Abstraction

Leo White

Jane Street

January 2016

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

$\Lambda\alpha$::*.λp:Bool.λx:α.λy:α.
  if p [α] x y

$\Lambda\alpha$::*.$\Lambda\beta$::*.λp:Bool.λx:α.λy:β.
  if p [$\exists\gamma.\gamma$]
    (**pack** α, x **as** $\exists\gamma.\gamma$)
    (**pack** β, y **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
```

$\forall\alpha$::*. Bool $\rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$

$\forall\alpha$::*.$\forall\beta$::*. 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))
```

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 Fω

Existential types

Existential types

$$
\begin{array}{l}
\texttt{NatSetImpl} = \\
\quad \lambda\alpha{::}*. \\
\qquad\quad \alpha \\
\qquad\quad \times\ (\alpha \to \texttt{Bool}) \\
\qquad\quad \times\ (\texttt{Nat} \to \alpha \to \texttt{Bool}) \\
\qquad\quad \times\ (\texttt{Nat} \to \alpha \to \alpha) \\
\qquad\quad \times\ (\texttt{Nat} \to \alpha \to \alpha) \\
\qquad\quad \times\ (\alpha \to \texttt{List Nat})
\end{array}
$$

$$
\begin{array}{l}
\texttt{empty} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_1\ \texttt{s} \\
\texttt{is\_empty} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_2\ \texttt{s} \\
\texttt{mem} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_3\ \texttt{s} \\
\texttt{add} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_4\ \texttt{s} \\
\texttt{remove} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_5\ \texttt{s} \\
\texttt{to\_list} = \Lambda\alpha{::}*.\lambda \texttt{s}{:}\texttt{NatSetImpl}\ \alpha.\pi_6\ \texttt{s}
\end{array}
$$

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

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

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:

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

Universal quantifications:

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

Existential quantifications:

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

Relations:

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

Term equality:

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

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 t_list = int list

let empty_list = []

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

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



let add_list x t =
  if (mem_list x t) then t
  else x :: t
```

Changing implementations

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

Changing implementations

```
type t_tree =
    | Empty
    | Node of t_tree * int * t_tree

let empty_tree = Empty

let is_empty_tree = function
  | Empty -> true
  | _ -> false

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

let rec add_tree 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(add_tree x l, y, r)
        else Node(l, y, add_tree x r)

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

Relations between types

$\texttt{type t}_{list} = \texttt{int list} \quad \sim \quad$

```
type t_tree =
  | Empty
  | Node of t_tree * int * t_tree
```

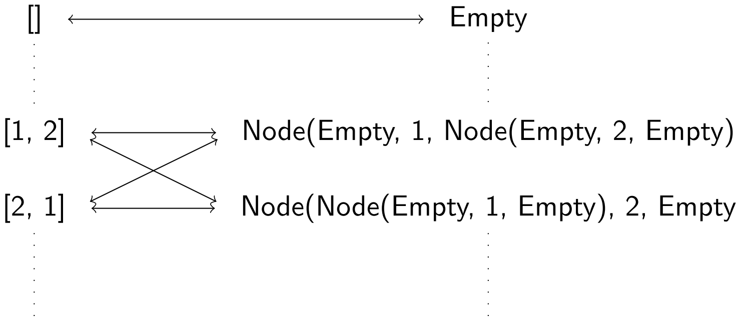
Relations between types

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

$[] \longleftrightarrow$ Empty

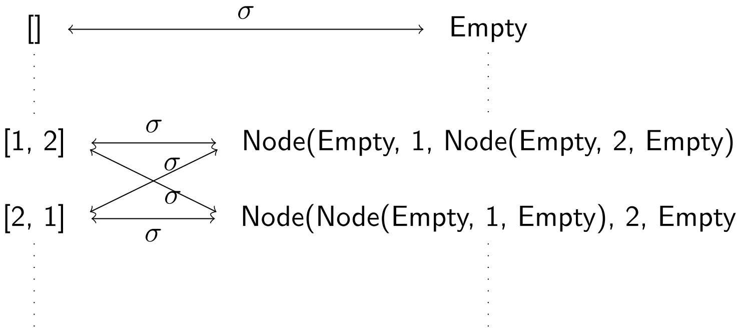
Relations between types

$$\texttt{type } t_{list} = \texttt{int list} \quad \sim \quad \begin{array}{l}\texttt{type } t_{tree} = \\ \quad | \ \texttt{Empty} \\ \quad | \ \texttt{Node of } t_{tree} \texttt{ * int * } t_{tree}\end{array}$$



Relations between types

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



Relations between values

$\texttt{let empty}_{list} = []$ $\sim$ $\texttt{let empty}_{tree} = \texttt{Empty}$

Relations between values

```
let empty_list = []          ~          let empty_tree = Empty
```

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

Relations between values

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

$\sim$

```
let is_empty_tree = function
  | Empty -> true
  | _ -> false
```

Relations between values

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

$\sim$

```
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 mem_list x = function
  | [] -> false
  | y :: rest ->
      if x = y then true
      else mem_list x rest
```

$\sim$

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

Relations between values

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

$\sim$

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

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

Relations between values

```
let add_list x t =
  if (mem_list x t) then t
  else x :: t
```

∼

```
let rec add_tree 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(add_tree x l, y, r)
      else
        Node(l, y, add_tree x r)
```

Relations between values

```
let add_list x t =
  if (mem_list x t) then t
  else x :: t
```

$\sim$

```
let rec add_tree 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(add_tree x l, y, r)
        else
          Node(l, y, add_tree x r)
```

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

Relations between values

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

$\sim$

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

Relations between values

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

$\sim$

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

$$
\begin{aligned}
&\forall \gamma.\, \forall \delta. \\
&\quad \forall x : t_{list}.\, \forall y : t_{tree}.\, \forall a : \gamma.\, \forall b : \gamma.\, \forall c : \delta.\, \forall d : \delta. \\
&\qquad \sigma(x, y) \Rightarrow (a = c) \Rightarrow (b = d) \Rightarrow \\
&\qquad\quad (\text{if\_empty}_{list}\, x\, a\, b \;=\; \text{if\_empty}_{tree}\, y\, c\, d)
\end{aligned}
$$

Relations between values

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

$$\texttt{if\_empty}_{list}\ \texttt{t}\ 5\ 6 \qquad \sim \qquad \texttt{if\_empty}_{tree}\ \texttt{s}\ 5\ 6$$

Relations between values

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

$$\texttt{if\_empty}_{list}\ \texttt{t 5 6} \quad \sim \quad \texttt{if\_empty}_{tree}\ \texttt{s 5 6}$$

$$\texttt{if\_empty}_{list}\ \texttt{t t (add}_{list}\ \texttt{t 1)}$$
$$\sim$$
$$\texttt{if\_empty}_{tree}\ \texttt{s s (add}_{tree}\ \texttt{s 1)}$$

Relations between values

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

$$\texttt{if\_empty}_{list}\ \texttt{t}\ 5\ 6 \quad \sim \quad \texttt{if\_empty}_{tree}\ \texttt{s}\ 5\ 6$$

$$\texttt{if\_empty}_{list}\ \texttt{t}\ \texttt{t}\ (\texttt{add}_{list}\ \texttt{t}\ 1)$$
$$\sim$$
$$\texttt{if\_empty}_{tree}\ \texttt{s}\ \texttt{s}\ (\texttt{add}_{tree}\ \texttt{s}\ 1)$$

$$\texttt{if\_empty}_{list}\ \texttt{t}\ \text{mem}_{list}\ \text{mem}_{list}$$
$$\sim$$
$$\texttt{if\_empty}_{tree}\ \texttt{t}\ \text{mem}_{tree}\ \text{mem}_{tree}$$

Relations between values

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

$\sim$

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

$\sim$

```
let if_empty_tree 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`	$\sigma(\mathsf{empty}_{list},\ \mathsf{empty}_{tree})$
`val is_empty:`	`t -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \sigma(x, y) \Rightarrow (\mathsf{is\_empty}_{list}\, x\ = \mathsf{is\_empty}_{tree}\, y)$
`val mem:`	`t -> int -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.\ \sigma(x, y) \Rightarrow (i = j) \Rightarrow (\mathsf{mem}_{list}\, x\, i\ = \mathsf{mem}_{tree}\, y\, j)$
`val add:`	`t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.\ \sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\mathsf{add}_{list}\, x\, i,\ \mathsf{add}_{tree}\, y\, j)$
`val if_empty:`	`t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.\ \forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.\ \sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow \rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Relations between values

`val empty:` `t`	$\sigma(\mathsf{empty}_{list},\ \mathsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$\forall x : t_{list}. \forall y : t_{tree}.$ $\sigma(x, y) \Rightarrow (\mathsf{is\_empty}_{list}\, x\ = \mathsf{is\_empty}_{tree}\, y)$
`val mem:` `t -> int -> bool`	$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\mathsf{mem}_{list}\, x\, i\ = \mathsf{mem}_{tree}\, y\, j)$
`val add:` `t -> int -> t`	$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\mathsf{add}_{list}\, x\, i,\ \mathsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

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`	$\sigma(\mathsf{empty}_{list},\ \mathsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.$ $\sigma(x, y) \Rightarrow (\mathsf{is\_empty}_{list}\, x\ = \mathsf{is\_empty}_{tree}\, y)$
`val mem:` `t -> int -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\mathsf{mem}_{list}\, x\, i\ = \mathsf{mem}_{tree}\, y\, j)$
`val add:` `t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\mathsf{add}_{list}\, x\, i,\ \mathsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Relational substitution

Given:

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

We define the relation:

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

Relational substitution: free variables

If T is α_i then

$$T[\vec{\rho}] \quad = \quad \rho_i$$

Relational substitution: products

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

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

Relational substitution: sums

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

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

Relational substitution: functions

If T is $T' \to T''$ then

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

Relational substitution: univerals

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

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

Relational substitution: existentials

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

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

Relations between values

`val empty:` `t`	$\sigma(\textsf{empty}_{list},\ \textsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.$ $\sigma(x, y) \Rightarrow (\textsf{is\_empty}_{list}\, x \ = \ \textsf{is\_empty}_{tree}\, y)$
`val mem:` `t -> int -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\textsf{mem}_{list}\, x\, i \ = \ \textsf{mem}_{tree}\, y\, j)$
`val add:` `t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\textsf{add}_{list}\, x\, i,\ \textsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\textsf{if\_empty}_{list}\, x\, a\, b,\ \textsf{if\_empty}_{tree}\, y\, c\, d)$

Relations between values

`val empty:` `t`	$(\alpha)[\sigma](\mathsf{empty}_{list},\ \mathsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.$ $\sigma(x, y) \Rightarrow (\mathsf{is\_empty}_{list}\, x\ =\ \mathsf{is\_empty}_{tree}\, y)$
`val mem:` `t -> int -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\mathsf{mem}_{list}\, x\, i\ =\ \mathsf{mem}_{tree}\, y\, j)$
`val add:` `t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\mathsf{add}_{list}\, x\, i,\ \mathsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Relations between values

`val empty:` `t`	$(\alpha)[\sigma](\textsf{empty}_{list},\ \textsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$(\alpha \to \gamma)[\sigma, =_{\textsf{Bool}}](\textsf{is\_empty}_{list},\ \textsf{is\_empty}_{tree})$
`val mem:` `t -> int -> bool`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\textsf{mem}_{list}\, x\, i\ =\ \textsf{mem}_{tree}\, y\, j)$
`val add:` `t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\textsf{add}_{list}\, x\, i,\ \textsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\textsf{if\_empty}_{list}\, x\, a\, b,\ \textsf{if\_empty}_{tree}\, y\, c\, d)$

Relations between values

`val empty:` `t`	$(\alpha)[\sigma](\mathsf{empty}_{list},\ \mathsf{empty}_{tree})$
`val is_empty:` `t -> bool`	$(\alpha \to \gamma)[\sigma, =_{\mathsf{Bool}}](\mathsf{is\_empty}_{list},\ \mathsf{is\_empty}_{tree})$
`val mem:` `t -> int -> bool`	$(\alpha \to \beta \to \gamma)[\sigma, =_{\mathsf{Int}}, =_{\mathsf{Bool}}](\mathsf{mem}_{list},\ \mathsf{mem}_{tree})$
`val add:` `t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.\ \sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\mathsf{add}_{list}\, x\, i,\ \mathsf{add}_{tree}\, y\, j)$
`val if_empty:` `t -> 'a -> 'a -> 'a`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.\ \forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.\ \sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow \rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Relations between values

`val empty:`	`t`	$(\alpha)[\sigma](\text{empty}_{list},\ \text{empty}_{tree})$
`val is_empty:`	`t -> bool`	$(\alpha \to \gamma)[\sigma, =_{\text{Bool}}](\text{is\_empty}_{list},\ \text{is\_empty}_{tree})$
`val mem:`	`t -> int -> bool`	$(\alpha \to \beta \to \gamma)[\sigma, =_{\text{Int}}, =_{\text{Bool}}](\text{mem}_{list},\ \text{mem}_{tree})$
`val add:`	`t -> int -> t`	$(\alpha \to \beta \to \alpha)[\sigma, =_{\text{Int}}](\text{add}_{list},\ \text{add}_{tree})$
`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`	$(\alpha)[\sigma](\mathsf{empty}_{list}, \mathsf{empty}_{tree})$
`val is_empty:`	
`t -> bool`	$(\alpha \to \gamma)[\sigma, =_{\mathsf{Bool}}](\mathsf{is\_empty}_{list}, \mathsf{is\_empty}_{tree})$
`val mem:`	
`t -> int -> bool`	$(\alpha \to \beta \to \gamma)[\sigma, =_{\mathsf{Int}}, =_{\mathsf{Bool}}](\mathsf{mem}_{list}, \mathsf{mem}_{tree})$
`val add:`	
`t -> int -> t`	$(\alpha \to \beta \to \alpha)[\sigma, =_{\mathsf{Int}}](\mathsf{add}_{list}, \mathsf{add}_{tree})$
`val if_empty:`	
`t -> 'a -> 'a -> 'a`	$(\forall \delta.\, \alpha \to \delta \to \delta \to \delta)[\sigma](\mathsf{if\_empty}_{list}, \mathsf{if\_empty}_{tree})$

Relations between values

$$
\begin{aligned}
&(\alpha \\
&\quad \times (\alpha \to \gamma) \\
&\quad \times (\alpha \to \beta \to \gamma) \\
&\quad \times (\alpha \to \beta \to \alpha) \\
&\quad \times (\forall \delta.\, \alpha \to \delta \to \delta \to \delta))[\sigma, =_{\mathsf{Int}}, =_{\mathsf{Bool}}](\mathsf{set}_{list}, \mathsf{set}_{tree})
\end{aligned}
$$

Relational abstraction

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

$$\forall \beta_1. \ldots \forall \beta_n. \forall x : (\exists \alpha. T). \forall y : (\exists \alpha. T).$$

$$x = y \quad \Leftrightarrow \quad \begin{array}{l} \exists \gamma.\ \exists \delta.\ \exists \sigma \subset \gamma \times \delta. \\ \quad \exists u : T[\gamma, \beta_1, \ldots \beta_n].\ \exists v : T[\delta, \beta_1, \ldots \beta_n]. \\ \qquad x = \textsf{pack}\ \gamma,\ u\ \textsf{as}\ T[\vec{A}] \\ \qquad \wedge\ y = \textsf{pack}\ \delta,\ v\ \textsf{as}\ T[\vec{B}] \\ \qquad \wedge\ T[\sigma, =_{\beta_1}, \ldots, =_{\beta_n}](u,\ v) \end{array}$$

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 γ as a relation $\rho \subset \gamma \times \gamma$:

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

Invariants

Given a type T with free variable α:

$$\begin{aligned}&\forall f : (\forall \alpha . T[\alpha] \to \alpha).\\ &\qquad \forall \gamma.\ \forall \rho \subset \gamma \times \gamma.\ \forall x : T[\gamma].\\ &\qquad\qquad T[\rho](x, x) \Rightarrow \rho(f[\gamma]\, x,\ f[\gamma]\, x)\end{aligned}$$

Invariants

Note that:

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

So:

$$\forall \rho \subset \gamma \times \gamma. \quad T[\rho](u, u) \Rightarrow$$
$$\rho\Big(\texttt{open}\ (\texttt{pack}\ \gamma,\ u\texttt{as}\ \exists\alpha.\ T[\alpha])\ \texttt{as}\ x,\ \alpha\texttt{in}\ t,\ \texttt{open}\ (\texttt{pack}\ \gamma,\ u\texttt{as}\ \exists\alpha.\ T[\alpha])\ \texttt{as}\ x,\ \alpha\texttt{in}\ t\Big)$$

Identity extension

Identity extension

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

$$\begin{aligned}&\forall \alpha_1 . \ldots \forall \alpha_n . \forall x : T . \forall y : T .\\ &\qquad (x =_T y) \quad \Leftrightarrow \quad T[=_{\alpha_1}, \ldots, =_{\alpha_n}](x,\, y)\end{aligned}$$

Next time

Parametricity