Functional Programming

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

Lecture 1: Logic

From logic rules to programming constructs

In the Beginning there was Logos

What logical connectives do you know?

$\top$	$\bot$	$\wedge$	$\vee$	$\rightarrow$
		$a \wedge b$	$a \vee b$	$a \rightarrow b$
truth	falsehood	conjunction	disjunction	implication
“trivial”	“impossible”	a and b	a or b	a gives b
	shouldn’t get	got both	got at least one	given a, we get b

How can we define them?

Think in terms of derivation trees:

$$\dfrac{\dfrac{\dfrac{}{\text{a premise}} \quad \dfrac{}{\text{another premise}}}{\text{some fact}} \qquad \dfrac{\dfrac{}{\text{this we have by default}}}{\text{another fact}}}{\text{final conclusion}}$$

Define by providing rules for using the connectives: for example, a rule $\frac{a \quad b}{c}$ matches parts of the tree that have two premises, represented by variables a and b, and have any conclusion, represented by variable c.

Try to use only the connective you define in its definition.

Rules for Logical Connectives

Introduction rules say how to produce a connective.
Elimination rules say how to use it.
Text in parentheses is comments. Letters are variables: stand for anything.

	Introduction Rules	Elimination Rules
$\top$	$\dfrac{}{\top}$	doesn't have
$\bot$	doesn't have	$\dfrac{\bot}{a}$ (i.e., anything)
$\wedge$	$\dfrac{a \quad b}{a \wedge b}$	$\dfrac{a \wedge b}{a}$ (take first) $\quad \dfrac{a \wedge b}{b}$ (take second)
$\vee$	$\dfrac{a}{a \vee b}$ (put first) $\quad \dfrac{b}{a \vee b}$ (put second)	$\dfrac{a \vee b \quad \begin{array}{c} \overline{a}^{x} \text{ (consider } a\text{)} \\ \Big\vert \\ c \end{array} \quad \begin{array}{c} \overline{b}^{y} \text{ (consider } b\text{)} \\ \Big\vert \\ c \end{array}}{c \text{ (since in both cases we get it)}}$ using x, y
$\rightarrow$	$\dfrac{\begin{array}{c} \overline{a}^{x} \\ \Big\vert \\ b \end{array}}{a \rightarrow b}$ using x	$\dfrac{a \rightarrow b \quad a}{b}$

Notations

$$\begin{array}{c}\overline{a}^{x},\\ | \\ b\end{array} \quad \text{or} \quad \begin{array}{c}\overline{a}^{x}\\ | \\ c\end{array}$$

match any subtree that derives b (or c) and can use a (by assumption $\overline{a}^{x}$) although otherwise a might not be warranted. For example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\text{sunny}}^{x}}{\text{go outdoor}}}{\text{playing}}}{\text{happy}}}{\text{sunny}\to\text{happy}}\ \text{using } x$$

Such assumption can only be used in the matched subtree! But it can be used several times, e.g. if someone's mood is more difficult to influence:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\text{sunny}}^{x}}{\text{go outdoor}}}{\text{playing}} \qquad \dfrac{\overline{\text{sunny}}^{x} \quad \dfrac{\overline{\text{sunny}}^{x}}{\text{go outdoor}}}{\text{nice view}}}{\text{happy}}}{\text{sunny}\to\text{happy}}\ \text{using } x$$

Elimination rule for disjunction represents **reasoning by cases**.
How can we use the fact that it is sunny$\vee$cloudy (but not rainy)?

$$\dfrac{\dfrac{}{\text{sunny}\vee\text{cloudy}}\ \text{forecast}\qquad \dfrac{\overline{\text{sunny}}^{\,x}}{\text{no-umbrella}}\qquad \dfrac{\overline{\text{cloudy}}^{\,y}}{\text{no-umbrella}}}{\text{no-umbrella}}\ \text{using } x, y$$

We know that it will be sunny or cloudy, by watching weather forecast. If it will be sunny, we won't need an umbrella. If it will be cloudy, we won't need an umbrella. Therefore, won't need an umbrella.

We need one more kind of rules to do serious math: **reasoning by induction** (it is somewhat similar to reasoning by cases). Example rule for induction on natural numbers:

$$\dfrac{p(0) \quad \begin{array}{c}\overline{p(x)}^{\,x}\\ \Big| \\ p(x+1)\end{array}}{p(n)}\ \text{by induction, using } x$$

So we get any p for any natural number n, provided we can get it for 0, and using it for x we can derive it for the successor $x + 1$, where x is a unique variable (we cannot substitute for it some particular number, because we write “using x” on the side).

Logos was Programmed in OCaml

Logic	Type	Expr.	Introduction Rules	Elimination Rules
$\top$	`unit`	`()`	$\dfrac{}{\texttt{()}\colon \texttt{unit}}$	
$\bot$	`'a`	`raise`		$\dfrac{\text{oops!}}{\texttt{raise Not\_found}\colon c}$
$\wedge$	`*`	`(,)`	$\dfrac{s\colon a \quad t\colon b}{s\texttt{,}t\colon a\texttt{*}b}$	$\dfrac{p\colon a\texttt{*}b}{\texttt{fst}\ p\colon a} \quad \dfrac{p\colon a\texttt{*}b}{\texttt{snd}\ p\colon b}$
$\vee$	`|`	`match`	$\dfrac{s\colon a}{A(s)\colon A\ \texttt{of}\ a \mid B\ \texttt{of}\ b}$ (need to name sides) $\dfrac{t\colon b}{B(t)\colon A\ \texttt{of}\ a \mid B\ \texttt{of}\ b}$	$\dfrac{t\colon A\ \texttt{of}\ a \mid B\ \texttt{of}\ b \quad \begin{array}{c}\overline{x\colon a}^{\,x}\\ \vert \\ e_1\colon c\end{array} \quad \begin{array}{c}\overline{y\colon b}^{\,y}\\ \vert \\ e_2\colon c\end{array}}{\texttt{match}\ t\ \texttt{with}\ A(x)\texttt{->}e_1 \mid B(y)\texttt{->}e_2\colon c}$ variables x, y
$\rightarrow$	`->`	`fun`	$\dfrac{\begin{array}{c}\overline{x\colon a}^{\,x}\\ \vert \\ e\colon b\end{array}}{\texttt{fun}\ x\texttt{->}e\colon a \rightarrow b}$ var x	$\dfrac{f\colon a \rightarrow b \quad t\colon a}{f t\colon b}$ (application)
induction		`rec`	$\dfrac{\begin{array}{c}\overline{x\colon a}^{\,x}\\ \vert \\ e\colon a\end{array}}{\texttt{rec}\ x\texttt{=}e\colon a}$ variable x	

- In the judgements above, $e\colon a$ means that expression e has type a.
  - Expressions describe computations, types say what the computed value will be, generally speaking.
- Variables are like in mathematics, it would be better to call them **names**.
- `Not_found` is just an example of an exception (any can be raised).
- Expressions $(s$`,`$t)$, $(r$`,`s`,`$t)$, ... are called tuples and types a`*`b, a`*`b`*`c, ... are called tuple types. For example we can have pair `(7,"Mary")` of pair type `int*string`. Parentheses are not necessary.
- Types A `of` a `|` B `of` b, A `of` a `|` B `of` b `|` C `of` c, ... are called variant types (or discriminated unions). The variants need to have labels, like A and B above, because we need to recognize which case we deal with!

Definitions

Writing out expressions and types repetitively is tedious: we need definitions. **Definitions for types** are written: `type ty =` some type.

- Writing $A(s)$: A `of` a`|`B `of` b in the table was cheating. Usually we have to define the type and then use it, e.g. using `int` for a and `string` for b:
  ```
  type int_string_choice = A of int | B of string
  ```
  allows us to write $A(s)$: `int_string_choice`.
- Without the type definition, it is difficult to know what other variants there are when one infers (i.e. "guesses", computes) the type!
- In OCaml we can write \`$A(s)$: [\`A `of` a`|`\`$B$ `of` $b$]. With "\`" variants, OCaml does guess what other variants are. These types are fun, but we will not use them in future lectures.

- Tuple elements don't need labels because we always know at which position a tuple element stands. But having labels makes code more clear, so we can define a record type:

  ```
  type int_string_record = {a: int; b: string}
  ```

  and create its values: `{a = 7; b = "Mary"}`.
- We access the fields of records using the dot notation:
  `{a=7; b="Mary"}.b = "Mary"`.

Recursive expression `rec` x`=`e in the table was cheating: `rec` (usually called `fix`) cannot appear alone in OCaml! It must be part of a definition.

Definitions for expressions are introduced by rules a bit more complex than these:

$$\dfrac{e_1 \colon a \quad \begin{matrix}\overline{x \colon a}^{\;x} \\ \vert \\ e_2 \colon b\end{matrix}}{\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2 \colon b}$$

(note that this rule is the same as introducing and eliminating $\rightarrow$), and:

$$\dfrac{\begin{matrix}\overline{x \colon a}^{\;x} \\ \vert \\ e_1 \colon a\end{matrix} \quad \begin{matrix}\overline{x \colon a}^{\;x} \\ \vert \\ e_2 \colon b\end{matrix}}{\texttt{let rec } x\texttt{=}e_1 \texttt{ in } e_2 \colon b}$$

We will cover what is missing in above rules when we will talk about **polymorphism.**

- Type definitions we have seen above are global: they need to be at the top-level, not nested in expressions, and they extend from the point they occur till the end of the source file or interactive session.
- `let-in` definitions for expressions: `let` x=e_1 `in` e_2 are local, x is only visible in e_2. But `let` definitions are global: placing `let` x=e_1 at the top-level makes x visible from after e_1 till the end of the source file or interactive session.
- In the interactive session, we mark an end of a top-level "sentence" by `;;` – it is unnecessary in source files.
- Operators like `+`, `*`, `<`, `=`, are names of functions. Just like other names, you can use operator names for your own functions:

```
let (+:) a b = String.concat "" [a; b];;     Special way of defining
"Alpha" +: "Beta";;                    but normal way of using operators.
```

- Operators in OCaml are **not overloaded**. It means, that every type needs its own set of operators. For example, `+`, `*`, `/` work for intigers, while `+.`, `*.`, `/.` work for floating point numbers. **Exception:** comparisons `<`, `=`, etc. work for all values other than functions.

Exercises

Exercises from Think OCaml. How to Think Like a Computer Scientist by Nicholas Monje and Allen Downey.

1. Assume that we execute the following assignment statements:
   ```
   let width = 17;;
   let height = 12.0;;
   let delimiter = '.';;
   ```
   For each of the following expressions, write the value of the expression and the type (of the value of the expression), or the resulting type error.

   a. `width/2`

   b. `width/.2.0`

   c. `height/3`

   d. `1 + 2 * 5`

   e. `delimiter * 5`

2. Practice using the OCaml interpreter as a calculator:

   a. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5?
   Hint: 392.6 is wrong!

   b. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?

   c. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

3. You've probably heard of the fibonacci numbers before, but in case you haven't, they're defined by the following recursive relationship:

$$\begin{cases} f(0) & = 0 \\ f(1) & = 1 \\ f(n+1) & = f(n) + f(n-1) \quad \text{for } n = 2, 3, \ldots \end{cases}$$

Write a recursive function to calculate these numbers.

4. A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

   The following are functions that take a string argument and return the first, last, and middle letters:

```
let first_char word = word.[0];;
let last_char word =
  let len = String.length word - 1 in
  word.[len];;
let middle word =
  let len = String.length word - 2 in
  String.sub word 1 len;;
```

   a. Enter these functions into the toplevel and test them out. What happens if you call middle with a string with two letters? One letter? What about the empty string, which is written `""`?

   b. Write a function called is_palindrome that takes a string argument and returns `true` if it is a palindrome and `false` otherwise.

5. The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.
   One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b, then gcd $(a, b)=$ gcd (b, r). As a base case, we can consider $\gcd(a, 0) = a$.
   Write a function called gcd that takes parameters a and b and returns their greatest common divisor.
   If you need help, see http://en.wikipedia.org/wiki/Euclidean_algorithm.