Finally tagless, partially evaluated

Tagless staged interpreters for simpler typed languages

Jacques Carette Oleg Kiselyov Chung-chieh Shan

December 2, 2011

tanakawho on flickr

The goal of this talk

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

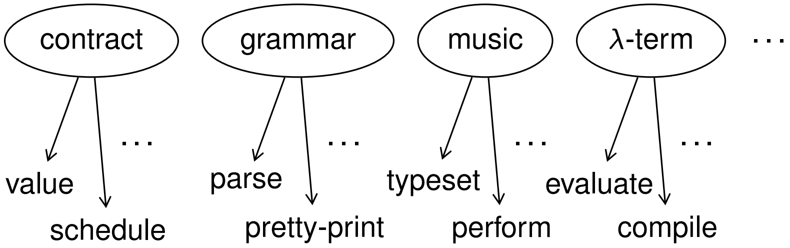
There’s interpretation everywhere

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

contract grammar music λ-term $\cdots$

There’s interpretation everywhere

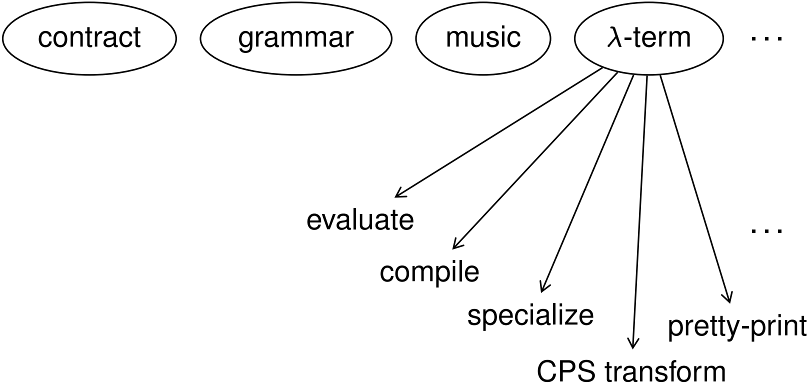
A fold on an inductive data type is an interpreter of a domain-specific language.



The same language can be interpreted in many useful ways.

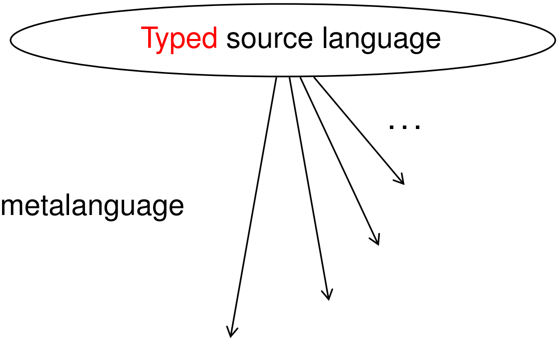
There’s interpretation everywhere

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



Typed source language

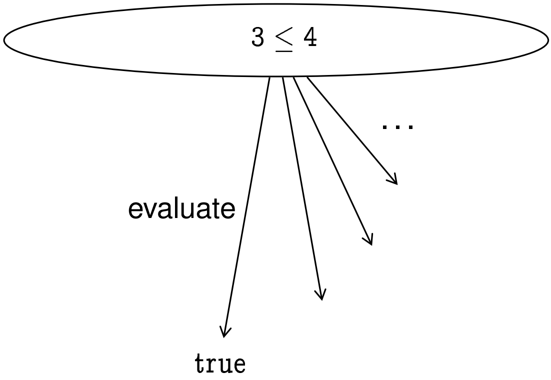
. . .

Typed metalanguage

Typed target languages

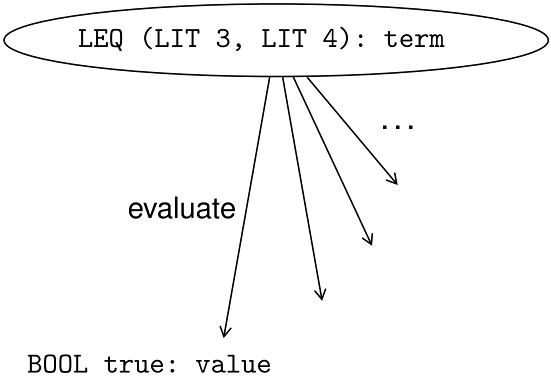
It should be obvious in the metalanguage that interpreting a well-typed source term yields a well-typed target term.

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`

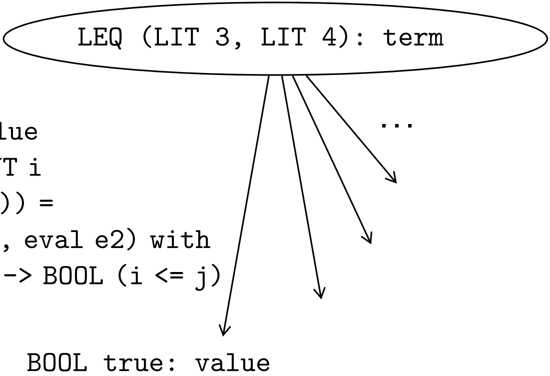
evaluate

...

`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

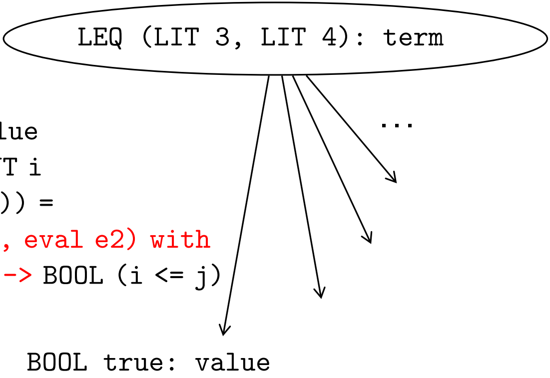
...

BOOL true: value

```
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)
```

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`

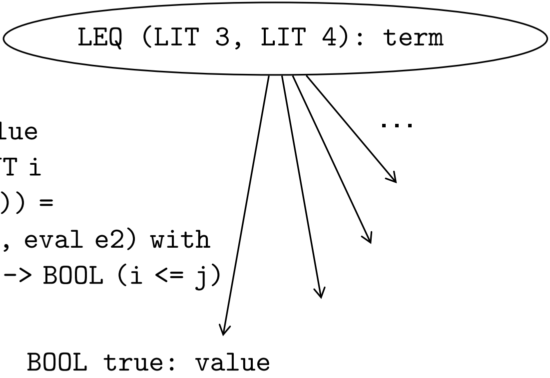
...

```
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.

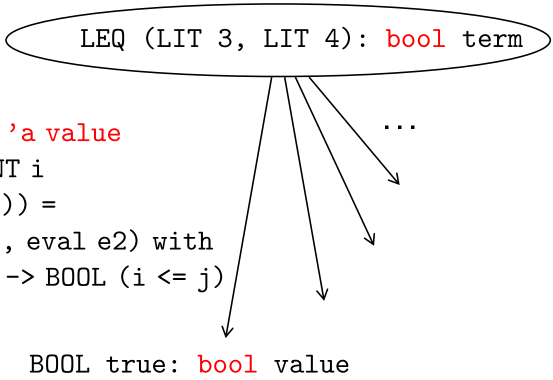
Simple type preservation



```
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)
```

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`

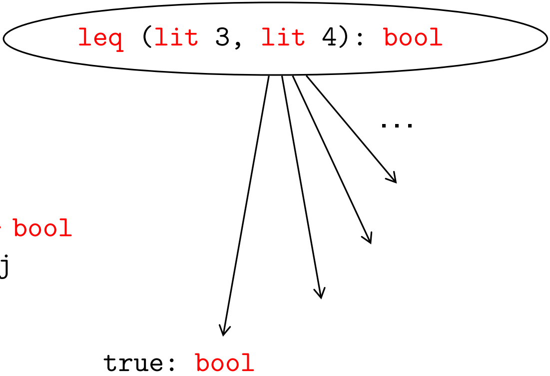
...

```
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



`leq (lit 3, lit 4): bool`

...

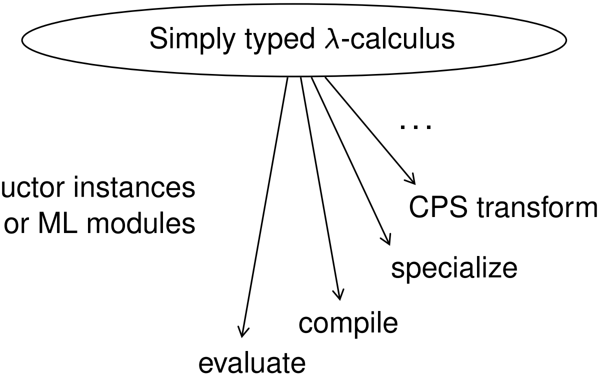
`true: bool`

```
lit: int -> int
lit i = i

leq: int * int -> bool
leq (i,j) = i <= j
```

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

Simple type preservation



Simply typed λ-calculus

. . .

Haskell constructor instances
or ML modules

CPS transform

specialize

compile

evaluate

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

$$\frac{\begin{array}{c}[x : t_1]\\ \vdots \\ e : t_2\end{array}}{\lambda x.\ e : t_1 \to t_2} \qquad \frac{\begin{array}{c}[f : t_1 \to t_2]\\ \vdots \\ e : t_1 \to t_2\end{array}}{\mathsf{fix}\ f.\ e : t_1 \to t_2} \qquad \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{e_1 e_2 : t_2}$$

$$\frac{n \text{ is an integer}}{n : \mathsf{int}} \qquad \frac{b \text{ is a boolean}}{b : \mathsf{bool}} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}$$

$$\frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 \times e_2 : \mathsf{int}} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 \le e_2 : \mathsf{bool}} \qquad \frac{e : \mathsf{bool} \quad e_1 : t \quad e_2 : t}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}$$

$$\begin{array}{l} \lambda x.\ \mathsf{fix}\ f.\ \lambda n. \\ \mathsf{if}\ n \le 0\ \mathsf{then}\ 1\ \mathsf{else} \\ x \times f(n-1) \end{array}$$

$$: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$

The object language

$$\frac{\begin{array}{c}[x : t_1]\\ \vdots\\ e : t_2\end{array}}{\lambda x.\ e : t_1 \to t_2} \qquad \frac{\begin{array}{c}[f : t_1 \to t_2]\\ \vdots\\ e : t_1 \to t_2\end{array}}{\mathrm{fix}\ f.\ e : t_1 \to t_2} \qquad \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{e_1 e_2 : t_2}$$

$$\frac{n \text{ is an integer}}{n : \mathrm{int}} \qquad \frac{b \text{ is a boolean}}{b : \mathrm{bool}} \qquad \frac{e_1 : \mathrm{int} \quad e_2 : \mathrm{int}}{e_1 + e_2 : \mathrm{int}}$$

$$\frac{e_1 : \mathrm{int} \quad e_2 : \mathrm{int}}{e_1 \times e_2 : \mathrm{int}} \qquad \frac{e_1 : \mathrm{int} \quad e_2 : \mathrm{int}}{e_1 \leq e_2 : \mathrm{bool}} \qquad \frac{e : \mathrm{bool} \quad e_1 : t \quad e_2 : t}{\mathrm{if}\ e\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2 : t}$$

$$\begin{array}{l}\lambda x.\ \mathrm{fix}\ f.\ \lambda n.\\ \mathrm{if}\ n \leq 0\ \mathrm{then}\ 1\ \mathrm{else}\\ x \times f(n-1)\end{array}$$

$$: \mathrm{int} \to \mathrm{int} \to \mathrm{int}$$

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
```

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

$: \mathrm{int} \to \mathrm{int} \to \mathrm{int}$

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
```

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

$: \mathrm{int} \to \mathrm{int} \to \mathrm{int}$

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

$\lambda x.\ \mathrm{fix}\ f.\ \lambda n.$	`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))))))))`
$:\ \mathrm{int} \to \mathrm{int} \to \mathrm{int}$	`:: 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

$\lambda x.\ \mathrm{fix}\, f.\ \lambda n.$	`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))))))))`
$: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$	`:: 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 ⟶ Haskell term

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

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

$: \text{int} \to \text{int} \to \text{int}$

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
```

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

$: \text{int} \to \text{int} \to \text{int}$

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 ⟶ ML functor

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

Object term ⟶ ML functor

$\lambda x.\ \mathrm{fix}\ f.\ \lambda n.$	`lam (fun x -> fix (fun f -> lam (fun n ->`
if $n \le 0$ then 1 else	`if_ (leq n (int 0)) (fun () -> int 1)`
$x \times f(n-1)$	`(fun () -> mul x (app f (add n (int (-1))))))))`
$: \mathtt{int} \to \mathtt{int} \to \mathtt{int}$	`('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
```

Object term $\longrightarrow$ ML functor

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

Object term ⟶ ML functor

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

```
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

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

Composing object programs as functors

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

Composing object programs as functors

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

```
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

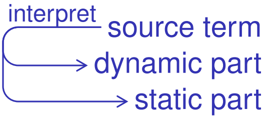
```
module P = struct
  type ('c, 'a) repr
     = ???
```

Partial evaluation

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

Partial evaluation

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



interpret

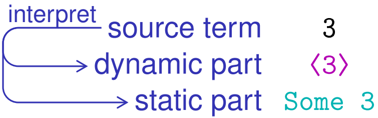
source term

dynamic part

static part

Partial evaluation

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



interpret

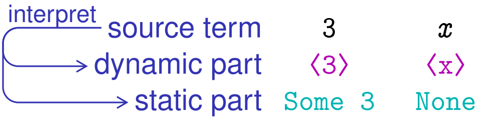
source term 3

dynamic part ⟨3⟩

static part `Some 3`

Partial evaluation

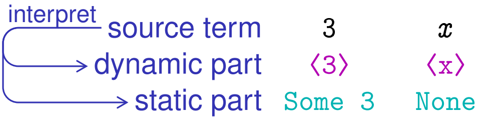
```
type ('c,int) repr
   = ('c,int) code
   * int option
```



source term	3	x
dynamic part	`⟨3⟩`	`⟨x⟩`
static part	`Some 3`	`None`

Partial evaluation

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



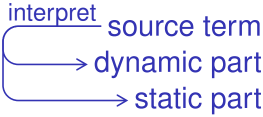
interpret

source term	3	x
dynamic part	⟨3⟩	⟨x⟩
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
   = ('c,int) code
   * int option
```



source term	3	x	f
dynamic part	⟨3⟩	⟨x⟩	⟨f⟩
static part	`Some 3`	`None`	`None`

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

Partial evaluation

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

interpret

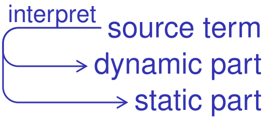
source term	3	x
dynamic part	⟨3⟩	⟨x⟩
static part	`Some 3`	`None`

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

$\lambda x.\, 1 \times x$	f
⟨fun x->x⟩	⟨f⟩
`Some (fun r->r)`	`None`

Partial evaluation

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



interpret

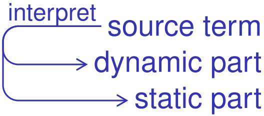
source term	3	x	$\lambda x.\,1\times x$	f
dynamic part	⟨3⟩	⟨x⟩	⟨fun x->x⟩	⟨f⟩
static part	Some 3	None	Some (fun r->r)	None

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

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

Partial evaluation

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



source term	3	x
dynamic part	⟨3⟩	⟨x⟩
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
⟨fun x->x⟩	⟨f⟩
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⟩, Some <fun>)
```

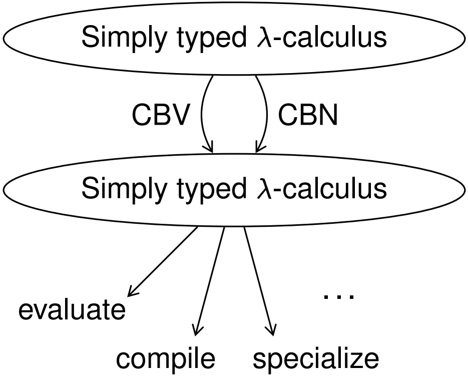
```
  type ('c, int)      static = int
  type ('c, bool)     static = bool
  type ('c, 'a->'b) static = ('c,'a) repr -> ('c,'b) repr
```

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.

$2 \times 3 + 2 \times 3$ `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.

$2 \times 3 + 2 \times 3$ `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.

$2 \times 3 + 2 \times 3$ `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