Recursion

March 2018

Last time: more staging

`.<e>.`

Generalizing **algebraic** optmisation

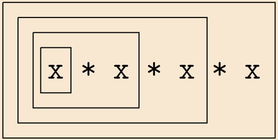
Staging and **effects**

This time: recursion

$\mu\texttt{x.e}$

Staging: a difficulty

Functional programs often use **mutual recursion**.
In MetaOCaml, expressions are built from smaller expressions:



```
.< .~(.< .~(.< .~(.< x >.) * x >.) * x >.) * x >.
```

However, binding groups are **not** built from smaller binding groups:

`let rec` $\mathtt{x}_1$ `=` $\mathtt{e}_1$
    `and` $\mathtt{x}_2$ `=` $\mathtt{e}_2$
    ...
    `and` $\mathtt{x}_n$ `=` $\mathtt{e}_n$

If n varies: **easy** to generate an expression with n multiplications
but **hard** to generate `let rec` with n bindings

Abstraction and builtin operators

Common pattern: **replace built-in operation** with custom version

Example: replace + with a partially-static variation

(**Benefit**: generate code with better performance)

Example: use $\gg=$ in place of `let`

(**Benefit**: can vary the semantics, defining new effects)

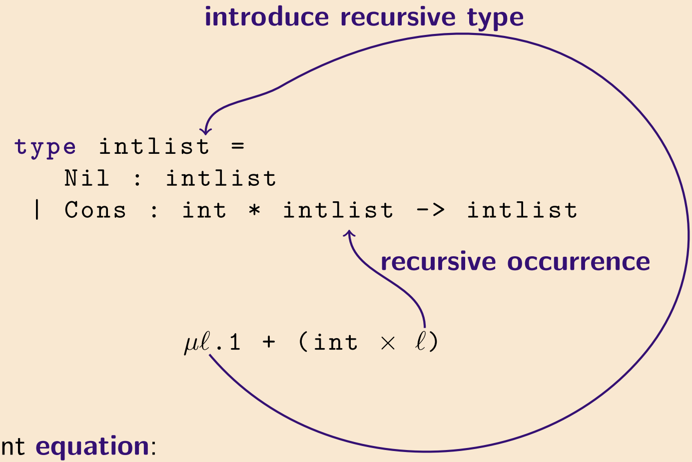
Example: define type equality $\equiv$ as a library

(**Benefit**: can switch to other relations: subtyping, $\simeq$, &c.)

This lecture: define our own version of `let rec`

(**Benefit**: generate mutual recursion with a single operator)

fixed point operators (μ)



introduce recursive type

```
type intlist =
   Nil : intlist
 | Cons : int * intlist -> intlist
```

recursive occurrence

$$\mu\ell.\mathtt{1 + (int} \times \ell\mathtt{)}$$

Fixed point **equation**:

$$\mu\ell.A \quad = \quad A[\ell := \mu\ell.A]$$

Fixed points in Haskell (CBN)

Fixed points in **Haskell**:

```
fix :: (a → a) → a
fix f = f (fix f)
```

Example: `len`, directly:

```
len [] = 0
len (_:t) = 1 + len t
```

Example: `len`, with `fix`:

```
len’ self [] = 0
len’ self (_:t) = 1 + self t

len = fix len’
```

Example: len (Haskell)

```
len’ self [] = 0                    fix :: (a → a) → a
len’ self (_:t) = 1 + self t        fix f = f (fix f)

len = fix len’

   len (’a’ : ’b’ : [])
```

Example: len (Haskell)

```
len' self [] = 0
len' self (_:t) = 1 + self t
```

```
fix :: (a → a) → a
fix f = f (fix f)
```

```
len = fix len'
```

```
     len ('a' : 'b' : [])
⇝    fix len' ('a' : 'b' : [])
```

Example: len (Haskell)

```
len' self [] = 0                    fix :: (a → a) → a
len' self (_:t) = 1 + self t        fix f = f (fix f)

len = fix len'
```

```
     len ('a' : 'b' : [])
⇝    fix len' ('a' : 'b' : [])
⇝    len' (fix len') ('a' : 'b' : [])
```

Example: len (Haskell)

```
len' self [] = 0                      fix :: (a → a) → a
len' self (_:t) = 1 + self t          fix f = f (fix f)

len = fix len'
```

```
     len ('a' : 'b' : [])
⇝    fix len' ('a' : 'b' : [])
⇝    len' (fix len') ('a' : 'b' : [])
⇝    1 + len' (fix len') ('b' : [])
```

Example: len (Haskell)

```
len’ self [] = 0                     fix :: (a → a) → a
len’ self (_:t) = 1 + self t         fix f = f (fix f)

len = fix len’
```

```
    len (’a’ : ’b’ : [])
⇝   fix len’ (’a’ : ’b’ : [])
⇝   len’ (fix len’) (’a’ : ’b’ : [])
⇝   1 + len’ (fix len’) (’b’ : [])
⇝   1 + (1 + (len’ (fix len’) []))
```

Example: len (Haskell)

```
len’ self [] = 0                       fix :: (a → a) → a
len’ self (_:t) = 1 + self t           fix f = f (fix f)

len = fix len’
```

```
     len (’a’ : ’b’ : [])
⇝    fix len’ (’a’ : ’b’ : [])
⇝    len’ (fix len’) (’a’ : ’b’ : [])
⇝    1 + len’ (fix len’) (’b’ : [])
⇝    1 + (1 + (len’ (fix len’) []))
⇝    1 + (1 + 0)
```

Example: len (Haskell)

```
len' self [] = 0                     fix :: (a → a) → a
len' self (_:t) = 1 + self t         fix f = f (fix f)

len = fix len'
```

```
    len ('a' : 'b' : [])
⇝   fix len' ('a' : 'b' : [])
⇝   len' (fix len') ('a' : 'b' : [])
⇝   1 + len' (fix len') ('b' : [])
⇝   1 + (1 + (len' (fix len') []))
⇝   1 + (1 + 0)
⇝   2
```

Fixed points in OCaml (CBV)

The fixed point operator, translated:

```
val fix : ('a → 'a) → 'a

let rec fix f = f (fix f)
```

The `len` function, via `fix`:

```
let len' self = function
  | []   → 0
  | _::t → 1 + self t

let len = fix len'
```

Example: len (OCaml)

```
let len' self = function
  | []   → 0
  | _::t → 1 + self t

let len = fix len'
```

```
let rec fix f = f (fix f)
```

```
len ('a'::'b'::[])
```

Example: len (OCaml)

```
let len' self = function
  | []   → 0
  | _::t → 1 + self t

let len = fix len'
```

```
let rec fix f = f (fix f)
```

```
    len ('a'::'b'::[])
⇝   fix len' ('a'::'b'::[])
```

Example: len (OCaml)

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fix len’
```

```
let rec fix f = f (fix f)
```

```
     len (’a’::’b’::[])
⇝    fix len’ (’a’::’b’::[])
⇝    len’ (fix len’) (’a’::’b’::[])
```

Example: len (OCaml)

```
let len' self = function              let rec fix f = f (fix f)
  | []   → 0
  | _::t → 1 + self t

let len = fix len'
```

```
    len ('a'::'b'::[])
⇝   fix len' ('a'::'b'::[])
⇝   len' (fix len') ('a'::'b'::[])
⇝   len' (len' (fix len')) ('a'::'b'::[])
```

Example: len (OCaml)

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fix len’
```

```
let rec fix f = f (fix f)
```

```
    len (’a’::’b’::[])
⇝   fix len’ (’a’::’b’::[])
⇝   len’ (fix len’) (’a’::’b’::[])
⇝   len’ (len’ (fix len’)) (’a’::’b’::[])
⇝   len’ (len’ (len’ (fix len’))) (’a’::’b’::[])
```

Example: len (OCaml)

```
let len' self = function          let rec fix f = f (fix f)
  | []    → 0
  | _::t → 1 + self t

let len = fix len'
```

```
    len ('a'::'b'::[])
⇝   fix len' ('a'::'b'::[])
⇝   len' (fix len') ('a'::'b'::[])
⇝   len' (len' (fix len')) ('a'::'b'::[])
⇝   len' (len' (len' (fix len'))) ('a'::'b'::[])
```

⇝ **. . . infinite regress!**

Fixed points in OCaml, eta expanded

In eager languages, **eta-expand** the fixed point operator:

```
val fixV : (('a → 'b) → ('a → 'b)) → ('a → 'b)
let rec fixV f x = f (fixV f) x
```

Observation: the type has changed, too.

`fixV` can only be used to create recursive **functions**

Example: `len` with `fixV`

```
let len' self = function
  | []    → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
let rec fixV f x = f (fixV f) x
```

```
    len ('a'::'b'::[])
```

Example: `len` with `fixV`

```
let len' self = function
  | []   → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
let rec fixV f x = f (fixV f) x
```

```
    len ('a'::'b'::[])
⇝   fixV len' ('a'::'b'::[])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fixV len’
```

```
let rec fixV f x = f (fixV f) x
```

```
      len (’a’::’b’::[])
⇝    fixV len’ (’a’::’b’::[])
⇝    len’ (fixV len’) (’a’::’b’::[])
```

Example: `len` with `fixV`

```
let len' self = function          let rec fixV f x = f (fixV f) x
  | []   → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
    len ('a'::'b'::[])
⇝   fixV len' ('a'::'b'::[])
⇝   len' (fixV len') ('a'::'b'::[])
⇝   len' (fun x→ fixV len' x) ('a'::'b'::[])
```

Example: `len` with `fixV`

```
let len' self = function        let rec fixV f x = f (fixV f) x
  | []   → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
     len ('a'::'b'::[])
⇝    fixV len' ('a'::'b'::[])
⇝    len' (fixV len') ('a'::'b'::[])
⇝    len' (fun x→ fixV len' x) ('a'::'b'::[])
⇝    1 + (fun x→ fixV len' x) ('b'::[])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fixV len’
```

```
let rec fixV f x = f (fixV f) x
```

```
     len (’a’::’b’::[])
⇝    fixV len’ (’a’::’b’::[])
⇝    len’ (fixV len’) (’a’::’b’::[])
⇝    len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝    1 + (fun x→ fixV len’ x) (’b’::[])
⇝    1 + fixV len’ (’b’::[])
```

Example: `len` with `fixV`

```
let len' self = function        let rec fixV f x = f (fixV f) x
  | []   → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
    len ('a'::'b'::[])
⇝   fixV len' ('a'::'b'::[])
⇝   len' (fixV len') ('a'::'b'::[])
⇝   len' (fun x→ fixV len' x) ('a'::'b'::[])
⇝   1 + (fun x→ fixV len' x) ('b'::[])
⇝   1 + fixV len' ('b'::[])
⇝   1 + len' (fixV len') ('b'::[])
```

Example: `len` with `fixV`

```
let len' self = function
  | []    → 0
  | _::t  → 1 + self t

let len = fixV len'
```

```
let rec fixV f x = f (fixV f) x
```

```
     len ('a'::'b'::[])
⇝    fixV len' ('a'::'b'::[])
⇝    len' (fixV len') ('a'::'b'::[])
⇝    len' (fun x→ fixV len' x) ('a'::'b'::[])
⇝    1 + (fun x→ fixV len' x) ('b'::[])
⇝    1 + fixV len' ('b'::[])
⇝    1 + len' (fixV len') ('b'::[])
⇝    1 + len' (fun x→ fixV len' x) ('b'::[])
```

Example: `len` with `fixV`

```
let len' self = function      let rec fixV f x = f (fixV f) x
  | []   → 0
  | _::t → 1 + self t

let len = fixV len'
```

```
     len ('a'::'b'::[])
⇝    fixV len' ('a'::'b'::[])
⇝    len' (fixV len') ('a'::'b'::[])
⇝    len' (fun x→ fixV len' x) ('a'::'b'::[])
⇝    1 + (fun x→ fixV len' x) ('b'::[])
⇝    1 + fixV len' ('b'::[])
⇝    1 + len' (fixV len') ('b'::[])
⇝    1 + len' (fun x→ fixV len' x) ('b'::[])
⇝    1 + (1 + (fun x→ fixV len' x) [])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t
```

```
let rec fixV f x = f (fixV f) x
```

```
let len = fixV len’
```

```
     len (’a’::’b’::[])
⇝   fixV len’ (’a’::’b’::[])
⇝   len’ (fixV len’) (’a’::’b’::[])
⇝   len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝   1 + (fun x→ fixV len’ x) (’b’::[])
⇝   1 + fixV len’ (’b’::[])
⇝   1 + len’ (fixV len’) (’b’::[])
⇝   1 + len’ (fun x→ fixV len’ x) (’b’::[])
⇝   1 + (1 + (fun x→ fixV len’ x) [])
⇝   1 + (1 + fixV len’ [])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t
```

```
let rec fixV f x = f (fixV f) x
```

```
let len = fixV len’
```

```
    len (’a’::’b’::[])
⇝   fixV len’ (’a’::’b’::[])
⇝   len’ (fixV len’) (’a’::’b’::[])
⇝   len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝   1 + (fun x→ fixV len’ x) (’b’::[])
⇝   1 + fixV len’ (’b’::[])
⇝   1 + len’ (fixV len’) (’b’::[])
⇝   1 + len’ (fun x→ fixV len’ x) (’b’::[])
⇝   1 + (1 + (fun x→ fixV len’ x) [])
⇝   1 + (1 + fixV len’ [])
⇝   1 + (1 + len’ (fixV len’) [])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fixV len’
```

```
let rec fixV f x = f (fixV f) x
```

```
     len (’a’::’b’::[])
⇝    fixV len’ (’a’::’b’::[])
⇝    len’ (fixV len’) (’a’::’b’::[])
⇝    len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝    1 + (fun x→ fixV len’ x) (’b’::[])
⇝    1 + fixV len’ (’b’::[])
⇝    1 + len’ (fixV len’) (’b’::[])
⇝    1 + len’ (fun x→ fixV len’ x) (’b’::[])
⇝    1 + (1 + (fun x→ fixV len’ x) [])
⇝    1 + (1 + fixV len’ [])
⇝    1 + (1 + len’ (fixV len’) [])
⇝    1 + (1 + len’ (fun x→ fixV len’ x) [])
```

Example: `len` with `fixV`

```
let len’ self = function
  | []   → 0
  | _::t → 1 + self t

let len = fixV len’
```

```
let rec fixV f x = f (fixV f) x
```

```
     len (’a’::’b’::[])
⇝    fixV len’ (’a’::’b’::[])
⇝    len’ (fixV len’) (’a’::’b’::[])
⇝    len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝    1 + (fun x→ fixV len’ x) (’b’::[])
⇝    1 + fixV len’ (’b’::[])
⇝    1 + len’ (fixV len’) (’b’::[])
⇝    1 + len’ (fun x→ fixV len’ x) (’b’::[])
⇝    1 + (1 + (fun x→ fixV len’ x) [])
⇝    1 + (1 + fixV len’ [])
⇝    1 + (1 + len’ (fixV len’) [])
⇝    1 + (1 + len’ (fun x→ fixV len’ x) [])
⇝    1 + (1 + 0)
```

Example: `len` with `fixV`

```
let len’ self = function     let rec fixV f x = f (fixV f) x
  | []   → 0
  | _::t → 1 + self t

let len = fixV len’
```

```
     len (’a’::’b’::[])
⇝    fixV len’ (’a’::’b’::[])
⇝    len’ (fixV len’) (’a’::’b’::[])
⇝    len’ (fun x→ fixV len’ x) (’a’::’b’::[])
⇝    1 + (fun x→ fixV len’ x) (’b’::[])
⇝    1 + fixV len’ (’b’::[])
⇝    1 + len’ (fixV len’) (’b’::[])
⇝    1 + len’ (fun x→ fixV len’ x) (’b’::[])
⇝    1 + (1 + (fun x→ fixV len’ x) [])
⇝    1 + (1 + fixV len’ [])
⇝    1 + (1 + len’ (fixV len’) [])
⇝    1 + (1 + len’ (fun x→ fixV len’ x) [])
⇝    1 + (1 + 0)
⇝    2
```

Mutual recursion with `fix`

So far: **monomorphic recursion** of **one** function.

What about **polymorphic** recursion and **mutual** recursion?

Example: `even` and `odd` functions

```
even n = n == 0 || odd (n - 1)
odd  n = n /= 0 && even (n - 1)
```

`even` and `odd` functions using `fix` to build a **pair**

```
(even, odd) = fix (λ~(even, odd) →
                      ((λn → n == 0 || odd (n - 1)),
                       (λn → n /= 0 && even (n - 1))))
```

Mutual recursion in OCaml

The type of `fix` in Haskell:

```
fix :: (a → a) → a
```

`fix` can build values of **any type**,
e.g. functions (`len`), pairs (**`even`**, **`odd`**)

The type of `fixV` in OCaml:

```
val fixV : (('a → 'b) → ('a → 'b)) → ('a → 'b)
```

`fixV` can only build values of **function type**

One (slightly unsatisfactory) solution: add a `unit` argument

```
let eo = fixV (fun eo () →
    ((fun n → n =  0 || snd (eo ()) (n - 1)),
     (fun n → n <> 0 && fst (eo ()) (n - 1))))
```

Mutual recursion and type isomorphisms

Recall: `a → b` corresponds to b^a.

We have the following **type isomorphism**:

$$\texttt{a * a} \quad\equiv\quad \texttt{a}^2 \quad\equiv\quad \texttt{2} \rightarrow \texttt{a} \quad\equiv\quad \texttt{bool} \rightarrow \texttt{a}$$

which turns the pair type into the function type we need for `fixV`:

```
type eo = Even | Odd  (* isomorphic to bool *)

let eo = fixV (fun eo → function
    Even → (fun n → n =  0 || eo Odd (n - 1))
  | Odd  → (fun n → n <> 0 && eo Even (n - 1)))

let even, odd = eo Even, eo Odd
```

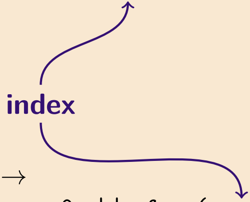
Mutual recursion: from 2 to n

This **indexed** approach generalizes from 2 functions to n functions.

Example: residuals modulo n (generalizes `even` and `odd`):

```
let rec f_0 x = x  = 0 || f_{n-1} (x-1)
    and f_1 x = x <> 0 && f_0 (x-1)
    ...
    and f_{n-1} x = x <> 0 && f_{n-2} (x-1)
```

With `fixV`:

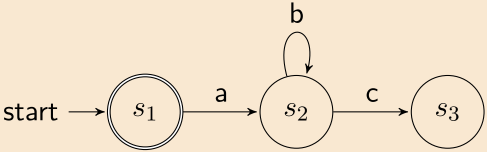

index

```
let fs n = fixV (fun fs i →
   if i = 0 then fun x → x  = 0 || fs (n - 1) (x - 1)
   else          fun x → x <> 0 && fs (i - 1) (x - 1))

let zero_mod_4 = fs 4 0
```

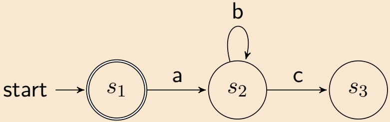
(Note: since n unknown, harder to do this using product types!)

Mutual recursion: state machine



```
let rec s1 = function
  | 'a' :: k → s2 k
  | _ → failwith "no transition"
and s2 = function
  | 'b' :: k → s2 k
  | 'c' :: k → s3 k
  | _ → failwith "no transition"
and s3 = function
  | [] → true
  | _ → false
```

Mutual recursion: state machine, with `fixV`



Define an **index type**:

```
type state = S1 | S2 | S3
```

Encode mutual recursion as **indexed recursion**:

```
let s = fixV (fun s -> function
 | S1 → (function 'a' :: k → s S2 k
                 | _ → failwith "no transition")
 | S2 → (function 'b' :: k → s S2 k
                 | 'c' :: k → s S3 k
                 | _ → failwith "no transition")
 | S3 → (function [] → true
                 | _ → false))
```

Staging indexed recursion

Staging indexed recursion

What we have: a single `fixV` for n-ary mutual recursion

Plan: stage `fixV` to generate mutually-recursive bindings

Starting point: binding-time analysis.
(What's **static**? What's **dynamic**?)

```
type eo = Even | Odd

let eo = fixV (fun eo → function
    Even → (fun n → n =  0 || eo Odd (n - 1))
  | Odd  → (fun n → n <> 0 && eo Even (n - 1)))
```

Analysis: **indexes** (`Even`, `Odd`) **static**; **everything else dynamic**.
(Indexes disappear from generated code; everything else remains.)

Staging plan

1 stage `fixV` to generate `let rec` bindings
(rather than performing the recursive calls directly)

2 add **memoization** on indexes
recursive references with same index generate a single binding

3 add support for `let rec` bodies:

$$\begin{array}{l}\texttt{let rec } x_1 = e_1 \\ \quad\quad\;\; \texttt{and } x_2 = e_2 \\ \quad\quad\;\; \ldots \\ \quad\quad\;\; \texttt{and } x_n = e_n \\ \boxed{\texttt{in } e}\end{array}$$

Mutual recursion: staging types

Analysis: **indexes** (`Even`, `Odd`) **static**; **everything else dynamic**

Staged `fixV` type:

```
val fixVS : (('a → 'b code) → ('a → 'b code)) →
            ('a → 'b code)
```

`fixV` specialized for `even` and `odd`:

```
((eo → (int -> bool) code) → (eo → (int -> bool))) →
 (eo → (int -> bool) code)
```

Mutual recursion: staging terms

Analysis: **indexes** (`Even`, `Odd`) **static**; **everything else dynamic**

Staged `fixV` in action:

```
fixVS (fun eo → function
 | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
 | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
```

Generate code by supplying the final parameter:

```
let even = fixVS (fun eo → function
 | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
 | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
Even
```

Adding support for `let rec` bodies

The `letrec` function adds support for `let rec` bodies

```
val letrec : (('a → 'b code) → ('a → 'b code)) →
             (('a → 'b code) → 'c code) →
             'c code
```

Writing `even` with `letrec`:

```
let even = letrec (fun eo → function
   | Even → .< fun x → x  = 0 || .~(eo Odd)  (x-1) >.
   | Odd  → .< fun x → x <> 0 && .~(eo Even) (x-1) >.)
  (fun eo → eo Even)
```

Adding support for `let rec` bodies

The `letrec` function adds support for `let rec` bodies

```
val letrec : (('a → 'b code) → ('a → 'b code)) →
             (('a → 'b code) → 'c code) →
             'c code
```

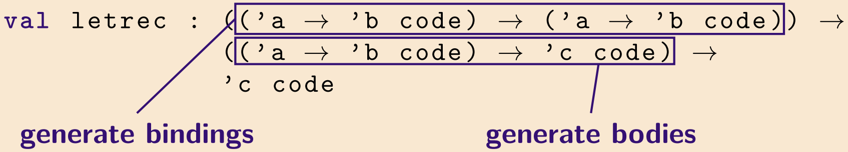
generate bindings

Writing `even` with `letrec`:

```
let even = letrec (fun eo → function
   | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
   | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
  (fun eo → eo Even)
```

Adding support for `let rec` bodies

The `letrec` function adds support for `let rec` bodies



```
val letrec : (('a → 'b code) → ('a → 'b code)) →
             (('a → 'b code) → 'c code) →
             'c code
```

generate bindings

generate bodies

Writing `even` with `letrec`:

```
let even = letrec (fun eo → function
  | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
  | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
 (fun eo → eo Even)
```

Tracing the behaviour of letrec

We'll trace the behaviour of the following call:

```
letrec (fun eo → function
  | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
  | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
 (fun eo → eo Even)
```

Overview:

The call to `letrec` inserts a `let rec` binding group

Each call to `eo` adds a binding to the group

At most one binding is inserted for each index (`Even`, `Odd`)

(Analogy: `letrec` and `eo` correspond to `let_locus` and `genlet`)

Tracing the behaviour of letrec

```
letrec (fun eo → function
  | Even → .<fun x → x  = 0 || .~(eo Odd)  (x-1)>.
  | Odd  → .<fun x → x <> 0 && .~(eo Even) (x-1)>.)
 (fun eo → eo Even)
```

Step 1: `letrec` starts a binding group and invokes the body:

```
.< let rec (* nothing *)
   .in ~(eo Even) >.
```

Step 2: `eo Even` inserts a binding x_e

```
.< let rec xₑ = fun x → x  = 0 || .~(eo Odd)  (x-1)
   .in ~((*eo Even*)) >.
```

Tracing the behaviour of letrec

Step 3: `eo Odd` inserts a second binding x_o:

```
.< let rec x_e = fun x → x  = 0 || .~((*eo Odd*))  (x-1)
       and x_o = fun x → x <> 0 && .~(eo Even) (x-1)
   .in ~((*eo Even*)) >.
```

Step 4: `eo Even` resolves to the existing binding x_e:

```
.< let rec x_e = fun x → x  = 0 || .~((*eo Odd*))  (x-1)
       and x_o = fun x → x <> 0 && x_e (x-1)
   .in ~((*eo Even*)) >.
```

Tracing the behaviour of letrec

Step 5: `eo Odd` completes and resolves to `xₒ`:

```
.< let rec xₑ = fun x → x  = 0 || xₒ (x-1)
       and xₒ = fun x → x <> 0 && xₑ (x-1)
   .in ~((*eo Even*)) >.
```

Step 6: `eo Even` resolves to the binding `xₑ`:

```
.< let rec xₑ = fun x → x  = 0 || xₒ (x-1)
       and xₒ = fun x → x <> 0 && xₑ (x-1)
   .in xₑ >.
```

Generalizing to other forms of recursion

So far: **monomorphic homogeneous mutual recursion**
(every function in a binding group has the same type)

What about **polymorphic** recursion and **heterogeneous** recursion?

Plan: generalize `letrec`
(make the existing function an instance of the generalized version)

Example: heterogeneous recursion for state machines

Add a function `fail` to the binding group:

```
let rec s1 = function
 | 'a' :: k → s2 k
 | _ → fail "no transition"
and s2 = function
 | 'b' :: k → s2 k
 | 'c' :: k → s3 k
 | _ → fail "no transition"
and s3 = function
 | [] → true
 | _ → false
and fail msg = failwith msg
```

Now the binding group contains functions of different types:

```
val s1 : char list → bool
val s2 : char list → bool
val s3 : char list → bool
val fail : string → 'a
```

The generalized `letrec`

With a GADT index, each constructor can have a different type:

```
type _ state =  S1 : (char list → bool) state
              | S2 : (char list → bool) state
              | S3 : (char list → bool) state
              | Fail : (string → 'a) state
```

Now `letrec` needs **higher-kinded & first-class polymorphism**

```
val letrec : ∀(index :: * → *).
   ((∀a.a index → a code) → (∀a.a index → a code)) →
   ((∀a.a index → a code) → 'c code) →
   'c code
```

The generalized `letrec` in OCaml

Use **functors** for higher-kinded polymorphism.
Use **records** for first-class polymorphism

```
module type SYMBOL = sig
 type _ t
 val eql : 'a t → 'b t → ('a, 'b) eql option
end

module Make (Sym:SYMBOL) : sig
 type resolve = { resolve: 'a.'a Sym.t → 'a code }
 type rhs = { rhs: 'a.resolve → 'a Sym.t → 'a code }
 val letrec : rhs → (resolve → 'b code) → 'b code
end
```

Example: state machine using the generalized interface

Generate a `letrec` function from a definition of indexes:

```
module Index = struct
  type 'a t = 'a state
  let eql : type a b. a t → b t → (a, b) eql option =
    fun x y → match x, y with
      | S1, S1 → Some Refl
      ...
end
module L = Make(Index)
```

Invoke `L.letrec` with functions to generate bindings & body:

```
let rhs : type a. L.resolve → a L.sym → a code =
  fun s → function
 | S1 → .<function
          | 'a' :: k → .~(s.resolve S2) k
          | _ → .~(s.resolve Fail) "no transition">.
 ...

L.letrec {rhs} (fun {resolve} → resolve S1)
```

Staged generic programming

(a sketch)

Generic programming recap

Type equality

```
val eqty : {A:TYPEABLE} → {B:TYPEABLE} →
    (A.t, B.t) eq option
```

Generic shallow traversals

```
type 'u genericQ = {D:DATA} → D.t → 'u
val gmapQ : 'u genericQ → 'u list genericQ
```

Generic recursive schemes

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

gshow in action

```
gshow [1;2;3]   ⇝   "(1 :: (2 :: (3 :: ([]))))"
```

Generic programming vs hand-written code

Generic show

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

Hand-written show

```
let rec show_list: ('a → string) → 'a list → string =
 fun f l →
   match l with
   | [] → "[]"
   | h::t → "("^ f h ^" :: "^ show_list f t ^")"
```

Performance difference: an **order of magnitude**

Plan: turn gshow into a **code generator**

Generic programming: binding-time analysis

```
gshow {Data_list{Data_int}} [1; 2; 3]
```

Type representations are **static**   **Values** are **dynamic**.

We've used type representations to traverse values.

Now we'll use type representations to generate code.

Goal: generate code that contains no `Typeable` or `Data` values.

Generic programming, staged

Type equality (unchanged)

```
val eqty : {A:TYPEABLE} → {B:TYPEABLE} →
    (A.t, B.t) eq option
```

Generic shallow traversals (staged)

```
type 'u genericQ = {D:DATA} → D.t code → 'u code
val gmapQ : 'u genericQ → 'u list genericQ
```

Generic recursive schemes (need a fixed point combinator)

```
let gshow = gfixQ_ (fun self {D:DATA} v →
.< "("^ .~(constructor_ v)
       ^ concat " " .~(gmapQ_ self v) ^")" >.)
```

Staging ‘gmapQ‘

The type of staged `gmapQ`

```
type 'u genericQ = {D:DATA} → D.t code → 'u code
val gmapQ : 'u genericQ → 'u list genericQ
```

Implementing staged `gmapQ`

```
implicit module rec DATA_list {A:DATA}
  : DATA with type t = A.t list =
struct
  let gmapQ q l =
  .< match .~l with
    | [] → []
    | h :: t → [.~(q .< h >.); .~(q .< t >.)] >.
  (* ... *)
end
```

Generic schemes and fixed point operators

Need: generate code for each recursive scheme (`gsize`, `gshow`, . . .)

Plan: rewrite schemes using a **fixpoint combinator**

```
let rec gfixQ :
   (u genericQ → u genericQ) → u genericQ =
  fun f {D:DATA} x → f {D} (gfixQ f) x

let gshow = gfixQ (fun self {D:DATA} v →
  "("^ constructor_ v
    ^ concat " " (gmapQ self v) ^ ")")
```

Next step: stage gfixQ using `letrec`

Further improvements

Fully staging *Scrap Your Boilerplate* involves several techniques:

Partially-static data, to simplify algebraic expressions

If insertion, to compute with dynamic values

Branch elimination, where both branches are the same

Inlining, of non-recursive functions

(and more!)

Generated code for ‘gshow‘

A call to `gshow`…

```
gshow {Data_list{Data_bool}}
```

…generates code **specialized to the instance type**:

```
let rec r l = match l with
  | [] → "[]"
  | h::t → if h then "(true :: "^ r t ^")"
              else "(false :: "^ r t ^")"
```

Notes:

non-recursive functions (e.g. `gmapQ` for `bool`) **inlined**
dynamic values (`true`, `false`) **exposed**
static strings **merged**
type representations **eliminated**

Summary

`letrec` generates **mutually recursive functions**

Generalized `letrec` supports **arbitrary recursion**

Careful staging generates **fast idiomatic code**

Staging: **high-level programming + low-level performance**