Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 3

λ-calculus as a programming language

Topics covered:

- Computability and Turing completeness
- λ-calculus and programming languages
- Representing basic data types
- Recursive functions and fixpoint combinators
- Let expressions
- Steps to a declarative programming language

Computability; Turing completeness

Before computers, stimulated by the *Entscheidungsproblem*, many equivalent definitions of 'computable' were proposed.

- Turing machines
- Register machines
- μ-recursive functions
- Markov algorithms
- λ-calculus

Many of them correspond to programming methods, even though they came before computers! For example, Markov algorithms are related to SNOBOL, and λ-*calculus to functional programming*.

The influence of λ-calculus

LISP took some ideas from λ-calculus, but was not really based on it. It was really with Landin that the influence of λ-calculus took off.

Landin pointed out how λ-calculus could be used to explain features of some existing programming languages, e.g. the scoping of variables in Algol 60.

He proposed a simple functional language ISWIM ('If you See What I Mean'), based on λ-calculus. Many of today's functional languages, including ML, are descended from ISWIM.

ML started life as the metalanguage for the Edinburgh LCF theorem prover. It added to ISWIM an innovative type system, and *exceptions*. The narrow design focus resulted in a coherent language.

We'll start with λ-calculus and work up!

Booleans (1)

We can define the truth values as follows:

$$\begin{aligned} \text{true} &\triangleq \lambda x\ y.\ x \\ \text{false} &\triangleq \lambda x\ y.\ y \end{aligned}$$

Then we get conditional expressions, like C's '`?:`' construct:

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E\ E_1\ E_2$$

For example:

$$\begin{aligned} \text{if true then } E_1 \text{ else } E_2 &= \text{true } E_1\ E_2 \\ &= (\lambda x\ y.\ x)\ E_1\ E_2 \\ &= E_1 \end{aligned}$$

Booleans (2)

and:

$$
\begin{aligned}
\text{if false then } E_1 \text{ else } E_2 &= \text{false } E_1\ E_2 \\
&= (\lambda x\ y.\ y)\ E_1\ E_2 \\
&= E_2
\end{aligned}
$$

Once we have the conditional, it is easy to define all the usual logical operations:

$$
\begin{aligned}
\text{not } p &\stackrel{\triangle}{=} \text{if } p \text{ then false else true} \\
p \text{ and } q &\stackrel{\triangle}{=} \text{if } p \text{ then } q \text{ else false} \\
p \text{ or } q &\stackrel{\triangle}{=} \text{if } p \text{ then true else } q
\end{aligned}
$$

Pairs

We can represent ordered pairs as follows:

$$(E_1, E_2) \triangleq \lambda f.\ f\ E_1\ E_2$$

Given this definition, the corresponding destructors for pairs can be defined as:

$$\begin{aligned} \mathrm{fst}\ p &\triangleq p\ \mathrm{true} \\ \mathrm{snd}\ p &\triangleq p\ \mathrm{false} \end{aligned}$$

These work as required, e.g.

$$\begin{aligned} \mathrm{fst}\ (p, q) &= (p, q)\ \mathrm{true} \\ &= (\lambda f.\ f\ p\ q)\ \mathrm{true} \\ &= \mathrm{true}\ p\ q \\ &= (\lambda x\ y.\ x)\ p\ q \\ &= p \end{aligned}$$

Tuples

We get tuples by iterating the pairing operation:

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots E_n))$$

We can then define general 'projections':

$$(p)_1 = \text{fst } p$$

and for $i > 1$:

$$(p)_i = \text{fst } (\text{snd}^{i-1}\ p)$$

We have $(E_1, E_2, \ldots, E_n)_i = E_i$ for $i \leq n$.

Uncurried functions

We can now define functions of multiple arguments that are non-curried. Here are functions to translate between curried and uncurried versions:

$$\begin{aligned} \mathrm{CURRY}_n\ f &\triangleq \lambda x_1 \ \cdots \ x_n.\ f(x_1, \ldots, x_n) \\ \mathrm{UNCURRY}_n\ g &\triangleq \lambda p.\ g\ (p)_1 \ \cdots \ (p)_n \end{aligned}$$

Now we can write $\lambda(x_1, \ldots, x_n).\ t$ for:

$$\mathrm{UNCURRY}_n\ (\lambda x_1 \ \cdots \ x_n.\ t)$$

Then we have generalized β-conversion:

$$(\lambda(x_1, \ldots, x_n).t[x_1, \ldots, x_n])(t_1, \ldots, t_n) = t[t_1, \ldots, t_n]$$

Natural numbers

We can encode numbers in many ways. The simplest seems to be as *Church numerals*:

$$\mathbf{n} \triangleq \lambda f\ x.\ f^n\ x$$

This means:

$$\begin{aligned}
\mathbf{0} &= \lambda f\ x.\ x \\
\mathbf{1} &= \lambda f\ x.\ f\ x \\
\mathbf{2} &= \lambda f\ x.\ f\ (f\ x) \\
\mathbf{3} &= \lambda f\ x.\ f\ (f\ (f\ x)) \\
\mathbf{4} &= \lambda f\ x.\ f\ (f\ (f\ (f\ x)))
\end{aligned}$$

and so on.

Successor

The following operation adds one to a Church numeral:

$$\mathrm{SUC} \triangleq \lambda n\ f\ x.\ n\ f\ (f\ x)$$

Indeed:

$$\begin{aligned}
\mathrm{SUC}\ \mathbf{n} &= (\lambda n\ f\ x.\ n\ f\ (f\ x))(\lambda f\ x.\ f^n\ x) \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^n\ x) f\ (f\ x) \\
&= \lambda f\ x.\ (\lambda x.\ f^n\ x)(f\ x) \\
&= \lambda f\ x.\ f^n\ (f\ x) \\
&= \lambda f\ x.\ f^{n+1}\ x \\
&= \mathbf{n+1}
\end{aligned}$$

We can also test if a Church numeral is zero with:

$$\mathrm{ISZERO}\ n \triangleq n\ (\lambda x.\ \mathrm{false})\ \mathrm{true}$$

Addition

Here's addition of Church numerals:

$$m + n \triangleq \lambda f\ x.\ m\ f\ (n\ f\ x)$$

Indeed:

$$\begin{aligned}
m + n &= \lambda f\ x.\ m\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^m\ x)\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda x.\ f^m\ x)\ (n\ f\ x) \\
&= \lambda f\ x.\ f^m\ (n\ f\ x) \\
&= \lambda f\ x.\ f^m\ ((\lambda f\ x.\ f^n\ x)\ f\ x) \\
&= \lambda f\ x.\ f^m\ ((\lambda x.\ f^n\ x)\ x) \\
&= \lambda f\ x.\ f^m\ (f^n\ x) \\
&= \lambda f\ x.\ f^{m+n} x
\end{aligned}$$

Multiplication

and here's multiplication:

$$m * n \triangleq \lambda f\ x.\ m\ (n\ f)\ x$$

Indeed:

$$\begin{aligned}
m * n &= \lambda f\ x.\ m\ (n\ f)\ x \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^m\ x)\ (n\ f)\ x \\
&= \lambda f\ x.\ (\lambda x.\ (n\ f)^m\ x)\ x \\
&= \lambda f\ x.\ (n\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda f\ x.\ f^n\ x)\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda x.\ f^n\ x)^m\ x \\
&= \lambda f\ x.\ (f^n)^m\ x \\
&= \lambda f\ x.\ f^{mn} x
\end{aligned}$$

Predecessor

This is much harder. First we define an auxiliary function on pairs:

$$\begin{aligned}\mathrm{PREFN} \triangleq \lambda f\ p.(\ &\mathrm{false},\\ &\mathrm{if\ fst}\ p\ \mathrm{then\ snd}\ p\ \mathrm{else}\ f(\mathrm{snd}\ p))\end{aligned}$$

We have:

$$\begin{aligned}\mathrm{PREFN}\ f\ (\mathrm{true}, x) &= (\mathrm{false}, x)\\ \mathrm{PREFN}\ f\ (\mathrm{false}, x) &= (\mathrm{false}, f\ x)\end{aligned}$$

We can use this to 'throw away' one of the applications of f in $f^n\ x$. This trick is due to Kleene. Now define:

$$\mathrm{PRE}\ n \triangleq \lambda f\ x.\, \mathrm{snd}(n\ (\mathrm{PREFN}\ f)\ (\mathrm{true}, x))$$

Recursive functions (1)

This looks difficult without using names for functions. We can do it though — the key is to use *fixed point combinators.*

A closed term Y is said to be a fixed point combinator if, when applied to a function f, it gives a fixed point x for f, i.e. an x with $f(x) = x$. Hence for all f:

$$f(Y\ f) = Y\ f$$

We can get an idea how to produce such a combinator by recalling the Russell paradox. We defined $R = \lambda x.\ \neg(x\ x)$ and found $R\ R = \neg(R\ R)$. All we need to do is replace $\neg$ with the function f concerned. Thus:

$$Y \triangleq \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$$

Recursive functions (2)

It is a simple matter to show that Y works as advertised:

$$
\begin{aligned}
Y f &= (\lambda f.\, (\lambda x.\, f(x\ x))(\lambda x.\, f(x\ x)))\ f \\
&= (\lambda x.\, f(x\ x))(\lambda x.\, f(x\ x)) \\
&= f((\lambda x.\, f(x\ x))(\lambda x.\, f(x\ x))) \\
&= f(Y\ f)
\end{aligned}
$$

An alternative to Y is given by:

$$T \triangleq (\lambda x\ y.\, y\ (x\ x\ y))\ (\lambda x\ y.\, y\ (x\ x\ y))$$

This has the even better property that $T\ f \longrightarrow f(T\ f)$.

Recursive functions (3)

How do we use a fixpoint combinator to define recursive functions? Consider the factorial function. We would like to define fact such that:

$$\text{fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

First transform it to:

$$\text{fact} = \lambda n.\, \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

so if we define:

$$F \triangleq \lambda f\ n.\, \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)$$

we find that fact is the fixpoint of F, i.e. $\text{fact} = F\ \text{fact}$. Therefore we define:

$$\text{fact} = Y\ F$$

Let expressions

It's often useful to bind expressions to names, to avoid repeating the expression many times. We will allow the following syntactic sugar:

$$\text{let } x = s \text{ in } t \triangleq (\lambda x.\, t)\ s$$

We also allow multiple *parallel* bindings using the following:

$$\text{let } x_1 = s_1 \text{ and } \cdots \text{ and } x_n = s_n \text{ in } t$$

which is syntactic sugar for:

$$(\lambda(x_1, \ldots, x_n).\, t)\ (s_1, \ldots, s_n)$$

Also, we may say $\text{let } f\ x_1\ \cdots x_n = t$, which is a sugaring for

$$\text{let } f = \lambda x_1\ \cdots\ x_n.\, t$$

Recursive let expressions

Normally let is interpreted nonrecursively. When we want to make a recursive definition, we add the word rec. For example:

$$\begin{aligned}\text{let rec fact}(n) = \; & \text{if ISZERO } n \text{ then } 1 \\ & \text{else } n * \text{fact}(\text{PRE } n)\end{aligned}$$

is a shorthand for

$$\text{let fact} = Y\ F$$

where as above:

$$F = \lambda f\ n.\, \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)$$

Functional programs

We have said that a functional program is simply an expression.

However, we normally extract certain common and interesting subexpressions as `let` constructs, so the program has the form: a series of definitions, followed by the main expression, e.g.

$$\begin{aligned} \text{let rec fact}(n) = \ & \text{if ISZERO } n \text{ then } 1 \\ & \text{else } n * \text{fact}(\text{PRE } n) \text{ in} \end{aligned}$$

$\ldots$

$fact(6)$

Thus the program can be read 'declaratively', as a set of mathematical equations giving some properties of the relevant notions. The computational information is all *implicit*.

Computational interpretation

The computational interpretation *is* still there though!

Roughly, all the equations are read left-to-right and used to expand away any instance of the relevant names. Then we repeatedly perform β-conversions.

The main question is how to perform the series of β-conversions. We know that this can make a difference with respect to termination.

Different functional languages make different choices, as we shall see.

The next step, however, is to think about *types*.