CSE399: Advanced Programming

Handout 22

The Lambda Calculus

The Lambda-Calculus

- A model of computation with functions invited by Alonzo Church and his co-workers in the 1930s. (I.e., pre-ENIAC!)
- Formally equivalent in power to Turing machines, Post Correspondence Problem, etc.
- The e. coli of programming language and compiler research
- Foundation of many real-world programming languages, including Haskell, OCaml, SML, Scheme, Lisp, ...

The Pure Lambda-Calculus

The expressions of the pure lambda-calculus are...

`x`	variable
`\x -> t`	abstraction
`t1 t2`	application

The Pure Lambda-Calculus

The expressions of the pure lambda-calculus are...

`x`	variable
`\x -> t`	abstraction
`t1 t2`	application

... and that's all! No `let`-bindings, recursion, numbers, booleans, conditionals, pattern matching, etc., etc.

The Pure Lambda-Calculus

The expressions of the pure lambda-calculus are...

`x`	variable
`\x -> t`	abstraction
`t1 t2`	application

... and that's all! No `let`-bindings, recursion, numbers, booleans, conditionals, pattern matching, etc., etc.

In this language, everything is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function

Encoding Let-Bindings

What's interesting about the lambda-calculus is that, even after we have thrown away all these useful features, we still have an extremely rich and powerful language.

One way to see this is to show how the features that we've removed can be simulated using just functions.

Simple example: *Instead of*

```
let x = s in t
```

we can write

```
(\x -> t) s,
```

which has the same effect.

Encoding Let-Bindings

What's interesting about the lambda-calculus is that, even after we have thrown away all these useful features, we still have an extremely rich and powerful language.

One way to see this is to show how the features that we've removed can be simulated using just functions.

Simple example: *Instead of*

```
let x = s in t
```

we can write

```
(\x -> t) s,
```

which has the same effect.

I.e., we can regard `let` as "just syntactic sugar" for a certain idiom involving function abstraction and application.

Formalities

We'll see many more of these encodings in the rest of the lecture.

But first, let's pause to clarify precisely how computation in the lambda-calculus works.

Formalities: Scoping

The abstraction term `\x -> t` **binds** the variable `x`.

The **scope** of this binding is the **body** `t`.

Occurrences of `x` inside `t` are said to be **bound** by the abstraction.

Occurrences of `x` that are **not** within the scope of an abstraction binding `x` are said to be **free**.

```
\x -> \y -> x y z
\x -> (\y. z y) y
```

Formalities: Substitution

We write $[x \mapsto s]t$ for the substitution of the term `s` for free occurrences of the variable `x` in the term `t`.

$$[\texttt{x} \mapsto \texttt{(\textbackslash y -> y)}]\ \texttt{(\textbackslash z -> z y x)}$$

$$[\texttt{x} \mapsto \texttt{(\textbackslash y -> y)}]\ \texttt{(\textbackslash z -> z (\textbackslash x -> x z) x)}$$

Formalities: Reduction

A primitive step of computation (known as beta-reduction or just reduction) involves substituting an argument for the bound variable in an adjacent abstraction:

$$\texttt{(\textbackslash x -> t) s} \quad \longrightarrow \quad [\texttt{x} \mapsto \texttt{s}]\ \texttt{t}$$

Formalities: Reduction

A primitive step of computation (known as beta-reduction or just reduction) involves substituting an argument for the bound variable in an adjacent abstraction:

$$\texttt{(\textbackslash x -> t) s} \quad \longrightarrow \quad [\texttt{x} \mapsto \texttt{s}]\ \texttt{t}$$

Write $\longrightarrow^*$ for the reflexive, transitive closure of the reduction relation—that is, $\texttt{s} \longrightarrow^* \texttt{t}$ if there is a sequence of zero or more single-step reductions leading from $\texttt{s}$ to $\texttt{t}$.

Encoding Booleans

```
true  = \t -> \f -> t
false = \t -> \f -> f
```

It is easy to calculate that, for any arguments `a` and `b`,

$$\texttt{true m n} \longrightarrow^{*} \texttt{m}$$

$$\texttt{false m n} \longrightarrow^{*} \texttt{n}$$

So, instead of `if b then m else n` we can simply write `b m n`.

```
not = \b -> b false true
```

That is, `not` is a function that, given a boolean value `v`, returns `false` if `v` is `true` and `true` if `v` is `false`.

Encoding Pairs

```
pair = \f -> \s -> \b -> b f s
fst  = \p -> p true
snd  = \p -> p false
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.

By the definition of booleans, this application yields `v` if `b` is `true` and `w` if `b` is `false`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean as an argument to the pair itself.

Encoding Numbers

Idea: represent the number *n* by a function that "repeats some action *n* times."

```
c0  =  \s -> \z -> z
c1  =  \s -> \z -> s z
c2  =  \s -> \z -> s (s z)
c3  =  \s -> \z -> s (s (s z))
```

That is, each number *n* is represented by a term `cn` that takes two arguments, `s` and `z` (for "successor" and "zero"), and applies `s`, *n* times, to `z`.

Computing with Numbers

```
-- Successor
succ  =  \n ->
            \s -> \z -> s (n s z)

-- Addition
plus  =  \m -> \n ->
            \s -> \z -> m s (n s z)

-- Multiplication
times  =  \m -> \n ->
             m (plus n) c0

-- Zero test
iszero  =  \m ->
              m (\x -> false) true
```

Computing with Numbers

```
-- Successor
succ  =  \n ->
            \s -> \z -> s (n s z)

-- Addition
plus  =  \m -> \n ->
            \s -> \z -> m s (n s z)

-- Multiplication
times  =  \m -> \n ->
             m (plus n) c0

-- Zero test
iszero  =  \m ->
              m (\x -> false) true
```

What about predecessor???

Predecessor

```
zz  =  pair c0 c0

ss  =  \p -> pair (snd p) (succ (snd p))

pred  =  \m -> fst (m ss zz)
```

Normal Forms

A normal form is a term that cannot take any reduction steps.

E.g.,

```
\x -> \y -> x
```

Normal Forms

A normal form is a term that cannot take any reduction steps.

E.g.,

```
\x -> \y -> x
```

A normalizable term is one that will eventually reach a normal form after some finite number of reduction steps.

E.g.

```
not true
```

```
pred c1000
```

Normal Forms

A normal form is a term that cannot take any reduction steps.

E.g.,

```
\x -> \y -> x
```

A normalizable term is one that will eventually reach a normal form after some finite number of reduction steps.

E.g.

```
not true
```

```
pred c1000
```

Question: Is every lambda-term normalizable?

Divergence

Answer: No!

```
omega  =  (\x -> x x) (\x -> x x)
```

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it **diverges**.

Answer: No!

```
omega  =  (\x -> x x) (\x -> x x)
```

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it **diverges**.

(N.b.: this example and the ones following cannot be typed in Haskell.)

Answer: No!

```
omega  =  (\x -> x x) (\x -> x x)
```

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it **diverges**.

(N.b.: this example and the ones following cannot be typed in Haskell.)

Being able to write a divergent computation does not seem especially useful in itself. However, there are variants of `omega` that are very useful...

The "fixed-point combinator":

```
fix  =  \f -> (\x -> f (x x)) (\x -> f (x x))
```

The "fixed-point combinator":

```
fix  =  \f -> (\x -> f (x x)) (\x -> f (x x))
```

Here the "pattern of divergence" becomes more interesting. Let `f` be some lambda-term. Then:

```
fix f
  ⟶
(\x -> f (x x)) (\x -> f (x x))
  ⟶
f ((\x -> f (x x)) (\x -> f (x x)))
  ⟶
f (f ((\x -> f (x x)) (\x -> f (x x))))
  ⟶
f (f (f ((\x -> f (x x)) (\x -> f (x x)))))
  ⟶
...
```

More concisely:

$$\texttt{fix f} \longrightarrow \texttt{f (fix f)}$$

```
factf = \fct ->
          \n ->
             (iszero n)
                c1
                (times n (fct (pred n)))

fact = fix factf
```

```
    fact c3
--> factf (fix factf) c3
--> (\n -> (iszero n) c1 (times n (fix factf (pred n)))) c3
--> (iszero c3) c1 (times n (fix factf (pred c3)))
--> times c3 (fix factf (pred c3))
--> times c3 (factf (fix factf) (pred c3))
--> times c3
      ((\n -> (iszero n)
                  c1
                  (times n (fix factf (pred n)))) (pred c3))
--> times c3
      ((iszero (pred c3))
          c1
          (times (pred c3) (fix factf (pred (pred c3)))))
--> times c3
      (times (pred c3)
         (fix factf (pred (pred c3))))
--> etc.
```