

Finally tagless, partially evaluated

Tagless staged interpreters for simpler typed languages

Jacques Carette Oleg Kiselyov Chung-chieh Shan

December 2, 2011



The goal of this talk

Write your interpreter by deforesting the object language,
to exhibit more static safety in a simpler type system.

There's interpretation everywhere

A fold on an inductive data type is an interpreter of a domain-specific language.

contract

grammar

music

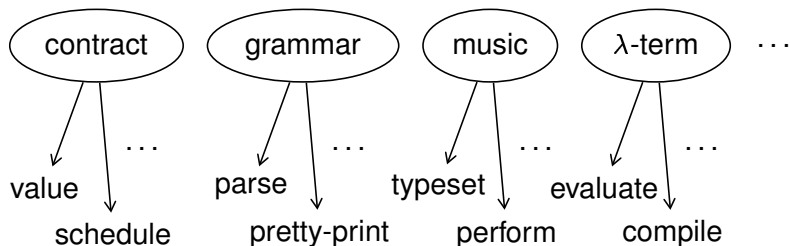
λ -term

...



There's interpretation everywhere

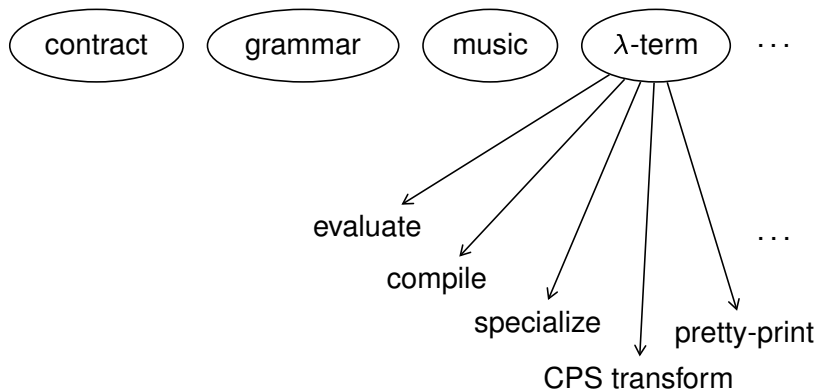
A fold on an inductive data type is an interpreter of a domain-specific language.



The same language can be interpreted in many useful ways.

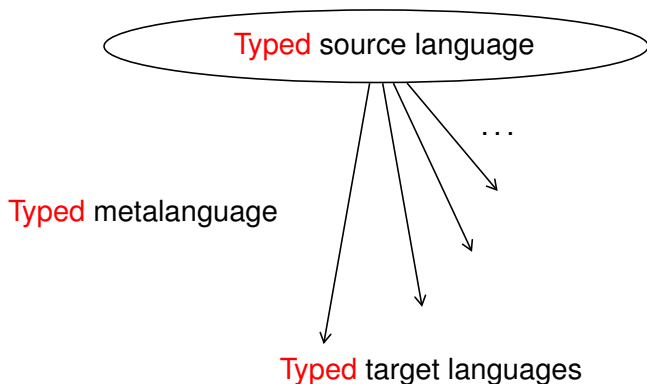
There's interpretation everywhere

A fold on an inductive data type is an interpreter of a domain-specific language.



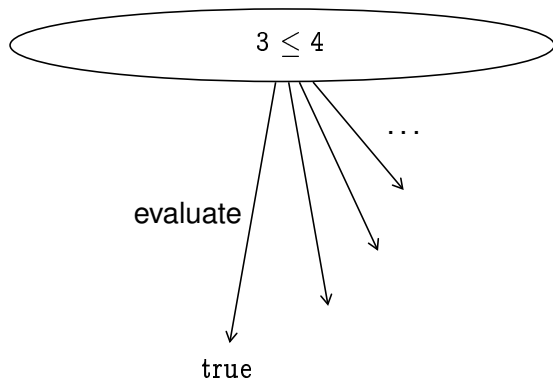
We focus on the λ -calculus as an example.

Simple type preservation



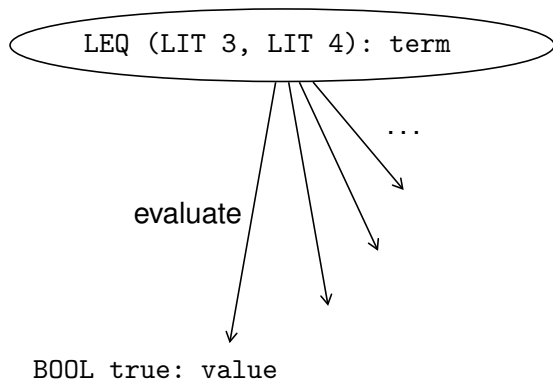
It should be obvious in the metalanguage that interpreting a well-typed source term yields a well-typed target term.

Simple type preservation



It should be obvious in the metalanguage that interpreting a well-typed source term yields a well-typed target term.

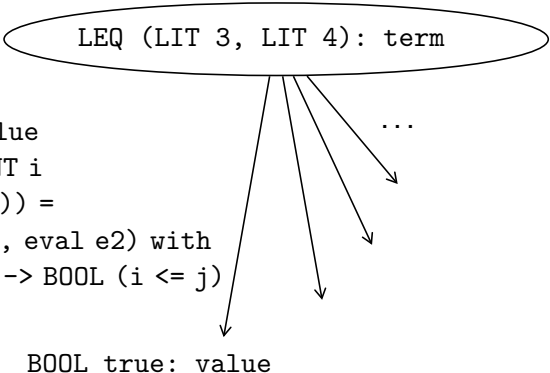
Simple type preservation



It should be obvious in the metalanguage that interpreting a well-typed source term yields a well-typed target term.

Simple type preservation

LEQ (LIT 3, LIT 4): term



eval: term -> value

eval (LIT i) = INT i

eval (LEQ (e1, e2)) =
 match (eval e1, eval e2) with
 (INT i, INT j) -> BOOL (i <= j)

BOOL true: value

It should be obvious in the metalanguage that interpreting a well-typed source term yields a well-typed target term.

Simple type preservation

LEQ (LIT 3, LIT 4): term

eval: term -> value

eval (LIT i) = INT i

eval (LEQ (e1, e2)) =

match (eval e1, eval e2) with
(INT i, INT j) -> BOOL (i <= j)

BOOL true: value

The term should be **well-typed**, so **pattern matching** in the metalanguage should always **obviously** succeed.

Simple type preservation

LEQ (LIT 3, LIT 4): term

eval: term -> value

eval (LIT i) = INT i

eval (LEQ (e1, e2)) =
 match (eval e1, eval e2) with
 (INT i, INT j) -> BOOL (i <= j)

BOOL true: value

The term should be **closed**, so **environment lookup** in the metalanguage should always **obviously** succeed.

Simple type preservation

LEQ (LIT 3, LIT 4): **bool** term

eval: 'a term -> 'a value

eval (LIT i) = INT i

eval (LEQ (e1, e2)) =

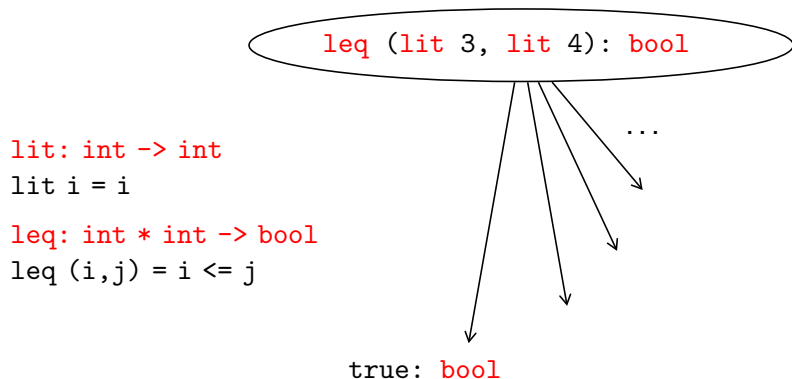
 match (eval e1, eval e2) with

 (INT i, INT j) -> BOOL (i <= j)

BOOL true: **bool** value

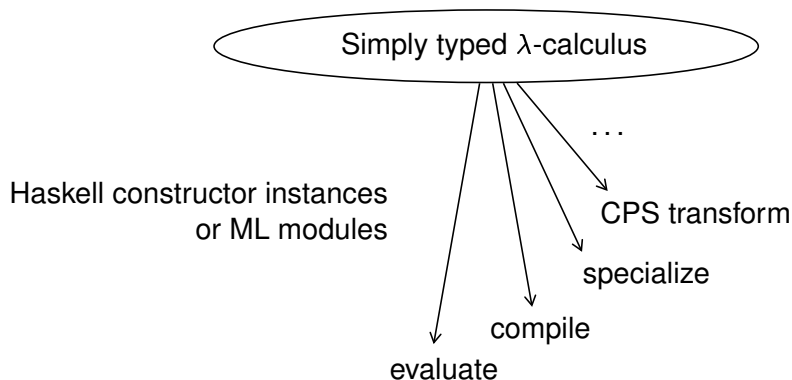
Previous solutions use (and motivate) fancier types:
generalized abstract data types (GADT) and dependent types.

Simple type preservation



Our simple solution is to be **finally tagless**:
replace term constructors by cogen functions.

Simple type preservation



The term accommodates **multiple interpretations** by abstracting over the cogen functions and their types.

Outline

► **The object language**

As a constructor class in Haskell

As a module signature in ML

Tagless interpretation

Evaluation

Compilation

Type-indexed types

Partial evaluation

CPS transformation

The object language

$$\frac{\begin{array}{c} [x : t_1] \\ \vdots \\ e : t_2 \end{array}}{\lambda x. e : t_1 \rightarrow t_2}$$

$$\frac{\begin{array}{c} [f : t_1 \rightarrow t_2] \\ \vdots \\ e : t_1 \rightarrow t_2 \end{array}}{\text{fix } f. e : t_1 \rightarrow t_2}$$

$$\frac{e_1 : t_1 \rightarrow t_2 \quad e_2 : t_1}{e_1 e_2 : t_2}$$

$$\frac{n \text{ is an integer}}{n : \text{int}}$$

$$\frac{b \text{ is a boolean}}{b : \text{bool}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \times e_2 : \text{int}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \leq e_2 : \text{bool}}$$

$$\frac{e : \text{bool} \quad e_1 : t \quad e_2 : t}{\text{if } e \text{ then } e_1 \text{ else } e_2 : t}$$

$\lambda x. \text{fix } f. \lambda n.$

$\text{if } n \leq 0 \text{ then } 1 \text{ else}$

$x \times f(n - 1)$

$: \text{int} \rightarrow \text{int} \rightarrow \text{int}$

The object language

$$\frac{\begin{array}{c} [x : t_1] \\ \vdots \\ e : t_2 \end{array}}{\lambda x. e : t_1 \rightarrow t_2}$$

$$\frac{\begin{array}{c} [f : t_1 \rightarrow t_2] \\ \vdots \\ e : t_1 \rightarrow t_2 \end{array}}{\text{fix } f. e : t_1 \rightarrow t_2}$$

$$\frac{e_1 : t_1 \rightarrow t_2 \quad e_2 : t_1}{e_1 e_2 : t_2}$$

$$\frac{n \text{ is an integer}}{n : \text{int}}$$

$$\frac{b \text{ is a boolean}}{b : \text{bool}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \times e_2 : \text{int}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \leq e_2 : \text{bool}}$$

$$\frac{e : \text{bool} \quad e_1 : t \quad e_2 : t}{\text{if } e \text{ then } e_1 \text{ else } e_2 : t}$$

$\lambda x. \text{fix } f. \lambda n.$
 $\text{if } n \leq 0 \text{ then } 1 \text{ else}$
 $x \times f(n - 1)$

$: \text{int} \rightarrow \text{int} \rightarrow \text{int}$

The object language as a constructor class

```
class Symantics repr where
  int :: Int -> repr Int
  lam :: (repr a -> repr b) -> repr (a -> b)
  fix :: (repr a -> repr a) -> repr a
  app :: repr (a -> b) -> repr a -> repr b
  add :: repr Int -> repr Int -> repr Int
  if_ :: repr Bool -> repr a -> repr a -> repr a
```

```
 $\lambda x. \text{fix } f. \lambda n.$   
 $\text{if } n \leq 0 \text{ then } 1 \text{ else}$   
 $x \times f(n - 1)$ 
```

```
: int  $\rightarrow$  int  $\rightarrow$  int
```

The object language as a constructor class

class Symantics repr where

int :: Int -> repr Int

lam :: (repr a -> repr b) -> repr (a -> b)

fix :: (repr a -> repr a) -> repr a

app :: repr (a -> b) -> repr a -> repr b

add :: repr Int -> repr Int -> repr Int

if_ :: repr Bool -> repr a -> repr a -> repr a

$\lambda x. \text{fix } f. \lambda n.$

$\text{if } n \leq 0 \text{ then } 1 \text{ else}$

$x \times f(n - 1)$

$: \text{int} \rightarrow \text{int} \rightarrow \text{int}$

The object language as a constructor class

class Symantics repr where

```
int  :: Int -> repr Int
lam  :: (repr a -> repr b) -> repr (a -> b)
fix  :: (repr a -> repr a) -> repr a
app  :: repr (a -> b) -> repr a -> repr b
add  :: repr Int -> repr Int -> repr Int
if_  :: repr Bool -> repr a -> repr a -> repr a
```

Object term \longrightarrow Haskell term

$\lambda x. \text{fix } f. \lambda n.$	<code>lam (\x -> fix (\f -> lam (\n -></code>
<code>if n ≤ 0 then 1 else</code>	<code>if_ (leq n (int 0)) (int 1)</code>
$x \times f(n - 1)$	<code>(mul x (app f (add n (int (-1))))))</code>

`: int → int → int` `:: Symantics repr => repr (Int -> Int -> Int)`

The object language as a constructor class

class Symantics repr where

```
int  :: Int -> repr Int
lam  :: (repr a -> repr b) -> repr (a -> b)
fix  :: (repr a -> repr a) -> repr a
app  :: repr (a -> b) -> repr a -> repr b
add  :: repr Int -> repr Int -> repr Int
if_  :: repr Bool -> repr a -> repr a -> repr a
```

Object term \longrightarrow Haskell term

```
λx. fix f. λn.      lam (\x -> fix (\f -> lam (\n ->
if n ≤ 0 then 1 else if_ (leq n (int 0)) (int 1)
x × f(n - 1)        (mul x (app f (add n (int (-1))))))))))
```

```
: int → int → int      :: Symantics repr => repr (Int -> Int -> Int)
```

The object language as a constructor class

```
class Symantics repr where
  int :: Int -> repr Int
  lam :: (repr a -> repr b) -> repr (a -> b)
  fix :: (repr a -> repr a) -> repr a
  app :: repr (a -> b) -> repr a -> repr b
  add :: repr Int -> repr Int -> repr Int
  if_ :: repr Bool -> repr a -> repr a -> repr a
```

Object term \longrightarrow Haskell term

```
 $\lambda x. \text{fix } f. \lambda n.$           lam (\x -> fix (\f -> lam (\n ->
if  $n \leq 0$  then 1 else         if_ (leq n (int 0)) (int 1)
 $x \times f(n - 1)$              (mul x (app f (add n (int (-1)))))))))
```

```
: int  $\rightarrow$  int  $\rightarrow$  int      :: Symantics repr => repr (Int -> Int -> Int)
```

The object language as a constructor class

class Symantics repr where

```
int :: Int -> repr Int
lam :: (repr a -> repr b) -> repr (a -> b)
fix  :: (repr a -> repr a) -> repr a
app  :: repr (a -> b) -> repr a -> repr b
add  :: repr Int -> repr Int -> repr Int
if_  :: repr Bool -> repr a -> repr a -> repr a
```

Object term \longrightarrow Haskell term

```
 $\lambda x. \text{fix } f. \lambda n.$           lam (\x -> fix (\f -> lam (\n ->
if  $n \leq 0$  then 1 else      if_ (leq n (int 0)) (int 1)
 $x \times f(n - 1)$           (mul x (app f (add n (int (-1)))))))))
```

```
: int  $\rightarrow$  int  $\rightarrow$  int      :: Symantics repr => repr (Int -> Int -> Int)
```


The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x -> 'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a -> 'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
      -> (('c,'a) repr as 'x)
end
```

```
 $\lambda x. \text{fix } f. \lambda n.$   
 $\text{if } n \leq 0 \text{ then } 1 \text{ else}$   
 $x \times f(n - 1)$ 
```

```
: int  $\rightarrow$  int  $\rightarrow$  int
```

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x -> 'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a -> 'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
      -> (('c,'a) repr as 'x)
end
```

```
λx. fix f. λn.  
if n ≤ 0 then 1 else  
x × f(n - 1)  
  
: int → int → int
```

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x->'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a->'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
    -> (('c,'a) repr as 'x)
end
```

Object term \longrightarrow ML functor

$\lambda x. \text{fix } f. \lambda n.$	<code>lam (fun x -> fix (fun f -> lam (fun n -></code>
<code>if $n \leq 0$ then 1 else</code>	<code>if_ (leq n (int 0)) (fun () -> int 1)</code>
<code>$x \times f(n - 1)$</code>	<code>(fun () -> mul x (app f (add n (int (-1))))))</code>
<code>: int \rightarrow int \rightarrow int</code>	<code>('c, int -> int -> int) repr</code>

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x -> 'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a -> 'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
    -> (('c,'a) repr as 'x)
end
```

Object term \longrightarrow ML functor

$\lambda x. \text{fix } f. \lambda n.$	$\text{lam (fun } x \rightarrow \text{fix (fun } f \rightarrow \text{lam (fun } n \rightarrow$
$\text{if } n \leq 0 \text{ then } 1 \text{ else}$	$\text{if_ (leq } n \text{ (int 0)) (fun () } \rightarrow \text{int 1)}$
$x \times f(n - 1)$	$\text{(fun () } \rightarrow \text{mul } x \text{ (app } f \text{ (add } n \text{ (int (-1)))))))))$
$: \text{int} \rightarrow \text{int} \rightarrow \text{int}$	$(\text{'c, int } \rightarrow \text{int } \rightarrow \text{int}) \text{ repr}$

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x -> 'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a->'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
    -> (('c,'a) repr as 'x)
end
```

Object term \longrightarrow ML functor

$\lambda x. \text{fix } f. \lambda n.$	<code>lam (fun x -> fix (fun f -> lam (fun n -></code>
<code>if $n \leq 0$ then 1 else</code>	<code>if_ (leq n (int 0)) (fun () -> int 1)</code>
<code>$x \times f(n - 1)$</code>	<code>(fun () -> mul x (app f (add n (int (-1)))))))))</code>
<code>: int \rightarrow int \rightarrow int</code>	<code>('c, int -> int -> int) repr</code>

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x->'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a->'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
    -> (('c,'a) repr as 'x)
end
```

Object term \longrightarrow ML functor

$\lambda x. \text{fix } f. \lambda n.$	<code>lam (fun x -> fix (fun f -> lam (fun n -></code>
<code>if $n \leq 0$ then 1 else</code>	<code>if_ (leq n (int 0)) (fun () -> int 1)</code>
$x \times f(n - 1)$	<code>(fun () -> mul x (app f (add n (int (-1)))))))))</code>
$: \text{int} \rightarrow \text{int} \rightarrow \text{int}$	<code>('c, int -> int -> int) repr</code>

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x -> 'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a -> 'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
      -> (('c,'a) repr as 'x)
end
```

ML functor

```
lam (fun x -> fix (fun f -> lam (fun n ->
  if_ (leq n (int 0)) (fun () -> int 1)
  (fun () -> mul x (app f (add n (int (-1))))))))))
('c, int -> int -> int) repr
```

The object language as a module signature

```
module type Symantics = sig type ('c,'a) repr
  val int: int -> ('c,int) repr
  val lam: (('c,'a) repr -> ('c,'b) repr) -> ('c,'a->'b) repr
  val fix: ('x->'x) -> (('c,'a->'b) repr as 'x)
  val app: ('c,'a->'b) repr -> ('c,'a) repr -> ('c,'b) repr
  val add: ('c,int) repr -> ('c,int) repr -> ('c,int) repr
  val if_: ('c,bool) repr -> (unit -> 'x) -> (unit -> 'x)
    -> (('c,'a) repr as 'x)
end
```

ML functor

```
module POWER (S:Symantics) = struct open S
  let term () = lam (fun x-> fix (fun f-> lam (fun n->
    if_ (leq n (int 0)) (fun ()->int 1)
    (fun ()->mul x (app f (add n (int (-1))))))))))
end: functor (S:Symantics) -> sig
  val term: unit -> ('c, int -> int -> int) S.repr
end
```


Composing object programs as functors

$(\lambda x. \text{fix } f. \lambda n. \text{if } n \leq 0 \text{ then } 1 \text{ else } x \times f(n - 1))$

Composing object programs as functors

$\lambda x. (\lambda x. \text{fix } f. \lambda n. \text{if } n \leq 0 \text{ then } 1 \text{ else } x \times f(n - 1)) \ x \ 7$

Composing object programs as functors

$\lambda x. (\lambda x. \text{fix } f. \lambda n. \text{if } n \leq 0 \text{ then } 1 \text{ else } x \times f(n - 1)) \ x \ 7$

```
module POWER7 (S:Symantics) = struct open S
  module P = POWER(S)
  let term () = lam (fun x -> app (app (P.term ()) x)
                                (int 7))
end: functor (S:Symantics) -> sig
  val term: unit -> ('c, int->int) S.repr
end
```

Outline

The object language

As a constructor class in Haskell

As a module signature in ML

► Tagless interpretation

Evaluation

Compilation

Type-indexed types

Partial evaluation

CPS transformation

Tagless interpretation: Evaluation

No worry about pattern matching or environment lookup!
Well-typed source programs **obviously** don't go wrong.

```
module R = struct
  type ('c,'a) repr = 'a
  let int (x:int) = x
  let lam f      = fun x -> f x
  let fix g      = let rec f n = g f n in f
  let app e1 e2  = e1 e2
  let add e1 e2  = e1 + e2
  let if_ e e1 e2 = if e then e1 () else e2 ()
end
```

Tagless interpretation: Evaluation

No worry about pattern matching or environment lookup!
Well-typed source programs **obviously** don't go wrong.

```
module R = struct
  type ('c,'a) repr = 'a
  let int (x:int) = x
  let lam f      = fun x -> f x
  let fix g      = let rec f n = g f n in f
  let app e1 e2 = e1 e2
  let add e1 e2 = e1 + e2
  let if_ e e1 e2 = if e then e1 () else e2 ()
end
module POWER7R = POWER7(R)
▶ POWER7R.term () 2
128
```

Tagless interpretation: Evaluation

No worry about pattern matching or environment lookup!
Well-typed source programs **obviously** don't go wrong.

```
module R = struct
  type ('c,'a) repr = 'a
  let int (x:int) = x
  let lam f      = fun x -> f x
  let fix g      = let rec f n = g f n in f
  let app e1 e2  = e1 e2
  let add e1 e2  = e1 + e2
  let if_ e e1 e2 = if e then e1 () else e2 ()
end
```

Tagless interpretation: Evaluation

No worry about pattern matching or environment lookup!
Well-typed source programs **obviously** don't go wrong.

```
module R = struct
  type ('c,'a) repr = 'a
  let int (x:int) = x
  let lam f      = fun x -> f x
  let fix g      = let rec f n = g f n in f
  let app e1 e2 = e1 e2
  let add e1 e2 = e1 + e2
  let if_ e e1 e2 = if e then e1 () else e2 ()
end
```


Tagless interpretation: Compilation

No worry about pattern matching or environment lookup!

Well-typed source programs **obviously** translate to well-typed target programs.

```
module C = struct
  type ('c,'a) repr = ('c,'a) code
  let int (x:int) = <x>
  let lam f      = <fun x -> ~(f <x>>>
  let fix g      = <let rec f n = ~(g <f>) n in f>
  let app e1 e2  = <~e1 ~e2>
  let add e1 e2  = <~e1 + ~e2>
  let if_ e e1 e2 = <if ~e then ~(e1 ()) else ~(e2 ())>
end
```

Tagless interpretation: Compilation

No worry about pattern matching or environment lookup!

Well-typed source programs **obviously** translate to well-typed target programs.

```
module C = struct
  type ('c,'a) repr = ('c,'a) code
  let int (x:int) = ⟨x⟩
  let lam f      = ⟨fun x -> ~(f ⟨x⟩)⟩
  let fix g      = ⟨let rec f n = ~(g ⟨f⟩) n in f⟩
  let app e1 e2  = ⟨~e1 ~e2⟩
  let add e1 e2  = ⟨~e1 + ~e2⟩
  let if_ e e1 e2 = ⟨if ~e then ~(e1 ()) else ~(e2 ())⟩
end
module POWER7C = POWER7(C)
▶ POWER7C.term ()
  ⟨fun x -> (fun x -> let rec self = fun x ->
    (fun x -> if x <= 0 then 1 else x * self (x + (-1)))
    x in self) x 7⟩
```

Outline

The object language

As a constructor class in Haskell

As a module signature in ML

Tagless interpretation

Evaluation

Compilation

► **Type-indexed types**

Partial evaluation

CPS transformation

Partial evaluation

```
module P = struct
  type ('c, 'a) repr
    = ???
```

Partial evaluation

```
type ('c,int) repr  
  = ('c,int) code  
  * int option
```

Partial evaluation

```
type ('c,int) repr  
  = ('c,int) code  
  * int option
```

interpret source term
→ dynamic part
→ static part

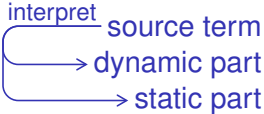
The diagram illustrates the process of partial evaluation. A source term is processed by an 'interpret' function. The result is split into two components: a 'dynamic part' and a 'static part'. The 'dynamic part' is the portion of the source term that remains to be evaluated at runtime, while the 'static part' is the portion that has been pre-evaluated and can be stored as a constant.

Partial evaluation

```
type ('c,int) repr
  = ('c,int) code
  * int option
```

interpret source term 3
dynamic part <3>
static part Some 3

Partial evaluation

type ('c,int) repr		3	x
= ('c,int) code		$\langle 3 \rangle$	$\langle x \rangle$
* int option		Some 3	None

Partial evaluation

type ('c,int) repr	interpret	source term	3	x
= ('c,int) code	→	dynamic part	<3>	<x>
* int option	→	static part	Some 3	None

```
type ('c,int->int) repr
= ('c,int->int) code
* (('c,int) repr ->
  ('c,int) repr) option
```

Partial evaluation

type ('c,int) repr		source term	3	x
= ('c,int) code		dynamic part	$\langle 3 \rangle$	$\langle x \rangle$
* int option		static part	Some 3	None

type ('c,int->int) repr				f
= ('c,int->int) code				$\langle f \rangle$
* (('c,int) repr ->				None
('c,int) repr) option				

Partial evaluation

type ('c,int) repr	<i>interpret</i>	source term	3	x
= ('c,int) code		dynamic part	$\langle 3 \rangle$	$\langle x \rangle$
* int option		static part	Some 3	None

type ('c,int->int) repr		$\lambda x. 1 \times x$		f
= ('c,int->int) code		$\langle \text{fun } x \rightarrow x \rangle$		$\langle f \rangle$
* (('c,int) repr ->		Some (fun r->r)		None
('c,int) repr) option				

Partial evaluation

type ('c,int) repr	interpret	source term	3	x
= ('c,int) code		dynamic part	$\langle 3 \rangle$	$\langle x \rangle$
* int option		static part	Some 3	None

type ('c,int->int) repr		$\lambda x. 1 \times x$		f
= ('c,int->int) code		$\langle \text{fun } x \rightarrow x \rangle$		$\langle f \rangle$
* (('c,int) repr -> ('c,int) repr) option		Some (fun r->r)		None

type ('c,'a) repr				
= ('c,'a) code				
* ??? option				

Partial evaluation

```
type ('c,int) repr
= ('c,int) code
* int option
```

interpret

source term	3	x
dynamic part	$\langle 3 \rangle$	$\langle x \rangle$
static part	Some 3	None

```
type ('c,int->int) repr
= ('c,int->int) code
* (('c,int) repr ->
   ('c,int) repr) option
```

$\lambda x. 1 \times x$	f
$\langle \text{fun } x \rightarrow x \rangle$	$\langle f \rangle$
Some (fun r->r)	None

```
type ('c,'a) repr
= ('c,'a) code
* ('c,'a) static option
```

```
type ('c, int) static = int
```

```
type ('c, bool) static = bool
```

```
type ('c, 'a->'b) static = ('c,'a) repr -> ('c,'b) repr
```

Type-indexed types

```
type ('c, int)      static = int
type ('c, bool)    static = bool
type ('c, 'a -> 'b) static = ('c, 'a) repr -> ('c, 'b) repr
```

Type-indexed types

```
module type Symantics = sig type ('c,'s,'a) repr
  val int: int -> ('c,int,int) repr
  val lam: 'x-> ('c, ('c,'s,'a) repr ->
                ('c,'t,'b) repr as 'x, 'a->'b) repr
  val fix: (('c, ('c,'s,'a) repr -> ('c,'t,'b) repr,
            'a -> 'b) repr as 'x -> 'x) -> 'x
  val app: ('c, ('c,'s,'a) repr ->
            ('c,'t,'b) repr as 'x, 'a->'b) repr -> 'x
  val add: 'x -> 'x -> (('c,int,int) repr as 'x)
  val if_: ('c,bool,bool) repr -> (unit->'x) -> (unit->'x)
            -> (('c,'s,'a) repr as 'x)
end
```

```
type ('c, int)      static = int
type ('c, bool)     static = bool
type ('c, 'a->'b) static = ('c,'a) repr -> ('c,'b) repr
```

Type-indexed types: Partial evaluation

```
module P = struct
  type ('c,'a) repr
    = ('c,'a) code
    * ('c,'a) static option
```

```
type ('c, int)      static = int
type ('c, bool)     static = bool
type ('c, 'a -> 'b) static = ('c,'a) repr -> ('c,'b) repr
```


Type-indexed types: Partial evaluation

```
module P = struct
  type ('c, 's, 'a) repr
    = ('c, 'a) code
    * 's option
  ...
end
```

```
type ('c, int)      static = int
type ('c, bool)    static = bool
type ('c, 'a -> 'b) static = ('c, 'a) repr -> ('c, 'b) repr
```

Type-indexed types: Partial evaluation

```
module P = struct
```

```
  type ('c, 's, 'a) repr
```

```
    = ('c, 'a) code
```

```
    * 's option
```

```
  ...
```

```
end
```

```
module POWER7P = POWER7(P)
```

```
▶ POWER7P.term ()
```

```
  (<fun x -> x*x*x*x*x*x*x*x>, Some <fun>)
```

```
type ('c, int)      static = int
```

```
type ('c, bool)    static = bool
```

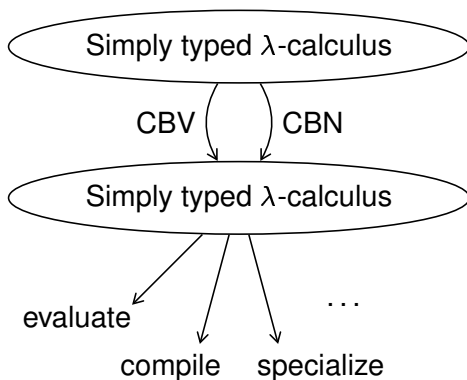
```
type ('c, 'a -> 'b) static = ('c, 'a) repr -> ('c, 'b) repr
```

Type-indexed types: CPS transformation

```
type ('c,'s,'a) repr
  = ('s -> ans) -> ans           (* CBN CPS evaluator *)
  = ('c, ('s -> ans) -> ans) code (* CBN CPS compiler *)
```

```
type ('c, int)      static = int
type ('c, bool)     static = bool
type ('c, 'a -> 'b) static = ('c,'a) repr -> ('c,'b) repr
```

CPS transformations



Payoffs: evaluation order independence, mutable state

Other benefits

Supports initial type-checking

Type-check once, even under λ , then interpret many times.

```
FilePath -> Maybe (exists a. Typeable a =>
                  forall repr. Symantics repr =>
                  repr a)
```

“Typing dynamic typing” (ICFP 2002) works. We have the code.

Preserves sharing in the metalanguage

Compute the interpretation of a repeated object term once, then use it many times.

```
2 × 3 + 2 × 3      let n = mul (int 2) (int 3) in add n n
```

Embed one object language in another

```
(Symantics repr, Symantics' repr') => repr (repr' Int)
```

Other benefits

Supports initial type-checking

Type-check once, even under λ , then interpret many times.

```
FilePath -> Maybe (exists a. Typeable a =>
                  forall repr. Symantics repr =>
                  repr a)
```

“Typing dynamic typing” (ICFP 2002) works. We have the code.

Preserves sharing in the metalanguage

Compute the interpretation of a repeated object term once, then use it many times.

```
2 × 3 + 2 × 3      let n = mul (int 2) (int 3) in add n n
```

Embed one object language in another

```
(Symantics repr, Symantics' repr') => repr (repr' Int)
```

Other benefits

Supports initial type-checking

Type-check once, even under λ , then interpret many times.

```
FilePath -> Maybe (exists a. Typeable a =>
                  forall repr. Symantics repr =>
                  repr a)
```

“Typing dynamic typing” (ICFP 2002) works. We have the code.

Preserves sharing in the metalanguage

Compute the interpretation of a repeated object term once, then use it many times.

```
2 × 3 + 2 × 3      let n = mul (int 2) (int 3) in add n n
```

Embed one object language in another

```
(Symantics repr, Symantics' repr') => repr (repr' Int)
```

Conclusion

Write your interpreter by deforesting the object language

- ▶ An abstract data type family
- ▶ Type-indexed types

Exhibit more static safety in a simpler type system

- ▶ Early, obvious guarantees
- ▶ Supports initial type-checking
- ▶ Preserves sharing in the metalanguage
- ▶ Embed one object language in another