An Introduction to Functional Programming

Tyng–Ruey Chuang

2007 Formosan Summer School
on Logic, Language, and Computation
July 2–13, 2007

This course note ...

• ... is prepared for the 2007 Formosan Summer School on Logic, Language, and Computation (held in Taipei, Taiwan),

• ... is made available from the Flolac '07 web site:

  http://www.iis.sinica.edu.tw/~scm/flolac07/

• ... and is released to the public under a Creative Commons Attribution-ShareAlike 2.5 Taiwan license:

  http://creativecommons.org/licenses/by-sa/2.5/

Course outline


Unit 2. Fold/unfold functions for data types;
( Untyped) lambda calculus.

Unit 3. Parametric modules.

Each unit consists of 2 hours of lecture and 1 hour of lab/tutor. Examples will be given in Objective Caml (O’Caml). Useful online resources about O’Caml:

• Web site: http://caml.inria.fr/

• Book: Developing Applications with Objective Caml.
  URL: http://caml.inria.fr/pub/docs/oreilly-book/
1 Lambda calculus

Untyped lambda calculus

- Introduced by Alonzo Church and his student Stephen Cole Kleene in the 1930s to study computable functions — even before there are computers!
- A (very simple) formal system for defining functions and their operational meanings, yet is shown to be as powerful as other systems.
- It is a basis of early programming languages (such as Lisp). Typed lambda calculi — there are many variations — are the bases of modern functional languages (such as O’Caml and Haskell).

Untyped lambda terms

The set of all (untyped) lambda terms $T$ consists of the following terms:

- $x$ where $x$ is a variable;
- $\lambda x . t$ where $x$ is a variable and $t \in T$ is a lambda term; (to denote function abstraction)
- $t_1 \ t_2$ where $t_1, t_2 \in T$ are lambda terms; (to denote function application)
- $(t)$ where $t \in T$ is a lambda term.

Examples:

$x, \ y, \ z, \ xyz$
$x \ y \ z, \ \lambda x . \lambda y . \ z, \ (\lambda x . \lambda y . x) u v, \ (\lambda x . x x)(\lambda x . x x)$

Notational conventions

- Function application is left associative. For example:
  
  
  $(\lambda x . \lambda y . x)(\lambda x . x)z$

  means

  $((\lambda x . \lambda y . x)(\lambda x . x))z$

- The body of a function abstraction extends to the right as far as possible. For example,

  $\lambda x . \lambda y . \lambda z . z \ y \ x$

  means

  $\lambda x . (\lambda y . (\lambda z . (z \ y \ x)))$

In case of doubt, use parentheses to make clear the intended meaning of a term.
Scope of variables

- An occurrence of variable \( x \) is *bound* if it appears in the body \( t \) of a function abstraction \( \lambda x . t \).
- An occurrence of variable \( x \) is *free* if it appears in a position where it is not bound by an enclosing abstraction of \( x \).

In the following example,

\[
(\lambda x . \lambda y . (\lambda z . y) x) x
\]

the outer occurrence of \( x \) is free while the inner occurrence of \( x \) is bound. The only occurrence of \( y \) is bound. The variable \( z \) does not occur in the function abstraction \( \lambda z . y \).

Two computational rules

**alpha renaming** Two lambda terms are equivalent if they differ only in the naming of bound variables. For example, these two terms are equivalent:

\[
(\lambda x . \lambda y . x y (\lambda x . x)) y \equiv_\alpha (\lambda x . \lambda z . x z (\lambda x . x)) y
\]

**beta reduction** A term \((\lambda x . t_1) t_2\) — called a redex — is converted to the term \(t_1 [t_2/x]\) where all free variables \( x \) in \( t_1 \) are replaced by term \( t_2 \). For example,

\[
(\lambda x . \lambda z . x z (\lambda x . x)) y \rightarrow_\beta \lambda z . y z (\lambda x . x)
\]

Use alpha renaming to avoid accidental capture of free variables during a beta reduction!

Normal forms and reduction strategies

A lambda term is in normal form if it has no more redex. A lambda term may contain many redexes. Several strategies to select redex for beta reduction:

**Normal order reduction** always selects the leftmost, outermost redex, until no more redexes is left.

**Call-by-name reduction** always selects the leftmost, outermost redex, but never reduces inside function abstractions. (Haskell; call-by-need actually)

**Call-by-value reduction** always selects the leftmost, innermost redex, but never reduces inside function abstractions. (O’Caml)

*Church-Rosser theorem:* the normal order reduction strategy will always lead to the normal form if there is one.
Non-terminating reduction sequences
There are lambda terms that have no normal form. An example:

\[(\lambda x.x x)(\lambda x.x x) \rightarrow_{\beta} (\lambda x.x x)(\lambda x.x x) \rightarrow_{\beta} \ldots\]

Let \(\omega\) denote the lambda term \(((\lambda x.x x)(\lambda x.x x))\), and \(Q\) denote the lambda term \((\lambda x.\lambda y.x)\omega\). Then,

Normal order reduction

\[Q \rightarrow_{\beta} \lambda y.\omega \rightarrow_{\beta} \lambda y.\omega \rightarrow_{\beta} \ldots\]

Call-by-name reduction

\[Q \rightarrow_{\beta} \lambda y.\omega \nrightarrow\]

Call-by-value reduction

\[Q \rightarrow_{\beta} Q \rightarrow_{\beta} Q \rightarrow_{\beta} \ldots\]

Church booleans

true := \(\lambda t.\lambda f.t\)
false := \(\lambda t.\lambda f.f\)
if := \(\lambda b.\lambda p.\lambda q. b \ p \ q\)

Example:

\[\text{if true } P \ Q = (\lambda b.\lambda p.\lambda q. b \ p \ q) \text{ true } P \ Q\]
\[= (\lambda t.\lambda f.t) \ P \ Q\]
\[\rightarrow P\]

More definitions:

and := \(\lambda p.\lambda q.\text{if } p \ q \text{ false}\)

or := \(\lambda p.\lambda q.\text{if } p \ true \ q\)
Church numerals

\[O := \lambda f . \lambda x . x\]

\[1 := \lambda f . \lambda x . f x\]

\[2 := \lambda f . \lambda x . f (f x)\]

\[n := \lambda f . \lambda x . f^{(n)} x\]

\[\text{succ} := \lambda x . \lambda f . \lambda n . f (x f n)\]

\[\text{plus} := \lambda x . \lambda y . \lambda f . \lambda n . x f (y f n)\]

\[\text{times} := \lambda x . \lambda y . x (\text{plus} y 0)\]

\[\text{iszero} := \lambda n . n (\lambda x . \text{false}) \text{true}\]

Example:

\[\text{succ} 2 = (\lambda x . \lambda f . \lambda n . f (x f n)) 2 \]
\[\quad\rightarrow \lambda f . \lambda n . f (2 f n)\]
\[\quad= \lambda f . \lambda n . f ((\lambda f . \lambda n . f (f n)) f n)\]
\[\quad\rightarrow \lambda f . \lambda n . f (f (f n)) = 3\]

Recursion via fixed-point

\[Y := \lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))\]

\(Y\) is a fixed-point computing function. For any lambda term \(F\),

\[Y F = (\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))) F\]
\[\quad\rightarrow (\lambda x . F (x x)) (\lambda x . F (x x))\]
\[\quad\rightarrow F ((\lambda x . F (x x)) (\lambda x . F (x x)))\]
\[\quad= F \,(Y \,F)\]

That is, \((Y \,F)\) is a fixed-point of \(F\).
The factorial function, once again

Let

\[ F := \lambda f . \lambda n . \text{if} \ (\text{iszero} \ n) \ 1 \ (\text{times} \ n \ (f \ (\text{pred} \ n))) \]

Then

\[ Y F \ 3 \rightarrow F (Y F) 3 \]
\[ = \text{if} \ (\text{iszero} \ 3) \ 1 \ (\text{times} \ 3 \ ((Y F) \ (\text{pred} \ 3))) \]
\[ \rightarrow \text{times} \ 3 \ (Y F \ 2) \]
\[ \rightarrow \text{times} \ 3 \ (F (Y F) 2) \]
\[ \rightarrow \text{times} \ 3 \ (\text{times} \ 2 \ (Y F \ 1)) \]
\[ \rightarrow \text{times} \ 3 \ (\text{times} \ 2 \ (F (Y F) 1)) \]
\[ \rightarrow \text{times} \ 3 \ (\text{times} \ 2 \ (\text{times} \ 1 \ (Y F \ 0))) \]
\[ \rightarrow \text{times} \ 3 \ (\text{times} \ 2 \ (\text{times} \ 1 \ (F (Y F) 0))) \]
\[ \rightarrow \text{times} \ 3 \ (\text{times} \ 2 \ (\text{times} \ 1 \ 1)) \]
\[ \rightarrow 6 \]

Is that all?

\[ \text{succ} := \lambda x . \lambda f . \lambda n . f \ (x \ f \ n) \]

\[ \text{pred} := \ ??? \]

The definition of pred turns out to be not so easy!