Functional Programming

by Łukasz Stafiniak

Email: `lukstafi@gmail.com`, `lukstafi@ii.uni.wroc.pl`
Web: `www.ii.uni.wroc.pl/~lukstafi`

Lecture 3: Computation

"Using, Understanding and Unraveling the OCaml Language" Didier Rémy, chapter 1

"The OCaml system" manual, the tutorial part, chapter 1

Function Composition

- The usual way function composition is defined in math is "backward":
  - math: $(f \circ g)(x) = f(g(x))$
  - OCaml: `let (-|) f g x = f (g x)`
  - F#: `let (<<) f g x = f (g x)`
  - Haskell: `(.) f g = \x -> f (g x)`
- It looks like function application, but needs less parentheses. Do you recall the functions `iso1` and `iso2` from previous lecture?

  ```
  let iso2 = step1l -| step2l -| step3l
  ```

- A more natural definition of function composition is "forward":
    - OCaml: `let (|-) f g x = g (f x)`
    - F#: `let (>>) f g x = g (f x)`
- It follows the order in which computation proceeds.

  ```
  let iso1 = step1r |- step2r |- step3r
  ```

- *Partial application* is e.g. `((+) 1)` from last week: we don't pass all arguments a function needs, in result we get a function that requires the remaining arguments. How is it used above?

- Now we define $f^n(x) := (f \circ \ldots \circ f)(x)$ (f appears n times).

  ```
  let rec power f n =
    if n <= 0 then (fun x -> x) else f -| power f (n-1)
  ```

- Now we define a numerical derivative:

  ```
  let derivative dx f = fun x -> (f(x +. dx) -. f(x)) /. dx
  ```

  where the intent to use with two arguments is stressed, or for short:

  ```
  let derivative dx f x = (f(x +. dx) -. f(x)) /. dx
  ```

- We have `(+): int -> int -> int`, so cannot use with `floating` point numbers – operators followed by dot work on `float` numbers.

  ```
  let pi = 4.0 *. atan 1.0
  let sin''' = (power (derivative 1e-5) 3) sin;;
  sin''' pi;;
  ```

Evaluation Rules (reduction semantics)

- Programs consist of **expressions**:

$a :=$	x	variables
$\mid$	$\texttt{fun}\ x\texttt{->}a$	(defined) functions
$\mid$	$a\,a$	applications
$\mid$	C^0	value constructors of arity 0
$\mid$	$C^n(a, \ldots, a)$	value constructors of arity n
$\mid$	f^n	built-in values (primitives) of a. n
$\mid$	$\texttt{let}\ x = a\ \texttt{in}\ a$	name bindings (local definitions)
$\mid$	$\texttt{match}\ a\ \texttt{with}$	
	$p\texttt{->}a \mid \ldots \mid p\texttt{->}a$	pattern matching
$p :=$	x	pattern variables
$\mid$	$(p, \ldots, p)$	tuple patterns
$\mid$	C^0	variant patterns of arity 0
$\mid$	$C^n(p, \ldots, p)$	variant patterns of arity n

- *Arity* means how many arguments something requires; (and for tuples, the length of a tuple).
- To simplify presentation, we will use a primitive `fix` to define a limited form of `let rec`:

$$\texttt{let rec } f\ x = e_1 \texttt{ in } e_2 \equiv \texttt{let } f = \texttt{fix (fun } f\ x\texttt{->}e_1\texttt{) in } e_2$$

- Expressions evaluate (i.e. compute) to **values**:

$$\begin{array}{rlrl} v := & \texttt{fun } x\texttt{->}a & & \text{(defined) functions} \\ \mid & C^n(v_1, \ldots, v_n) & & \text{constructed values} \\ \mid & f^n\, v_1 \ldots v_k & k < n & \text{partially applied primitives} \end{array}$$

- To *substitute* a value v for a variable x in expression a we write $a[x := v]$ – it behaves as if every occurrence of x in a was *rewritten* by v.
  - (But actually the value v is not duplicated.)

- Reduction (i.e. computation) proceeds as follows: first we give *redexes*

$$\begin{array}{rcl}
(\texttt{fun } x\texttt{->}a)\, v & \rightsquigarrow & a[x := v]\\
\texttt{let } x = v \texttt{ in } a & \rightsquigarrow & a[x := v]\\
f^n\, v_1 \ldots v_n & \rightsquigarrow & f(v_1, \ldots, v_n)\\
\texttt{match } v \texttt{ with } x\texttt{->}a \texttt{ | } \ldots & \rightsquigarrow & a[x := v]\\
\texttt{match } C_1^n(v_1, \ldots, v_n) \texttt{ with} & & \\
C_2^n(p_1, \ldots, p_k)\texttt{->}a \texttt{ | } \mathrm{pm} & \rightsquigarrow & \texttt{match } C_1^n(v_1, \ldots, v_n)\\
 & & \qquad \texttt{with } \mathrm{pm}\\
\texttt{match } C_1^n(v_1, \ldots, v_n) \texttt{ with} & & \\
C_1^n(x_1, \ldots, x_n)\texttt{->}a \texttt{ | } \ldots & \rightsquigarrow & a[x_1 := v_1; \ldots; x_n := v_n]
\end{array}$$

  If $n = 0$, $C_1^n(v_1, \ldots, v_n)$ stands for C_1^0, etc. By $f(v_1, \ldots, v_n)$ we denote the actual value resulting from computing the primitive. We omit the more complex cases of pattern matching.

- Rule variables: x matches any expression/pattern variable; $a, a_1, \ldots, a_n$ match any expression; $v, v_1, \ldots, v_n$ match any value. Substitute them so that the left-hand-side of a rule is your expression, then the right-hand-side is the reduced expression.

- The remaining rules evaluate the arguments in arbitrary order, but keep the order in which `let`…`in` and `match`…`with` is evaluated.

  If $a_i \rightsquigarrow a_i'$, then:

$$\begin{array}{rcl} a_1\, a_2 & \rightsquigarrow & a_1'\, a_2 \\ a_1\, a_2 & \rightsquigarrow & a_1\, a_2' \\ C^n(a_1, \ldots, a_i, \ldots, a_n) & \rightsquigarrow & C^n(a_1, \ldots, a_i', \ldots, a_n) \\ \texttt{let}\ x = a_1\ \texttt{in}\ a_2 & \rightsquigarrow & \texttt{let}\ x = a_1'\ \texttt{in}\ a_2 \\ \texttt{match}\ a_1\ \texttt{with}\, \mathrm{pm} & \rightsquigarrow & \texttt{match}\ a_1'\ \texttt{with}\, \mathrm{pm} \end{array}$$

- Finally, we give the rule for the primitive `fix` – it is a binary primitive:

$$\texttt{fix}^2\, v_1\, v_2 \ \rightsquigarrow\ v_1\, (\texttt{fix}^2\, v_1)\, v_2$$

  Because `fix` is binary, $(\texttt{fix}^2\, v_1)$ is already a value so it will not be further computed until it is applied inside of v_1.

- Compute some programs using the rules by hand.

Symbolic Derivation Example

Go through the examples from the `Lec3.ml` file in the toplevel.

```
eval_1_2 <-- 3.00 * x + 2.00 * y + x * x * y
  eval_1_2 <-- x * x * y
    eval_1_2 <-- y
    eval_1_2 --> 2.
    eval_1_2 <-- x * x
      eval_1_2 <-- x
      eval_1_2 --> 1.
      eval_1_2 <-- x
      eval_1_2 --> 1.
    eval_1_2 --> 1.
  eval_1_2 --> 2.
  eval_1_2 <-- 3.00 * x + 2.00 * y
    eval_1_2 <-- 2.00 * y
      eval_1_2 <-- y
      eval_1_2 --> 2.
      eval_1_2 <-- 2.00
      eval_1_2 --> 2.
    eval_1_2 --> 4.
    eval_1_2 <-- 3.00 * x
      eval_1_2 <-- x
      eval_1_2 --> 1.
      eval_1_2 <-- 3.00
      eval_1_2 --> 3.
    eval_1_2 --> 3.
  eval_1_2 --> 7.
eval_1_2 --> 9.
- : float = 9.
```

Tail Calls (and tail recursion)

- Excuse me for not defining what a *function call* is...
- Computers normally evaluate programs by creating *stack frames* on the stack for function calls (roughly like indentation levels in the above example).
- A **tail call** is a function call that is performed last when computing a function.
- Functional language compilers will often insert a "jump" for a tail call instead of creating a stack frame.
- A function is **tail recursive** if it calls itself, or functions it mutually-recurively depends on, only using a tail call.
- Tail recursive functions often have special *accumulator* arguments that store intermediate computation results which in a non-tail-recursive function would just be values of subexpressions.
- The accumulated result is computed in "reverse order" – while climbing up the recursion rather than while descending (i.e. returning) from it.

- The issue is not relevant for *lazy* programming languages like Haskell.
- Compare:

```
# let rec unfold n = if n <= 0 then [] else n :: unfold (n-1);;
val unfold : int -> int list = <fun>
# unfold 100000;;
- : int list =
[100000; 99999; 99998; 99997; 99996; 99995; 99994; 99993; ...]
# unfold 1000000;;
Stack overflow during evaluation (looping recursion?).
# let rec unfold_tcall acc n =
  if n <= 0 then acc else unfold_tcall (n::acc) (n-1);;
  val unfold_tcall : int list -> int -> int list = <fun>
# unfold_tcall [] 100000;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; ...]
# unfold_tcall [] 1000000;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; ...]
```

- Is it possible to find the depth of a tree using a tail-recursive function?

First Encounter of Continuation Passing Style

We can postpone doing the actual work till the last moment:

```
let rec depth tree k = match tree with
    | Tip -> k 0
    | Node(_,left,right) ->
      depth left (fun dleft ->
        depth right (fun dright ->
          k (1 + (max dleft dright))))

let depth tree = depth tree (fun d -> d)
```

Homework

By “traverse a tree” below we mean: write a function that takes a tree and returns a list of values in the nodes of the tree.

1. Write a function (of type `btree -> int list`) that traverses a binary tree: in prefix order – first the value stored in a node, then values in all nodes to the left, then values in all nodes to the right;
2. in infix order – first values in all nodes to the left, then value stored in a node, then values in all nodes to the right (so it is “left-to-right” order);
3. in breadth-first order – first values in more shallow nodes.
4. Turn the function from ex. 1 or 2 into continuation passing style.

5. Do the homework from the end of last week slides: write `btree_deriv_at`.

6. Write a function `simplify: expression -> expression` that simplifies the expression a bit, so that for example the result of `simplify (deriv exp dv)` looks more like what a human would get computing the derivative of `exp` with respect to `dv`.

   - Write a `simplify_once` function that performs a single step of the simplification, and wrap it using a general `fixpoint` function that performs an operation until a *fixed point* is reached: given f and x, it computes $f^n(x)$ such that $f^n(x) = f^{n+1}(x)$.