Relational abstraction

- Changing implementations
- Preserving invariants
- Phantom types

Changing implementations

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
```

Changing implementations

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

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
```

Changing implementations

```
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)

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

Changing implementations

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

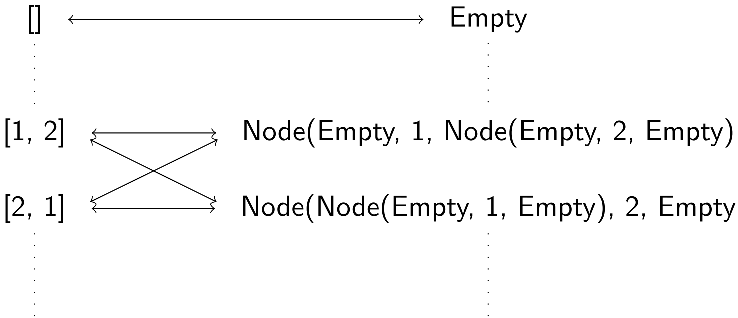
Changing implementations

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

$[] \longleftrightarrow$ Empty

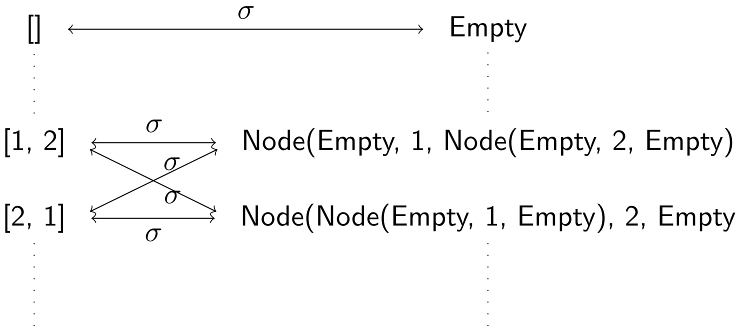
Changing implementations

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



Changing implementations

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



Changing implementations

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

Changing implementations

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

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

Changing implementations

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

$\sim$

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

Changing implementations

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

$\sim$

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

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

Changing implementations

```
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
```

Changing implementations

```
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
```

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

Changing implementations

```
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)
```

Changing implementations

```
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)
```

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

Changing implementations

```
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
```

Changing implementations

```
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
```

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

Changing implementations

Given t : t_{list} and s : t_{tree} such that σ(t, s):

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

Changing implementations

Given t : t_{list} and s : t_{tree} such that σ(t, s):

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

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

$$\sim$$

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

Changing implementations

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

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

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

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

Changing implementations

```
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
```

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

Changing implementations

```
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
```

$$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$$
$$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$$
$$\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$$
$$\rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$$

Changing implementations

`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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\ x\, a\, b,\ \mathsf{if\_empty}_{tree}\ y\, c\, d)$

Changing implementations

`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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

`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`	$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a, c) \Rightarrow \rho(b, d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\, x\, a\, b, \text{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

`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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

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`	$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\ x\ a\ b,\ \text{if\_empty}_{tree}\ y\ c\ d)$

Changing implementations

`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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

`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`	$\forall x : t_{list}. \forall y : t_{tree}. \forall i : Int. \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\mathsf{mem}_{list}\, x\, i\ = \mathsf{mem}_{tree}\, y\, j)$
`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`	$\forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

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`	$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\mathsf{if\_empty}_{list}\, x\, a\, b,\ \mathsf{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

`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`	$\forall\gamma.\ \forall\delta.\ \forall\rho \subset \gamma \times \delta.\ \forall x : t_{list}.\ \forall y : t_{tree}.\ \forall a : \gamma.\ \forall b : \gamma.\ \forall c : \delta.\ \forall d : \delta.\ \sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow \rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$

Changing implementations

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

Changing implementations

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

Abstraction

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

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

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

Parametricity

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

$$\begin{aligned}
&\forall \beta_1. \ldots \forall \beta_n. \forall x : (\forall \alpha. T). \\
&\quad \forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta. \\
&\qquad T[\rho, =_{\beta_1}, \ldots, =_{\beta_n}](x[\gamma],\ x[\delta])
\end{aligned}$$

Identity extension

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

$$\forall \alpha_1. \dots \forall \alpha_n. \forall x : T. \forall y : T.$$
$$(x =_T y) \quad \Leftrightarrow \quad T[=_{\alpha_1}, \dots, =_{\alpha_n}](x,\ y)$$

Invariants

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
```

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:

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

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
```

Phantom types

```
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
```

Phantom types

```
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)
```

Phantom types

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

Phantom types

```
# 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
```

Phantom types

```
# 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
```

Phantom types

```
# 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".
```

Phantom types

```
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)
```

Phantom types

```
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)
```

Phantom types

```
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
```

Phantom types

```
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
```

Phantom types

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

val brand : 'a array -> 'a brand
```

Phantom types

```
# 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
```

Phantom types

```
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
```

Phantom types

```
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
```

Phantom types

```
let position idx = idx

let zero = 0
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
```

Phantom types

```
    let set = Array.set

    let get = Array.get
end
```

Phantom types

```
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
```

Phantom types

```
let set = Array.unsafe_set

let get = Array.unsafe_get
```

Next time

GADTs

Next time

GADTs

(First-class phantom types)