Functional Programming
Exercise 1: Functions, Values, and Types

Shin-Cheng Mu

2012 Formosan Summer School on Logic, Language, and Computation
Aug 27 – Sep 7, 2012

Functions

1. Define a function that computes the area of a circle with given radius r (you may use 22/7 as an approximation to π).

> **Solution:** $\mathbf{let}\ \mathit{area}\ r\ =\ 22.0/.7.0 *.r *.r$

2. Recall the definition of *curry*:

$$\#\ \mathbf{let}\ \mathit{curry}\ f\ x\ y\ =\ f\ (x, y)$$
$$\mathbf{val}\ \mathit{curry}\ :\ ('a * 'b \to 'c) \to 'a \to 'b \to 'c\ =\ \langle \mathit{fun} \rangle$$

Define $\mathit{uncurry} : ('a \to 'b \to 'c) \to ('a * 'b \to c)$. Prove that

$$\mathit{curry}\ (\mathit{uncurry}\ f) = f$$
$$\mathit{uncurry}\ (\mathit{curry}\ f) = f$$

> **Solution:** Given the type, there is only one possibility:
>
> $$\mathbf{let}\ \mathit{uncurry}\ f\ (x, y) = f\ x\ y$$
>
> By extensional equality of functions, $\mathit{curry}\ (\mathit{uncurry}\ f) = f$ is equivalent to that $\mathit{curry}\ (\mathit{uncurry}\ f)\ x\ y = f\ x\ y$ for all x and y. We reason:
>
> $$\begin{array}{ll} & \mathit{curry}\ (\mathit{uncurry}\ f)\ x\ y \\ = & \{\ \text{definition of } \mathit{curry}\ \} \\ & (\mathit{uncurry}\ f)\ (x, y) \\ = & \{\ \text{definition of } \mathit{uncurry}\ \} \\ & f\ x\ y \end{array}$$
>
> Similarly, $\mathit{uncurry}\ (\mathit{curry}\ f) = f$ is equivalent to that $\mathit{uncurry}\ (\mathit{curry}\ f)\ z = f\ z$ for all $z : (a * b)$. All such z must be of the form (x, y) for some $x : 'a$ and $y : 'b$ (since we are not considering $\perp$). We

reason:

$$
\begin{array}{cl}
 & \mathit{uncurry}\ (\mathit{curry}\ f)\ (x, y) \\
= & \{\ \text{definition of } \mathit{uncurry}\ \} \\
 & (\mathit{curry}\ f)\ x\ y \\
= & \{\ \text{definition of } \mathit{curry}\ \} \\
 & f\ (x, y)
\end{array}
$$

Playing Around With Lists

The purpose of this exercise is to familiarise one with list processing and the "combinator" style of programming, in which programs are composed from smaller parts.

1. The list is traditionally an important datatype in functional languages. Start the OCaml interpreter, type in the following expression:

   $\#\ [1; 2; 3; 4];;$
   $-\ :\ \mathit{int\ list}\ =\ [1; 2; 3; 4]$

   OCaml says that $[1; 2; 3; 4]$ is has type *int list* — a list whose elements are integers. In OCaml, all elements in a list must be of the same type.

   Guess the type of the following lists, before finding out the answer in OCaml.

   - $[\mathit{true}; \mathit{false}; \mathit{true}]$.
   - $[[1; 2]; [\,]; [6; 7]]$.
   - $[(+); (-); (/)]$.
   - $[\,]$.
   - $[[\,]]$.

   Solution: The types:

   - *bool list*
   - *int list list*
   - $(\mathit{int} \to \mathit{int} \to \mathit{int})\ \mathit{list}$
   - $'a\ \mathit{list}$
   - $'a\ \mathit{list\ list}$

2. **Take**. Download the file `"Utils.ml"` from the course website and save it in your current working directory. Load the file by issueing the command

   \# *use* `"Utils.ml"`

   We have defined some functions that might be useful later.

   Try the following expressions:

   - *take* $3\ [0; 1; 2; 3; 4]$

- *take* 0 [0; 1; 2; 3; 4]
- *take* 4 [0; 1; 2]

Describe in words what the function *take* does.

Solution: *take* n xs yields the longest prefix of xs whose length is at most n.

3. **Drop**. Try the following expressions:
   - *drop* 3 [0; 1; 2; 3; 4]
   - *drop* 0 [0; 1; 2; 3; 4]
   - *drop* 4 [0; 1; 2]

   Describe in words what the function *drop* does.

   Solution: *drop* n xs drops from xs its longest prefix whose length is at most n.

4. **Length**. The function *length* has type $'a\ list \rightarrow int$. Try some inputs, and describe in words what this function does.

   Solution: *length* xs computes the length of xs.

5. **Append** The operator (@) has type $'a\ list \rightarrow 'a\ list \rightarrow 'a\ list$.
   1. Try the following expressions:
      - [0; 1; 2] @[3; 4].
      - [0; 1; 0] @[1; 0].

      Describe in words what the operator (@) does.

      Solution: xs @ ys concatenates the two lists xs and ys.

   2. Which of the following expressions are type correct? For the type-correct expressions, what do they evaluate to?
      - []@[1; 2; 3]@[4]
      - [[ ]]@[1; 2; 3]
      - [[ ]]@[ ]
      - [ ]@[[ ]]
      - [ ]@[ ]

      Solution:

      - $-\ :\ int\ list = [1; 2; 3; 4]$.
      - Type error. The first argument has type $'a\ list\ list$, while the second argument has type $int\ list$.

- $- : {'a}\ list\ list = [[\,]]$.
- $- : {'a}\ list\ list = [[\,]]$.
- $- : {'a}\ list = [\,]$.

3. Can you think of a property that relates *take*, *drop* and (@)?

Solution: For all n and xs, we have $take\ n\ xs\ @\ drop\ n\ xs\ =\ xs$.

6. Strings and lists of characters are different types in OCaml (unlike in Haskell). We have defined functions *explode* and *implode* in `Utils.ml` that perform the version. Try

   - *explode* `"functional programming"`
   - *implode* `['f';'u';'n';'c';'t';'i';'o';'n']`

7. Define function $rotate : int \rightarrow {'a}\,list \rightarrow {'a}\,list$ such that $rotates\,n\,xs$, when $0 \leq n \leq length\,xs$, rotates xs leftwards by n positions. For example:

   - $rotate\,2\,[0;1;2;3;4;5] = [2;3;4;5;0;1]$
   - $implode\,(rotate\,3\,(explode$ `"flolac"`$)) =$ `"lacflo"`

   Hint: use *take*, *drop*, and (@).

Solution: **let** $rotate\ n\ xs\ =\ drop\ n\ xs\ @\ take\ n\ xs$

8. We have also defined a function $fromTo : int \rightarrow int \rightarrow int\ list$ in `Utils.ml`. Knowing its type, try some inputs, and describe in words what this function does.

Solution: $fromTo\ m\ n$ generates the list $[m; m+1; \ldots n-1]$.

9. In the OCaml toplevel, issue the coomand `open List`, to gain access to some more functions on lists. The function *combine* has type ${'a}\ list \rightarrow {'b}\ list \rightarrow {'a} * {'b}\ list$. Try

   - *combine* $[0;1;2;3]$ (*explode* `"abcd"`)
   - *combine* $[0;1;2]$ (*explode* `"abcd"`)

   and describe in words what this function does.

10. Now we will take a look at some *higher-order functions* — functions that takes functions as inputs or returns functions. The first candidate is *filter*, having type $({'a} \rightarrow bool) \rightarrow {'a}\,list \rightarrow {'a}\,list$. Try

    - *filter* *is_even* $[0;1;2;3;4]$
    - *filter* $(\mathbf{fun}\,x \rightarrow x\ mod\ 3 = 0)\ [0;1;2;3;4]$

    Describe in words what *filter* does.

11. Try the function *map*:

    - $map\ not\ [true; true; false]$

- $map\ (\mathbf{fun}\, x \to x\ mod\ 4)\ (fromTo\ (-10)\ 10)$

Answer the questions:

- What should the type of map be?
- Describe in words what map does.

> **Solution:**
>
> - $('a \to 'b) \to 'a\ list \to 'b\ list$
> - $map\ f\ xs$ applies f to each element of xs.

12. Define $count\ :\ ('a \to bool) \to 'a\, list \to int$ such that $count\ p\ xs$ returns the number of elements in xs that satisfies p.

> **Solution:** $\mathbf{let}\ count\ p\ =\ length \ll filter\ p$

13. Define $index\ :\ 'a\, list \to (int * 'a)\, list$ such that $index\ xs$ labels each element in xs with its index. For example,

$$index\ (explode\ \texttt{"flolac"})\ =\ [(0,'f'); (1,'l'); (2,'o'); (3,'l'); (4,'a'); (5,'c')]$$

> **Solution:** $\mathbf{let}\ index\ xs\ =\ combine\ (fromTo\ 0\ (length\ xs))\ xs$

14. Define $positions\ :\ ('a \to bool) \to 'a\, list \to int\, list$ such that $positions\ p\ xs$ returns the indexes of elements in xs that satisfies p. For example

$$positions\ is\_even\ [2;4;5;3;6]\ =\ [0;1;4]$$

> **Solution:** $\mathbf{let}\ positions\ x\ xs\ =\ map\ fst\ (filter\ (\mathbf{fun}\ (\_, y) \to y == x)\ (index\ xs))$.
>
> Equivalently, $\mathbf{let}\ positions\ x\ xs\ =\ map\ fst\ (filter\ (((==)\, x) \ll snd)\ (index\ xs))$.

Types

1. Suppose f and g have the following types:

$$f : int \to int$$
$$g : int \to int \to int$$

Let h be defined by

$$h\ x\ y =\ f\ (g\ x\ y)$$

   1. What is the type of h?

Solution: $int \to int \to int$

2. Which, if any, of the following statements is true?

$$\begin{aligned} &h &&= f \ll g \\ &h\ x &&= f \ll (g\ x) \\ &h\ x\ y &&= (f \ll g)\ x\ y \end{aligned}$$

Solution:

- $h = f \ll g$: not true. This is equvalent to $h\ x\ y = (f \ll g)\ x\ y = ((f \ll g)\ x)\ y = (f\ (g\ x))\ y$.
- $h\ x = f \ll (g\ x)$: true. We reason:

$$\begin{aligned} & h\ x\ y \\ = \quad & \{ \text{ definition of } h \ \} \\ & f\ (g\ x\ y) \\ = \quad & \{ \text{ application associates to the left } \} \\ & f\ ((g\ x)\ y) \\ = \quad & \{ \text{ definition of } \ll \ \} \\ & (f \ll g\ x)\ y \end{aligned}$$

  Thus the definition of h is equivalent to $h\ x = f \ll g\ x$.

- $h\ x\ y = (f \ll g)\ x\ y$: not true. This is equivalent to $h = f \ll g$.

2. Give suitable polymorphic type assignments for the following functions:

$$\begin{aligned} &\textbf{let}\ \mathit{const}\ x\ y &&= x \\ &\textbf{let}\ \mathit{subst}\ f\ g\ x &&= f\ x\ (g\ x) \\ &\textbf{let}\ \mathit{apply}\ f\ x &&= f\ x \\ &\textbf{let}\ \mathit{flip}\ f\ x\ y &&= f\ y\ x \end{aligned}$$

Solution:

$$\begin{aligned} &\mathit{const} &&: {}'a \to {}'b \to {}'a \\ &\mathit{subst} &&: ({}'a \to {}'b \to {}'c) \to ({}'a \to {}'b) \to {}'a \to {}'c \\ &\mathit{apply} &&: ({}'a \to b) \to {}'a \to {}'b \\ &\mathit{flip} &&: ({}'a \to b \to {}'c) \to {}'b \to {}'a \to {}'c \end{aligned}$$

3. Define a function *swap* such that:

$$\mathit{flip}\ (\mathit{curry}\ f) = \mathit{curry}\ (f \ll \mathit{swap})$$

for all $f : {}'a * {}'b \to {}'c$.
Hint: there are at least two ways to construct *swap*:

1. use equational reasoning, construct a definition of *swap* such that both sides simply to the same expression, or
2. deduce its type, guess a definition using the type, and prove the equality above.

Solution: We will try constructing *swap* by equational reasoning. The definition of *flip* was given in the previous exercise:

$$\textbf{let}\ \mathit{flip}\ f\ x\ y = f\ y\ x$$

We start with simplifying the left-hand side:

$$\begin{array}{cl} & \mathit{flip}\ (\mathit{curry}\ f)\ x\ y \\ = & \{\ \text{definition of } \mathit{flip}\ \} \\ & (\mathit{curry}\ f)\ y\ x \\ = & f\ (y, x) \end{array}$$

And then the right-hand side:

$$\begin{array}{cl} & \mathit{curry}\ (f \cdot \mathit{swap})\ x\ y \\ = & \{\ \text{definition of } \mathit{curry}\ \} \\ & (f \cdot \mathit{swap})\ (x, y) \\ = & \{\ \text{definition of } (\ll)\ \} \\ & f\ (\mathit{swap}\ (x, y)) \end{array}$$

The goal is to have $f\ (y, x) = f\ (\mathit{swap}\ (x, y))$:

$$\begin{array}{cl} & f\ (y, x) = f\ (\mathit{swap}\ (x, y)) \\ \Leftarrow & \{\ \text{Leibniz}\ \} \\ & (y, x) = \mathit{swap}\ (x, y) \end{array}$$

Thus we pick $\mathit{swap}\ (x, y) = (y, x)$. The rule "Leibniz" states that $f\ m = fn$ if $m = n$.

You may also try to guess what *swap* could be from its type. We haven't properly talked about type inference. However, assuming that $f :: (a, b) \to c$, the left-hand side has type

$$\mathit{flip}\ (\mathit{curry}\ f) : {}'b \to {}'a \to {}'c$$

and *swap* must have type

$$\mathit{swap} :: ({}'b * {}'a) \to ({}'a * {}'b)$$

You may then guess that $\mathit{swap}\ (x, y) = (y, x)$.

However, this does not consitute a proof. To prove that $\mathit{flip}\ (\mathit{curry}\ f) = \mathit{curry}\ (f \cdot \mathit{swap})$ you still have to go through the equational reasoning above.

4. Can you find polymorphic type assignments for the following functions?

$$\begin{array}{l} \textbf{let}\ \mathit{strange}\ f\ g = g\ (f\ g) \\ \textbf{let}\ \mathit{stranger}\ f\ \ = f\ f \end{array}$$

Solution: *strange* : $(('a \to 'b) \to 'a) \to ('a \to 'b) \to 'b$

stranger cannot be typed in the Hindly-Milner system.