Chapter 5: λ-Calculus and Laziness

Learning targets of this chapter:

1. Syntax of the simple, untyped λ-calculus
   - symbolic manipulation of λ-expressions in Haskell, substitution
2. Reduction
   - types of reduction, computational progress
   - strategies and normal forms, fixed-point combinator, Church-Rosser theorem
   - lazy/eager evaluation; explicit emulation in the program

λ-Calculus

- Model for computability
  - equivalence with Turing machines
- Formalization of the concept of a function
  - intensional: constructive, via a computational rule
  - symbolically manipulable: composition, application etc.
- Foundation for functional programming languages
- Untyped calculus with minimal syntax
  - just three constructs: variable, application and lambda-abstraction
  - unary functions are sufficient (currying)

Syntax of the Minimal, Untyped λ-Calculus

Let V be a countable set of variables. The language Λ of λ-expressions is defined inductively over the alphabet $V \cup \{(,),\lambda\}$:

(i)	$x \in V$	$\Longrightarrow$	$x \in \Lambda$	variable
(ii)	$M, N \in \Lambda$	$\Longrightarrow$	$(M\,N) \in \Lambda$	application
(iii)	$M \in \Lambda \;\wedge\; x \in V$	$\Longrightarrow$	$(\lambda x M) \in \Lambda$	lambda-abstraction

Shorthand (with "**.**" following the parameters):

$$\lambda x_1 \ldots x_n \,.\, M \quad = (\lambda x_1 \ldots (\lambda x_n M) \ldots)$$

$$M N_1 N_2 \ldots N_n \quad = (\ldots ((M N_1) N_2) \ldots N_n)$$

Λ as Algebraic Data Type

```
data LExp = V String            variable
          | LExp :@: LExp       application
          | L String LExp       lambda-abstraction
          deriving (Show)

fv, bv :: LExp -> [String]

fv (V x)       = [x]                      free variables
fv (f :@: x) = union (fv f) (fv x)
fv (L x e)     = fv e \\ [x]

bv (V _)       = []                       bound variables
bv (f :@: x) = union (bv f) (bv x)
bv (L x e)     = union (bv e) [x]
```

Substitution

- Notation: $E\,[x := A]$
- Semantics: replace in E each free instance of x by A
- Bound names in E, that are free in A, must be renamed!
  (For reasons of simplicity, we rename *all* bound names.)

```
substitute :: String -> LExp -> LExp -> LExp
substitute x a = s where
 s (V w) | x==w = a
         | x/=w = V w
 s (f :@: x)    = s f :@: s x
 s (L w exp)
  | x==w = L w exp
  | x/=w = let fresh=occursNot ([w,x] ++ fv exp ++ bv exp ++ fv a)
           in L fresh (s (substitute w (V fresh) exp))
```

de-Bruijn Indices

Variable renaming is avoidable!

- Replace each instance of a λ-bound variable by $\mathtt{L}\ i$, where i is the nesting depth relative to the corresponding λ-abstraction
- Remove all variables following the λs
- Example: $\lambda x.\lambda y.\lambda f.f\,(\lambda x.x)\,(x+y)$
  in de-Bruijn notation: $\lambda.\lambda.\lambda.(\mathtt{L}\,0)\,(\lambda.\mathtt{L}\,0)\,(\mathtt{L}\,2+\mathtt{L}\,1)$
- Benefit: simple implementation (categorical abstract machine)
- Drawback: difficult manual manipulation

Types of Reduction and Conversion

- β-Reduction: $(\lambda x.E)\ A \rightarrow_\beta E\,[x := A]$
  - formal specification of function application
  - Haskell ex.: `(\x -> x*x) 5` $\rightarrow_\beta$ `5*5`
- δ-Reduction: Ex.: `5*5` $\rightarrow_\delta$ `25`
  - application of a predefined function (not part of the minimal λ-calculus)
- α-Conversion: $y \notin \mathrm{fv}(E) \implies (\lambda x.E) =_\alpha (\lambda y.E[x := y])$
  - renaming of a bound variable (for substitution)
- η-Conversion: $x \notin \mathrm{fv}(E) \implies (\lambda x.E\,x) =_\eta E$
  - switch from pointwise to pointfree form

Redex, Reduction Order, Normal Form

- Redex: reducible expression
- Result not influenced by the reduction order (Leibniz Rule)
  ⇒ local confluence of β-reduction

$$\begin{tikzcd} & \underline{(\lambda x.\underline{(\lambda y.x)N})M} \arrow[dl, "\beta"] \arrow[dr, "\beta"] & \\ \underline{(\lambda x.x)M} \arrow[dr, "\beta"] & & \underline{(\lambda y.M)N} \arrow[dl, "\beta"] \\ & M & \end{tikzcd}$$

- Expression wihout redex: normal form
- Leftmost-outermost redex: start farmost left
- Leftmost-innermost redex: end farmost left

λ-Expressions without Normal Form

1. - $\Omega \stackrel{\text{def}}{=} (\lambda x.(x\,x))\,(\lambda x.(x\,x))$.
   - There is just one redex: the entire expression.
   - $\Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \dots$
2. - Fixed-point combinator Y, applied to variable (f):
   - $Y \stackrel{\text{def}}{=} \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x))$
   - $$\begin{aligned} Y\,f \; &= \; (\lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)))\;f \\ \rightarrow_\beta \; & (\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)) \\ \rightarrow_\beta \; & f\,((\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))) \\ =_\beta \; & f\,(Y\,f) \end{aligned}$$
   - Application: recursion,
     recursion terminates if $Y\,f$ is eliminated by f.

Recursion Example: Factorial

Arithmetic and if can be coded as λ-expressions

$$\mathrm{fac} \stackrel{\mathrm{def}}{=} \lambda\, h\, n.\ \text{if } n=0 \text{ then } 1 \text{ else } n * h\,(n-1)$$

$Y\ \mathrm{fac}\ 3 \twoheadrightarrow_\beta\ \mathrm{fac}\ (Y\ \mathrm{fac})\ 3$ [$\twoheadrightarrow_\beta$: sequence of $\rightarrow_\beta$]

$\rightarrow_\beta\ (\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n * Y\ \mathrm{fac}\ (n-1))\ 3$

$\twoheadrightarrow_\beta\ 3 * Y\ \mathrm{fac}\ 2$

$\rightarrow_\beta\ 3 * \mathrm{fac}\ (Y\ \mathrm{fac})\ 2$

$\rightarrow_\beta\ 3 * (\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n * Y\ \mathrm{fac}\ (n-1))\ 2$

$\twoheadrightarrow_\beta\ 3 * (2 * Y\ \mathrm{fac}\ 1)$

$\rightarrow_\beta\ 3 * (2 * \mathrm{fac}\ (Y\ \mathrm{fac})\ 1)$

$\rightarrow_\beta\ 3 * (2 * (\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n * Y\ \mathrm{fac}\ (n-1))\ 1)$

$\twoheadrightarrow_\beta\ 3 * (2 * (1 * Y\ \mathrm{fac}\ 0))$

$\rightarrow_\beta\ 3 * (2 * (1 * \mathrm{fac}\ (Y\ \mathrm{fac})\ 0))$

$\rightarrow_\beta\ 3 * (2 * (1 * (\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n * Y\ \mathrm{fac}\ (n-1))\ 0))$

$\twoheadrightarrow_\beta\ 3 * (2 * (1 * 1)) \twoheadrightarrow_\delta\ 6$

Church-Rosser Theorem and its Consequences

Definitions:

- $\twoheadrightarrow_\beta$: reflexive transitive closure of $\rightarrow_\beta$
- $=_\beta$: reflexive symmetric transitive closure of $\rightarrow_\beta$

Church-Rosser Theorem (exploiting the local confluence of $\rightarrow_\beta$ (Slide 5–8))

$$\boxed{M_1 =_\beta M_2 \implies \exists P : M_1 \twoheadrightarrow_\beta P \;\wedge\; M_2 \twoheadrightarrow_\beta P}$$

Consequences:

1. If M has a (β-)normal form N, then: $M \twoheadrightarrow_\beta N$
   (existence of a sequence of β-reductions from M to N)
2. A λ-expression has at most one (β-)normal form
   (uniqueness modulo other reduction types, e.g., renaming of bound variables)

Reduction Strategies (1)

Ex.: $\text{fac}\ (Y\ \text{fac})\ 2$

1. Normal-order reduction
   - choice of the leftmost-outermost redex (starts farmost left)
   - $(\ (\ \underline{\text{fac}\ (\ Y\ \text{fac}\ )}\ )\ 2\ )$
     $\rightarrow_\beta ((\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * Y\ \text{fac}\ (n{-}1))\ 2)$

2. Applicative-order reduction
   - Choice of the leftmost-innermost redex (ends farmost left)
   - $(\ (\ \text{fac}\ (\ \underline{Y\ \text{fac}}\ )\ )\ 2\ )$
     $\rightarrow_\beta\ (\text{fac}\ (\text{fac}\ (Y\ \text{fac}))\ 2)$

Reduction Strategies (2)

Theorem: If a λ-expression has a normal form, the normal form is always reached by normal-order reduction.

Note: applicative-order reduction does not have this property.

Ex.:

- normal order: $\underline{(\lambda\, x\, y\, .\, y)\ (\ (\lambda x.(x\, x))\, (\lambda x.(x\, x))\ )} \rightarrow_\beta (\lambda\, y.\, y)$
- applicative order: $(\lambda\, x\, y\, .\, y)\ (\ \underline{(\lambda x.(x\, x))\, (\lambda x.(x\, x))}\ )$
  $\rightarrow_\beta (\lambda\, x\, y\, .\, y)\ (\ (\lambda x.(x\, x))\, (\lambda x.(x\, x))\ )$

Reduction Strategies (3)

Lazy evaluation in Haskell:

- normal-order reduction
- unique copies of common subexpressions (sharing)
- non-strict constructor application (no premature evaluation)

strategy	applicative order	normal order	lazy evaluation
application	strict	non-strict	
parameter evaluation	at reduction	at use	at first use
problems	non-termination	time*, space**	space**
programming languages	ML, OCaml	—	Haskell
parameter passing	call by value	call by name	call by need

*multiple evaluation of terms **unevaluated subterms

Restricted Normal Forms (1)

- Weak normal form (WNF): all redexes are inside λ-abstractions
- Weak-head normal form (WHNF): no redex at the start of the λ-expression

Examples in the minimal λ-calculus:

$(\lambda x.y)$	in normal form, no redex
$(\lambda x.\underline{(\lambda y.y)\ z})$	not in normal form, but in WNF
$x\ \underline{((\lambda y.y)\ z)}$	not in WNF, but in WHNF
$\underline{(\lambda x.x)\ ((\lambda y.y)\ z)}$	not in WHNF

Restricted Normal Forms (2)

- Weak normal form (WNF), weaker than normal form
  - form: all redexes are inside λ-abstractions
  - programming languages: OCaml, ML
  - consequence: no "optimization" of functions at execution time
  - examples in OCaml (`(fun x -> f)` for $(\lambda x.f)$):
    - not in WNF: term with redex not inside a λ-abstraction

      ```
      # let x = (fun y -> 1/y) 0;;
      Exception: Division_by_zero.
      ```

    - in WNF: term with redex only inside a λ-abstraction

      ```
      # let f = (fun x -> (fun y -> 1/y) 0);;
      val f : 'a -> int = <fun>
      # let f x = (fun y -> 1/y) 0;;            with syntactic sugar
      val f : 'a -> int = <fun>
      ```

Restricted Normal Forms (3)

- Weak-head normal form (WHNF), weaker than WNF
  - form: no redex at the start of the entire λ-expression
  - programming language: Haskell
  - consequence: no evaluation of constructor arguments
  - examples in OCaml and Haskell:
    - OCaml (ex. in WHNF, but OCaml evaluates on to WNF)

```
# let xs = [ 1, 2, 3/0 ];;                    redex
Exception: Division_by_zero.
```

    - Haskell (evaluates only to WHNF)

```
Prelude> let xs = [ 1, 2, 3 `div` 0 ]     no redex at start
Prelude> (no error)
Prelude> null xs
Prelude> False        (no error, constr. args not evaluated)
```

Eager Evaluation in Haskell (1)

- f `$!` x: evaluation of x to WHNF before call of function f

```
Prelude> (const "foo") $ (error "bar")
"foo"
Prelude> (const "foo") $! (error "bar")     evaluation
"*** Exception: bar
Prelude> (const "foo") $! [error "bar"]     [.] is in WHNF
"foo"
```

- x `‘seq‘` y: evaluation of x to WHNF, then return of y

```
Prelude> let x = error "bar"  in  x ‘seq‘ 5
*** Exception: bar
Prelude> let x = error "bar"  in  5
5
```

Definition of `$!`: `f $! x = x ‘seq‘ f x`

Eager Evaluation in Haskell (2)

Effect of laziness: memory often full of unfinished work

Strictness annotations for

- reduction of memory consumption
- search with short response time (data structure need not be generated)
- Glasgow Parallel Haskell: parallel processes must start work immediately

Problem with data structures: evaluation to WHNF insufficient

Ways of reaching hyperstrictness:

1. strictness annotations for all functions that generate the structure
2. more elegant: annotate of the arguments of the data constructors

```
data Tree a = Leaf !a | Fork !a !(Tree a) !(Tree a)
```

Laziness in OCaml

Exploitation of WNF (no evaluation inside λ-abstractions)

- Infinite list (producer: `from`, consumer: `take`)

```
type 'a lazylist = Nil | Cons of 'a * (unit -> 'a lazylist)
let rec from n = Cons (n, fun _ -> from (n+1))
let head (Cons (x,_)) = x
let tail (Cons (_,xs)) = xs
let rec take n xs = if n=0 || xs=Nil
                    then []
                    else head xs :: take (n-1) ((tail xs) ())
```

- Use

```
# from 4;;
- : int lazylist = Cons (4, <fun>)
# take 10 (from 4);;
- : int list = [4; 5; 6; 7; 8; 9; 10; 11; 12; 13]
```

Isomorphic New Data Type with `newtype`

As with `data`, a new data type can be defined with `newtype`.
Restriction: exactly one constructor with exactly one argument.

```
newtype KeyValueList a b = KVL [(a,b)]
```

Since there is exactly one constructor, the implementation does not require storage space for the distinction of constructors. Thus, a value of type `KeyValueList a b` requires exactly as much space as a value of type `[(a,b)]`.

Self Application

```
Prelude> let multiple n f = foldl (.) id [f | _ <- [1..n]]
Prelude>
Prelude> let m3 = multiple 3
Prelude> (m3 (m3 (+1))) 0   -- computes  multiple 9 (+1)
9
Prelude> ((m3 m3) (+1)) 0   -- computes  multiple 27 (+1)
27
```

Self application `m3 m3` ?

```
Prelude> :t m3
m3 :: forall a. (a -> a) -> a -> a
m3 :: ((Int->Int)->(Int->Int)) -> (Int->Int) -> (Int->Int)
m3 :: (Int->Int) -> Int -> Int
```

Polymorphism (`forall a`) permits the use of `m3` with multiple types!

Rank-1 Polymorphism

Because of the type

```
Prelude> :t m3
m3 :: forall a. (a -> a) -> a -> a
```

the following works:

```
Prelude> let m3 = multiple 3  in  m3 m3 (+1) 0
27
```

Reason: Because of polymorphism, every caller (of the two functions `m3` in the in-block) is allowed to call the function with its own suitable type.

Haskell 98 has only rank-1 polymorphism, i.e., the `forall`s are in the signature at the very left. With option `-XRankNTypes` in GHC, one can specify polymorphism of any rank. However, there is no type inference for ranks higher than 1.

Rank-2 Polymorphism

If `m3` is bound by a lambda, the call of `m3 m3` does not work:

```
Prelude> let m3 = multiple 3  in  m3 m3 (+1) 0
27                                            however
Prelude> (\ m3 -> m3 m3 (+1) 0) (multiple 3)
<interactive>:1:
  Occurs check: cannot construct the infinite type: t = t -> t1 -> t2
        Expected type: t
        Inferred type: t -> t1 -> t2
    In the first argument of ‘m3’, namely ‘m3’
    In a lambda abstraction: \ m3 -> (m3 m3 (+1)) 0
```

Cause of error: (without explicit signature) lambdas bind monomorphically, i.e., both instances of `m3` on the right side should have the same type but do not.

Specify Rank-2 Polymorphism in the Signature

Solution: use an explicit `forall` to make `m3` polymorphic

```
Prelude> let { m = \ m3 -> m3 m3 (+1) 0  ;
               m :: (forall a. (a -> a) -> a -> a) -> Integer }
Prelude> m (multiple 3)
27
```

Note: The `forall` appears inside the scope of the type for the first argument of `m` ($\leadsto$ rank-2 polymorphism). Thus, `m3` has on the right side of the lambda-expression now type `forall a. (a -> a) -> a -> a)`, i.e., `m3` is polymorphic.