 's index -> 's index

val next : 's index -> 's index -> 's index option
val previous : 's index -> 's index ->
                's index option
```


Example: lightweight static capabilities I

```
struct
  type ('s,'a) t = 'a array

  type 'a brand =
    | Brand : ('s, 'a) t -> 'a brand
    | Empty : 'a brand

  let brand arr =
    if Array.length arr > 0 then Brand arr
    else Empty

  type 's index = int
  let index arr i =
    if i >= 0 && i < Array.length arr then Some i
    else None

  let position idx = idx
  let zero = 0
```

Example: lightweight static capabilities II

```
let last arr = (Array.length arr) - 1
let middle idx1 idx2 = (idx1 + idx2)/2

let next idx limit =
  let next = idx + 1 in
  if next <= limit then Some next
  else None
let previous limit idx =
  let prev = idx - 1 in
  if prev >= limit then Some prev
  else None

let set = Array.set
let get = Array.get
end
```

Example: lightweight static capabilities

```
let bsearch cmp arr v =
  let open BArray in
  let rec look barr low high =
    let mid = middle low high in
    let x = get barr mid in
    let res = cmp v x in
    if res = 0 then Some (position mid)
    else if res < 0 then
      match previous low mid with
      | Some prev → look barr low prev
      | None → None
    else
      match next mid high with
      | Some next → look barr next high
      | None → None
  in
  match brand arr with
  | Brand barr → look barr zero (last barr)
  | Empty → None
```

Example: lightweight static capabilities

```
let set = Array.unsafe_set
```

```
let get = Array.unsafe_get
```