Computing Science (CMPUT) 325

Nonprocedural Programming

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2016

Lambda Calculus

- We can - in principle - write any kind of program in Lisp now
- Pure Lisp is a pretty small language
- In lambda calculus we make it even smaller
- Remove features from Lisp while showing that we can still compute the same functions
- End goal: lambda calculus as a minimal but complete model of computation

Lambda Functions

- First step: get rid of named functions
- In Lisp so far, each function had a name
- Example `(defun myfunction (x y) body)`
- This was a minor headache when we tried to return a newly computed function as the result of a higher-order function
- It also complicates the language (but simplifies the writing of programs)
- OK, let's have **lambda functions** without names...

Review - Function Definition vs Function Application

- Function definition - what a function does
  - In Lisp: `(defun f (args) body)`
- Function application - using the function in a computation, with some supplied arguments
  - In Lisp: `(f 2 '(a b c))`
- In pure functional programming, **any** computation is just some (possibly complex, nested) process of function applications
- Giving a function a name (such as `f`) simplifies using the function
- But it is not strictly necessary. Instead we can just write down the body of the function whenever we need it

Lambda Function

- Lambda function in Lisp:
- `(lambda (x1 ... xn) body)`
  - `lambda` is a built-in Lisp keyword
  - `(x1 ... xn)` is the list of arguments, same as with named functions
  - *body* is the function definition (what the lambda function does)
- Example: `(lambda (x y) (+ x y))`
- Compare with named function:

  `(defun plus (x y) (+ x y))`

Lambda Function Application

- **Important: A lambda function is not an application**
- To apply a lambda function, need to provide actual values for the arguments:
- `((lambda (x1 ... xn)` *`body`*`) a1 ... an)`
- Note where the brackets go
- `x1 ... xn` are the arguments of the function, used in the body
- `a1 ... an` are the actual parameters
- Example: `((lambda (x y) (+ x y)) 2 3)`
- Compare with named function application: `(plus 2 3)`

Examples of Applying Lambda


```
* ((lambda (x y) (+ (* x x) y)) 4 6)
22
* (mapcar '(lambda (x) (+ x 1)) '(1 2 3 4 5))
debugger invoked on a TYPE-ERROR:
The value (LAMBDA (X) (+ X 1)) is not
of type (OR FUNCTION SYMBOL).
* (mapcar (function (lambda (x) (+ x 1))) '(1
(2 3 4 5 6)
```

- Quoting a lambda expression works with some Lisp's but not sbcl
- Using the function `function` helps Lisp to understand that this is a lambda function definition
- See next slide for details

The Function `function` in Lisp

- `function` is a built-in function that:
  - takes a lambda function as its argument and
  - returns its definition in a form used by the Lisp system
- In sbcl, `(function arg)` compiles the lambda function given by `arg` and returns an internal representation, a **closure**, of the compiled code

  ```
  * (function  (lambda (x) (+ x 1)))
  #<FUNCTION (LAMBDA (X)) {11EAF6E5}>
  ```
- We will soon discuss an example how to implement a closure for the case of an interpreter (and a kind of compiler but not a real one)
- Again, we can wonder why Lisp needs the `function` keyword. Why can't it see that this is a lambda function definition?

`function` vs `funcall` and `apply`

- Note how `function` differs from `funcall` and `apply`
- `function` takes as argument a *function definition* and returns an internal representation of that definition. But it does not apply the function
- `funcall` and `apply` are for *function application*
- Can we apply lambda functions with `funcall` and `apply`? Yes, absolutely!

Applying Lambda Functions

- We can use `funcall` and `apply` as usual
- Instead of the *name* of a function as before, now give it a lambda function

```
* (funcall
      (function (lambda (x y) (+ (* x x) y)))
      4 6)
22

* (apply (function (lambda (x) (+ x 1)))
         '(3))
4
```

Lambda Calculus

- Lambda calculus is a formal, abstract language
- All functions are defined without giving a name
- We can understand the foundations of functional programming by studying the properties of this formal language
- Lisp is based on lambda calculus but not a different language
- Lisp can deal with expressions from lambda calculus, but needs some help to understand them

Lambda Calculus vs Lisp

- Lambda notation is not frequently used in practical Lisp programming
- In most situations, using a name to refer to a function is more convenient...
- ...especially if you want to call the function more than once
- Lambda calculus is the foundation for functional programming
- Minimal, abstract model of computation
- We can study the essence of functional programming (or more general computing) by using such a minimal language

Language - the Syntax of Lambda Calculus

- Formal language with only four concepts:

```
[function]       := (lambda (x) [expression])
[application]    := ([expression] [expression])
[expression]     := [identifiers]
                  | [application]
                  | [function]
[identifiers]    := a | b | ...
```

Comments on Syntax


```
[identifiers] := a | b | ...
```

- `identifiers` corresponds to atoms in Fun or Lisp

```
[function] := (lambda (x) [expression])
```

- `function` defines a lambda function

```
[application] := ([expression]
                  [expression])
```

- A function application. Both function and argument can be any expressions

```
[expression]     := [identifiers]
                  | [application]
                  | [function]
```

- `expression` corresponds to s-expression in Lisp.

More Comments

- All valid expressions defined by this language are called **lambda expressions**
- The definition is **recursive**:
  an application consists of two expressions,
  each of which can again be an application,
  ...
- Lambda expressions can be nested
- We will see that lambda expressions can represent any computation (!)

Comments on Functions


```
[function] := (lambda (x) [expression])
```

- We only have unary functions - functions that take one parameter
- Why?
- We will soon see that this is not a restriction
- Any n-ary function (function with n arguments) can be equivalently defined using only unary functions
- To understand the model of computation for general functional programming, it is enough to understand computation with unary functions

Curried Functions

- Goal: define an n-ary function by an equivalent unary function
- Sounds impossible? It's not!
- Can solve this using higher order functions
- Main idea: split one n-ary function application into a series of n unary function applications
- Each application "eats" one argument and produces a new function

Curried Functions - Main Idea

- Function takes only the first argument
- It produces as result a new function
- This function now takes the second argument
- It produces as result a new function...
- etc
- The function that takes the last argument will have all other argument values “hardcoded”
- It was computed on the fly from all these previous function applications

Example

- A function that returns the smaller of two numbers
- In Lisp, using a named function:

```
(defun smaller (x y)
        (if (< x y) x y)
)
```

- Equivalent in lambda calculus:

```
(lambda (x)
             (lambda (y) (if (< x y) x y))
)
```

- This is a lambda function with argument x
- The body of the lambda function is *another lambda function*

Example Continued


```
(lambda (x)
            (lambda (y) (if (< x y) x y))
)
```

((lambda (x) body) 5)

- What does this mean?
- The overall structure is `(lambda (x) body)`
- This is a function of a single argument x
- This function, when applied, returns another function (the one in the body)
- When that function `(lambda (y) ...)` is created, it will contain the argument that was supplied for x

(lambda (y) (if (< 5 y) 5 y))

((lambda (y) (if (< 5 y) 5 y)7)

Example Continued

(if (< 5 7) 5 7)

```
(lambda (x)
        (lambda (y) (if (< x y) x y))
)
```

- In other words `(lambda (x) ...)` will return a function:
- `(lambda (y) (if (< x y) x y))`
- But in that function, the x will have been replaced by the given argument
- It is like returning a specialized 1-argument function, as in `(defun smaller5y (y) (if (< 5 y) 5 y))`
- Next, this function can take single argument y and compute the smaller of (hardcoded) x and y

Example of Reduction of an Application

- **Reduction** is the process of evaluating lambda expressions
- Example now, details later.

```
(((lambda (x)
        (lambda (y) (if (< x y) x y))) 4) 9)
→ ((lambda (y) (if (< 4 y) 4 y)) 9)
→ (if (< 4 9) 4 9)
→ 4
```

- Note carefully where the brackets are put in the application. Are you surprised?
- The last step was not “pure lambda calculus”, we assumed a built-in function < that behaves as in Lisp

Same in Lisp

- We can do the same example in Lisp, but:
- We need to tell Lisp what the `function`'s and the `funcall`'s are
- See next slide

Same in Lisp


```
* (funcall
    (funcall
      (function
        (lambda (x)
           (function
              (lambda (y)
                 (if (< x y) x y))))
      )
      4
    )
    9
  )
4
```

Example: Calling a "Computed" Function


```
  ((lambda (x) (x 2)) (lambda (z) (+ z 1)))
→ ((lambda (z) (+ z 1)) 2)
→ (+ 2 1)
```

- The expression is a function application
- The function is (lambda (x) (x 2))
- The argument is (lambda (z) (+ z 1))
- The body of the function is (x 2)
- In the application, replace the x in the body with the given argument

Example continued

- `((lambda (z) (+ z 1)) 2)`

$\rightarrow$ `(+ 2 1)`

- The resulting expression is again a function application
- The function is (lambda (z) (+ z 1))
- The argument is 2
- The body of the function is (+ z 1)
- In the application, replace the z in the body with the given argument

Summary

- Lambda calculus: expression is either, identifier, or function, or application
- Recursive definition - any expression can contain nested expressions in applications.
- Reduction is the process of computation in lambda calculus
- Two examples:
  - a function to be called is itself the result of another application
  - a function to be called is the argument to another function
- Next: more about reductions