Lecture 25 — Lazy evaluation

- **A small change in the evaluation rules of OCaml makes for a language that is in some ways more powerful. The change makes evaluation "lazy;" lazy, or "delayed," evaluation is the basis of the popular functional language Haskell. It also bring the language closer to the grand-daddy of functional languages, the λ-calculus, known for it extreme simplicity.**
- **We will discuss**
  - **Lazy evaluation**
  - **How to remove features without losing power**

MiniOCaml with lazy evaluation

- **In this lecture, we explore the impact of making one small change in the rules for evaluation via substitution:**

(Const) Const x $\Downarrow_\ell$ Const x

(Fun) Fun(a,e) $\Downarrow_\ell$ Fun(a,e)

(Rec) Rec(f,Fun(a,e)) $\Downarrow_\ell$ Fun(a,e[Rec(f,Fun(a,e))/f]

(δ) $e \; op \; e' \Downarrow_\ell v \; OP \; v'$
$e \Downarrow_\ell v$
$e' \Downarrow_\ell v'$

$\boxed{\begin{array}{l} \text{(App) } e \; e' \Downarrow_\ell v \\ \qquad e \Downarrow_\ell \text{Fun}(a,\, e'') \\ \qquad e''[e'/a] \Downarrow_\ell v \end{array}}$

(If) If(e_1, e_2, e_3) $\Downarrow_\ell v$
$e_1 \Downarrow_\ell$ True
$e_2 \Downarrow_\ell v$

(If) If(e_1, e_2, e_3) $\Downarrow_\ell v$
$e_1 \Downarrow_\ell$ False
$e_3 \Downarrow_\ell v$

(Let) Let(a, e_1, e_2) $\Downarrow_\ell v$
$e_2[e_1/a] \Downarrow_\ell v$

(Note the ℓ subscript.)

Lazy evaluation

- $\Downarrow_\ell$ **is called** *lazy evaluation* **because the evaluation of arguments to functions is delayed until the last moment:**

> (fun x -> x+1) (3+4) $\Downarrow$ 8
>
>   fun x -> x+1 $\Downarrow$ fun x -> x+1
>
>   3+4 $\Downarrow$ 7
>
>   3 $\Downarrow$ 3
>
>   4 $\Downarrow$ 4
>
>   7+1 $\Downarrow$ 8
>
>   7 $\Downarrow$ 7
>
>   1 $\Downarrow$ 1

> (fun x -> x+1) (3+4) $\Downarrow_\ell$ 8
>
>   fun x -> x+1 $\Downarrow_\ell$ fun x -> x+1
>
>   (3+4)+1 $\Downarrow_\ell$ 8
>
>   3+4 $\Downarrow_\ell$ 7
>
>   3 $\Downarrow_\ell$ 3
>
>   4 $\Downarrow_\ell$ 4
>
>   1 $\Downarrow_\ell$ 1

Lazy evaluation (cont.)

- **Since closed expressions have the same value regardless of $when$ they are evaluated, $\Downarrow$ and $\Downarrow_\ell$ almost always produce the same result. But there are exceptions:**

`(fun x -> 3) (4/0)` $\Downarrow$

`(fun x -> 3) (4/0)` $\Downarrow_\ell$

Lazy evaluation (cont.)

- **Another exception:** `let rec f x = f x in (fun x -> 3)(f 0)`.

- **To save writing, let** ϕ = `Rec(f, fun x -> f x)`:

  `let f =` ϕ `in (fun x->3)(f 0)` $\Downarrow$

Lazy evaluation (cont.)

`let f =` ϕ `in (fun x->3)(f 0)` $\Downarrow_\ell$

Removing features

- To show the power of abstractions and applications alone, especially with lazy evaluation, we will begin to remove all features that are just “syntactic sugar.”
- This language — abstraction and application and nothing else, with lazy evaluation — is the language called λ-calculus.
- We will eliminate features in this order:
  - Lists
  - If and boolean values
  - Integers
  - Recursion (which is the most surprising one of all)

Removing features: lists

- **Need to define a representation (`type list =` *something*) and operations:**
  - `nil : list = ...`
  - `cons (h:value) (t:list) : list = ...`
  - `isempty (l:list) : bool = ...`
  - `hd (l:list) : value = ...`
  - `tl (l:list) : list = ...`

  These must behave like lists, e.g. `hd (cons 3 ..)` **=** `3`**.**

- **Here is a representation using only functions (where** `value` **can be any value, including bool — we're assuming dynamic type-checking):**
  - `type list = (value -> list -> value -> value) -> value`
  - `nil : list = fun f -> f 0 0 true`
  - `cons (h:value) (t:list) : list = fun f -> f h t false`
  - `isempty (l:list) : bool = l (fun h t n -> n)`
  - `hd (l:list) : value = l (fun h t n -> h)`
  - `tl (l:list) : list = l (fun h t n -> t)`

Removing features: lists (cont.)

`hd (cons 3 nil)` $\Downarrow$ 3

Lazy lists

- If we are using lazy evaluation, the previous definitions of list operations aren't quite the same as our previous definitions. They correspond to "lazy lists." If we were to make lazy lists a built-in type, we would change the SOS rules like this:
  - Expressions of the form $e\texttt{::}e'$ and `[]` are values. ($\texttt{[}e_1\texttt{;}\ e_2\texttt{;}\ \ldots\texttt{;}\ e_n\texttt{]}$ is just syntactic sugar for $e_1\texttt{::}e_2\texttt{::}\ldots\texttt{::}e_n\texttt{::[]}$.)
  - Rules for list operations:

(Cons) $e\texttt{::}e' \Downarrow_\ell e\texttt{::}e'$

(Nil) $\texttt{[]} \Downarrow_\ell \texttt{[]}$

(Head) $\texttt{hd}\ e \Downarrow_\ell v$

$$e \Downarrow_\ell e' :: e''$$
$$e' \Downarrow_\ell v$$

(Tail) $\texttt{tl}\ e \Downarrow_\ell v$

$$e \Downarrow_\ell e' :: e''$$
$$e'' \Downarrow_\ell v$$

- (Note: the same definitions would give the ordinary, "strict" list operations if the underlying language were not lazy.)

Lazy lists (cont.)

- **Lazy lists are really useful. For example, they allow us to build infinite lists:**

```
let rec ints = fun i -> i :: ints (i+1) in hd (tl (tl (ints 0)))
```

- **Infinite lists are not just a curiosity. They allow some computations to be written in a more modular way.**

Lazy lists (cont.)

- **E.g Newton's method:**

  To find sqrt(x), generate sequence: $< a_i >$, where a_0 is arbitrary, and $a_{i+1} = \frac{a_i + x/a_i}{2}$. Then choose first a_i s.t. $|a_i - a_{i-1}| < \epsilon$.

- **This can be programmed elegantly by creating the *infinite* list $< a_i >$, and then iterating over it:**

```
let next x a = (a+x/a)/2
let rec repeat f a = a :: repeat f (f a)
let candidates x = repeat (next x) (x/2);;  (* list of candidates *
let find test (a1::a2::as) = if test a1 a2 then a2
               else find test (a2::as)
let withineps eps = fun a b -> (abs (a-b)) < eps
let sqrt x eps = find (withineps eps) candidates
```

Removing features: if, and boolean values

- **With lazy evaluation, can define if. This is impossible in an eager evaluation language like OCaml because all functions defined using `fun` are strict.**

```
let true = fun x y -> x
let false = fun x y -> y
let if a b c = a b c
let lessthan i j = if i<j then true else false
let and b b' = if b then b' else true     (or just, b b' true)
```

`if (lessthan 3 2) 5 10` $\Downarrow_\ell$ 10

Where we are...

- **We have shown that `lists`, and `if`, as well as boolean values, are syntactic sugar.**
  - **This eliminates the need for boolean constants and lists, and the δ rules for them.**
  - **Character can be regarded as integers, and Strings as lists of characters, so we will eliminate both as syntactic sugar.**
- **We already know that `let` is syntactic sugar.**
- **We are left with integers and integer operations, and recursion.**

Removing features: integers

- **Alonzo Church invented λ-calculus. On the way to proving that it is Turing-complete, he produced a representation of integers in terms of functions, which has been given the name "Church numerals."**

- **As usual, we need a *representation* of integers (`type intgr = `*`something`*), and then definitions of constants and functions:**

  - `let zero : intgr = ...`
  - `let one : intgr = ...`
  - `let plus (i1:intgr) (i2:intgr) : intgr = ...`
  - `let lessthan (i1:intgr) (i2:intgr) : bool = ...`
  - **... and so on**

Church numerals

- **These have to act in "integer-like" ways, e.g.** `lessthan i (plus i one)` **must be true,** `equals (mult three four) (mult two six)` **must be true, etc.**
- **Church numerals represent numbers by lambda terms:**

```
type intgr = ('a -> 'a) -> ('a -> 'a)
zero  = fun f -> fun x -> x
one   = fun f -> fun x -> f x
two   = fun f -> fun x -> f (f x)
three = fun f -> fun x -> f (f (f x))
```

Church numerals (cont.)

- **Plus and times are easy to define:**

```
let plus m n = fun f -> fun x -> (m f (n f x)
let mult m n = fun f -> fun x -> (m (n f)) x
```

Removing features: `let rec`

- **Implementing recursion without `let rec` is the trickiest part. It uses the so-called "paradoxical combinator," a.k.a. the "Y combinator":**

```
let W = fun F -> fun f -> F (f f)
let Y = fun F -> (W F)(W F)
```

- **For recursive function `let rec f = fun x -> ...`, instead write `let f = Y (fun f -> fun x ->...)`.**

```
let fac = Y (fun fac -> fun x -> if x=0 then 1 else x * fac (x-1))
```

- **We leave it as an exercise to show that, e.g., `fac 3` evaluates to 6. Note that there is no explicit recursion used here: each name defined (`W`, `Y`, and `fac`) refers only to names defined previously, never to themselves.**

λ-calculus

- **The λ-calculus is a language with only three types of expressions:**

$$\textit{expr} = \textit{variable} \mid \textit{expr}\ \textit{expr} \mid \texttt{fun}\ \textit{variable} \rightarrow \textit{expr}$$

and two evaluation rules:

(Fun) $\texttt{fun}\ a \rightarrow e \Downarrow_\ell \texttt{fun}\ a \rightarrow e$

(App) $e\ e' \Downarrow_\ell v$

$$e \Downarrow_\ell \texttt{fun}\ a \rightarrow e''$$
$$e''[e'/a] \Downarrow_\ell v$$

- **Lacking only features that are syntactic sugar, and the ability to display values in a "natural" form, λ-calculus is as powerful as (dynamically-typed, lazy) OCaml.**

Aside: β-reduction

- **Here is an even simpler semantics for λ-calculus:**
  - **Given an expression, apply the following transformation wherever it occurs:**
    (β-reduction) $\;(\texttt{fun}\ x\ \texttt{->}\ e)\ e' \ \rightarrow_\beta\ e[e'/x]$
  - **Applying β may produce new places where β can be applied. If *it is possible* to reduce the term to a value by applying β-reduction repeatedly, then that is the value of the term.**
- **There may be many ways to apply β, which could result in different values; some might never terminate. The computation rule says if *any* sequence of β's results in a value, that is the value of the term. Furthermore, a famous theorem (Church-Rosser) says that all values of a term obtained this way are in a certain sense equivalent.**

Haskell

- **OCaml uses non-lazy, or "eager," evaluation ($\Downarrow$)**
- **Haskell is a popular functional language that uses lazy evaluation ($\Downarrow_\ell$).**
- **Haskell is statically-typed and has a syntax somewhat like OCaml, including pattern-matching.**
- **As an example, this remarkable definition of the list of Fibonacci numbers works if you just follow our rules for lazy lists, where `map` and `zip` have exactly the same definitions as in OCaml:**

```
fib = 1 : 2 : map (+) (zip fib (tail fib))
```

Implementation of lazy languages

- **Using the substitution model, you need to change just the rules shown (App and Let).**
- **Real implementations are based on the environment model. SOS rules for lazy evaluation in the environment model are a little tricky, for this reason: If we want to evaluate arguments *later*, then we need to remember the environment in which they were *supposed* to be evaluated.**
- **Solution: Arguments are stored in the environment as closure-like things called *thunks* — pairs containing the expression and the environment.**
- **Evaluating a variable x: x is bound to a thunk $< e, \rho >$; evaluate e in ρ, and replace the binding of x by this value.**

Wrap-up

- Today we discussed lazy evaluation, and showed how higher-order functions with lazy evaluation provides tremendous power. The λ-calculus, a functional language simplified to the bare bones, is as powerful, ignoring syntactic sugar, as OCaml. Haskell is a popular statically-typed functional language that uses lazy evaluation.
- We discussed this both to introduce you to Haskell, and to demonstrate the real power of higher-order functions.
- What to do now:
  - Um, nothing much. (If you're ambitious, make the small change in your MP9 solution and try these examples!)