

# Abstraction

Leo White

Jane Street

February 2017

# 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.

# Abstraction in OCaml

# Modules

## 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 ->
      if (equal_member x y) then true
      else mem x rest

  let add x t =
```

## Modules: structures

```
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



# Invariants

# Invariants

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

Abstraction allows us to preserve invariants on types.

# Invariants

```
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 meaning of types

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.

# Phantom types

# 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
```

## Phantom types

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

Exception: Invalid\_argument "write: file is read-only".

# 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
```



# Phantom types

```
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
```

## Phantom types

```
# 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
```

## 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.

## Existential types in OCaml

# Existential types in OCaml

```
 $\Lambda\alpha::*. \lambda p:\text{Bool}. \lambda x:\alpha. \lambda y:\alpha.$   
  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)
```

# Existential types in OCaml

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

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

# Existential types in OCaml

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

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

## Existential types in OCaml

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



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

```
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

```
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
```

# Abstraction in System $F\omega$

# Existential types

# 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

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



# Relational abstraction

## Relational abstraction

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

# Definable relations

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  $y$ .

# Definable relations

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:

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

## Changing implementations

```
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
```

## Changing implementations

```
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
```

## Changing implementations

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

## Changing implementations

```
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)
```



## Changing implementations

```
| 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
```

## Relations between types

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

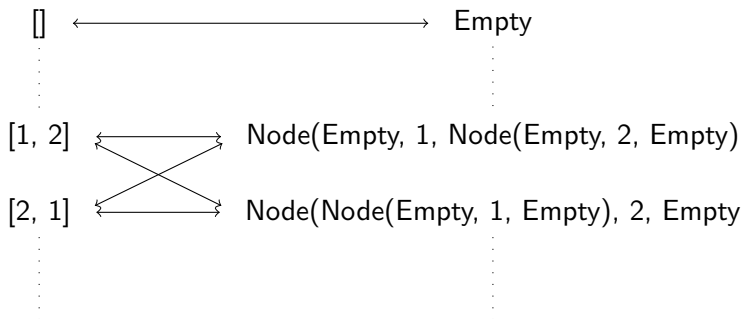
## Relations between types

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

[] ←————→ Empty

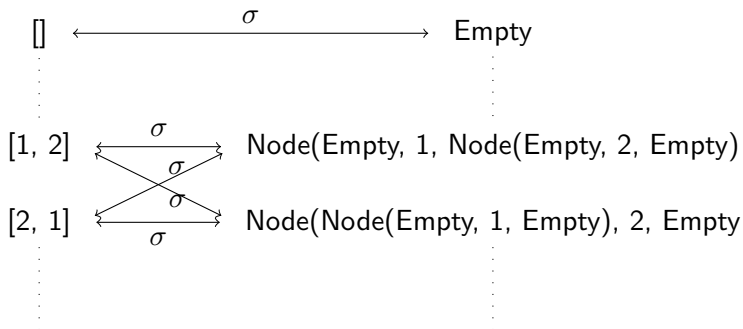
## Relations between types

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



## Relations between types

`type tlist = int list`  $\sim$  `type ttree =`  
`| Empty`  
`| Node of ttree * int * ttree`



## Relations between values

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

## Relations between values

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

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

## Relations between values

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



## Relations between values

```
let is_empty_list = function
| [] -> true
| _ -> false      ~      let is_empty_tree = function
| Empty -> true
| _ -> false
```

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

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

## Relations between values

```
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
```

## Relations between values

```
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
```

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

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

## Relations between values

```
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)
```

## Relations between values

```
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)
```

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

## Relations between values

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

~

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

## Relations between values

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

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

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

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

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

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

## Relations between values

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

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



## Relations between values

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

$$\text{if\_empty}_{list} \ t \ 5 \ 6 \quad \sim \quad \text{if\_empty}_{tree} \ s \ 5 \ 6$$
$$\text{if\_empty}_{list} \ t \ t \ (\text{add}_{list} \ t \ 1)$$
$$\sim$$
$$\text{if\_empty}_{tree} \ s \ s \ (\text{add}_{tree} \ s \ 1)$$

## Relations between values

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

$$\text{if\_empty}_{list} \ t \ 5 \ 6 \quad \sim \quad \text{if\_empty}_{tree} \ s \ 5 \ 6$$
$$\text{if\_empty}_{list} \ t \ t \ (\text{add}_{list} \ t \ 1)$$
$$\sim$$
$$\text{if\_empty}_{tree} \ s \ s \ (\text{add}_{tree} \ s \ 1)$$
$$\text{if\_empty}_{list} \ t \ \text{mem}_{list} \ \text{mem}_{list}$$
$$\sim$$
$$\text{if\_empty}_{tree} \ t \ \text{mem}_{tree} \ \text{mem}_{tree}$$

## Relations between values

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

~

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

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

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

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

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

## Relations between values

```
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:`  
 $t \rightarrow \text{bool}$   $\forall x : t_{list}. \forall y : t_{tree}. \sigma(x, y) \Rightarrow (\text{is\_empty}_{list} x = \text{is\_empty}_{tree} y)$

---

`val mem:`  
 $t \rightarrow \text{int} \rightarrow \text{bool}$   $\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}. \sigma(x, y) \Rightarrow (i = j) \Rightarrow (\text{mem}_{list} x i = \text{mem}_{tree} y j)$

---

`val add:`  
 $t \rightarrow \text{int} \rightarrow t$   $\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}. \sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

---

`val if_empty:`  
 $t \rightarrow 'a \rightarrow 'a \rightarrow 'a$   $\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta. \forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta. \sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow \rho(\text{if\_empty}_{list} x a b, \text{if\_empty}_{tree} y c d)$

---

# 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:`  
 $t \rightarrow \text{bool}$   $\forall x : t_{list}. \forall y : t_{tree}. \sigma(x, y) \Rightarrow (\text{is\_empty}_{list} x = \text{is\_empty}_{tree} y)$

---

`val mem:`  
 $t \rightarrow \text{int} \rightarrow \text{bool}$   $\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}. \sigma(x, y) \Rightarrow (i = j) \Rightarrow (\text{mem}_{list} x i = \text{mem}_{tree} y j)$

---

`val add:`  
 $t \rightarrow \text{int} \rightarrow t$   $\forall x : t_{list}. \forall y : t_{tree}. \forall i : \text{Int}. \forall j : \text{Int}. \sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\text{add}_{list} x i, \text{add}_{tree} y j)$

---

`val if_empty:`  
 $t \rightarrow 'a \rightarrow 'a \rightarrow 'a$   $\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta. \forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta. \sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow \rho(\text{if\_empty}_{list} x a b, \text{if\_empty}_{tree} y c d)$

---

# The relational interpretation

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:

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

## Relational interpretation: free variables

If  $T$  is  $\alpha_i$  then

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

## Relational interpretation: products

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

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

## Relational interpretation: sums

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

$$\begin{aligned} \llbracket T \rrbracket[\vec{\rho}] &= (x : T[\vec{A}], y : T[\vec{B}]). \\ &\quad \exists u' : T'[\vec{A}]. \exists v' : T'[\vec{B}]. \\ &\quad x = \text{inl}(u') \wedge y = \text{inl}(v') \\ &\quad \wedge \llbracket T' \rrbracket[\vec{\rho}](u', v') \\ &\vee \\ &\quad \exists u'' : T''[\vec{A}]. \exists v'' : T''[\vec{B}]. \\ &\quad x = \text{inr}(u'') \wedge y = \text{inr}(v'') \\ &\quad \wedge \llbracket T'' \rrbracket[\vec{\rho}](u'', v'') \end{aligned}$$

## Relational interpretation: functions

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

$$\begin{aligned} \llbracket T \rrbracket[\vec{\rho}] &= (f : T[\vec{A}], g : T[\vec{B}]). \\ &\quad \forall u : T'[\vec{A}]. \forall v : T'[\vec{B}]. \\ &\quad \llbracket T' \rrbracket[\vec{\rho}](u, v) \Rightarrow \llbracket T'' \rrbracket[\vec{\rho}](f u, g v) \end{aligned}$$

## Relational interpretation: universals

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

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

## Relational interpretation: existentials

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

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

---

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

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

---

## Relations between values

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

## Relational abstraction

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

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

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

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

$x = y \iff$

$x = \text{pack } \gamma, u \text{ as } \exists \alpha. T[\alpha, B_1, \dots, B_n]$

$\wedge y = \text{pack } \delta, v \text{ as } \exists \alpha. T[\alpha, B_1, \dots, B_n]$

$\wedge \llbracket T \rrbracket[\sigma, =_{B_1}, \dots, =_{B_n}](u, v)$



## Relational abstraction

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

# Invariants

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

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

# Invariants

Given a type  $T$  with free variable  $\alpha$ :

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

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

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

# Invariants

Note that:

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

So:

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

## Identity extension

## Identity extension

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

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

Next time

# Parametricity