The Lambda Calculus

Luca Abeni
luca.abeni@santannapisa.it

Minimalistic Functional Programming Languages

- What is the simplest possible functional programming language?
- Difficult to say what is the *simplest*, but a lot of high-level features are not essential...
  - Global environment / let expressions
  - Multivariable functions
  - Data types
  - ...
- What is really needed?
  - Names / identifiers (irreducible terms)
  - Function definition (abstraction)
  - Function application

Defining Functions: Lambda!

- Function definition: expression evaluating to a function
  - Various languages have it: Standard ML has `fn x => e`, C++ has `[](auto x){return e;};`, ...
  - `x`: formal parameter
  - `e`: expression dependent on `x`
- Mathematical notation: λ parameter . expression
  - $\lambda x.e$
  - x is called *bound variable*
  - e is the expression
- This is the core of Lambda Calculus!!!
  - Yes, but... What can it be used for?
  - Formal mathematical definitions for FP!

Applying Functions

- Avoid "useless" parentheses
- All functions have the same domain and codomain: set of λ-expressions
  - Functions apply to functions and return functions...
- Function application is left-associative
  - abc means $(ab)c$
  - Possible interpretation: "the a function is applied to b and c"...
    - Remember the currying thing?

Lambda Calculus: Formal Definitions

- Lambda Calculus expression (λ-expression): name, function or function application
  - Or a combination of the three...
- Function: λname.expression; Application: expression expression
- More formally, `e = x | λx.e | e e`
  - `x` is an identifier (variable, function, ...)
  - `e` is a generic λ-expression
- In practice, some parentheses can make things more readable:
  - `e = x | (λx.e) | (e e)`
  - Not really needed, but $(((f_1 f_2) f_3) f_4)$ is more understandable than $f_1 f_2 f_3 f_4$...

Lambda Calculus and Functional Programming

- Looking at the definition of λ-expressions, we can recognize abstractions ($\lambda x.e$) and applications ($e\ e$)
  - Abstractions: bind the x variable in e
    - Changing λx into λy and changing all the x of e into y, the meaning of e does not change!!!
    - Example in "standard" math: $f(x) = x^2$ is equivalent to $f(y) = y^2$
  - Applications: performed by substitution
- This recalls the reduction of functional programs!

Lambda Calculus and Functional Programming — 2

- Lambda Calculus: based on abstraction and application
- Same concepts used for executing/evaluating/reducing functional programs
- The Lambda Calculus is based on more formal definitions and can be the mathematical model for functional programming!

Variables: Free or Bound?

- Informally speaking, a variable x is *bound* by $\lambda x.$; a variable is free if it is not bound by any λ
- More formally... $F_v(e)$: set of free variables in e; $B_v(e)$: set of bound variables in e
  - If $e = x$, with x variable/identifier, $F_v(x) = \{x\}$ and $B_v(x) = \emptyset$
    - If an expression is composed of a single variable, such a variable is free!
  - $F_v(e_1 e_2) = F_v(e_1) \cup F_v(e_2)$ and $B_v(e_1 e_2) = B_v(e_1) \cup B_v(e_2)$
    - Function application does not “modify the state” (free or bound) of variables

Binding a Variable

- $F_v(\lambda x.e) = F_v(e) \setminus \{x\}$ and $B_v(\lambda x.e) = B_v(e) \cup \{x\}$
  - The λ operator (abstraction) binds a variable, removing it from the set of free variables and adding it to the set of bound variables
- Looks simple... No?

Substitution

- Based on the concept of free and bound variables, it is possible to formally define substitution:
  - $e[e'/x]$ (sometimes indicated as $e[x \rightarrow e']$): replace "x" with "e'" in expression "e"
  - This replacement is often indicated with "$\rightarrow$"
- Works on λ-expressions, which are defined by cases:
  - If x is an identifier, $x[e'/x] = e'$
  - If $x \neq y$, $y[e'/x] = y$
    - Replacing x with e' in "x", the result is e'
    - Replacing x with e' in "y", the expression does not change

Substitution - 2

- Let's see more complex cases... Application:
  - $(e_1 e_2)[e'/x] = (e_1[e'/x] e_2[e'/x])$
- In case of abstraction:
  - If $x \neq y$ and $y \notin F_v(e')$, $(\lambda y.e)[e'/x] = (\lambda y.e[e'/x])$
    - $y \notin F_v(e')$: avoids "capturing" y!!!
  - If $x = y$, $(\lambda y.e)[z/x] = (\lambda y.e)$
    - Replacing the variable bound by λ does not change the expression...

Capturing Free Variables: Example

- Consider $(\lambda x.\lambda y.xy)(yz)$: in $\lambda y.xy$, try to replace x with yz
  - $(\lambda y.xy)[yz/x]$
- If we simply applied $(\lambda y.e)[e'/x] \rightarrow \lambda y.(e[e'/x])$, we would get
  - $(\lambda y.xy)[yz/x] \rightarrow \lambda y.(xy[yz/x]) = \lambda y.yzy$
  - The y variable in yz has been "captured"...
  - See the problem, now?
- Solution: change $\lambda y.xy$ into $\lambda v.xv$
  - $(\lambda v.xv)[yz/x] \rightarrow \lambda v.(xv[yz/x]) = \lambda v.yzv$
  - This looks better...

Capturing a Free Variable

- If $x \neq y$ and $y \notin F_v(e')$, $(\lambda y.e)[e'/x] = (\lambda y.e[e'/x])$
  - $y \notin F_v(e')$: avoids "capturing" y!!!
  - What does this mean?
  - What happens if $y \in F_v(e')$?
- To avoid issues, rename the variable bound by λ!
  - The behaviour of a function must not depend on the formal parameter's name...
  - $\lambda x.x = \lambda y.y$ and so on... (in general: $\lambda x.e = \lambda y.(e[y/x])$
- So, rename to use a variable which is not free in e'!

Equivalence between Expressions

- When can we say that two expressions e_1 and e_2 are equivalent?
  - Intuitive answer: when the only differences are in the names of bound variables!
- If y is not used in e, $\lambda x.e \equiv \lambda y.e[y/x]$
  - λx becomes λy
  - All the occurrences of x in expression e are changed into y
- This is named Alpha Equivalence!!! $\equiv_\alpha$
- Two expressions are α-equivalent if one of the two can be obtained by replacing parts of the other one with α-equivalent parts

So, α, ... β!

- As we know, functional computation works by replacement/simplification/reduction...
- More formally, this is called β-reduction!!!
  - $(\lambda x.e)e' \rightarrow_\beta e[e'/x]$
- e_1 is β-reduced to e_2 if e_2 can be obtained from e_1 by β-reduction of some sub-expression
  - Note: $(\lambda x.e)e'$ is called redex!
  - And $e[e'/x]$ is its reduced form...
  - What to do when there are multiple redexes? It does not matter! (confluence theorem)

β Reduction

- β reduction: introduces a relation between λ-expressions
- It is not a symmetric relation: $e_1 \rightarrow_\beta e_2 \not\Rightarrow e_2 \rightarrow_\beta e_1$
  - So, it is not an equivalence relation...
  - ...But we can define a β-equivalence relation $\equiv_\beta$ (reflexive, symmetric, transitive closure of $\rightarrow_\beta$)
- Informally: $e_1 \equiv_\beta e_2$ means that there is a chain of β-reductions that somehow "links" e_1 and e_2
  - The "direction" of such β-reductions does not matter!

β Equivalence

- β-equivalece $\equiv_\beta$: defined based on β-reduction $\rightarrow_\beta$
  - Reflexive, symmetric, transitive closure of $\rightarrow_\beta$...
  - WTH does this mean???
- Extend $e_1 \rightarrow_\beta e_2$ to be reflexive ($e_1 \equiv_\beta e_2 \Rightarrow e_2 \equiv_\beta e_1$) and transitive ($e_1 \equiv_\beta e_2 \equiv_\beta e_3 \Rightarrow e_1 \equiv_\beta e_3$)
  - $e_1 \rightarrow_\beta e_2 \Rightarrow e_1 \equiv_\beta e_2$
  - $\forall e, e \equiv_\beta e$
  - $e_1 \equiv_\beta e_2 \Rightarrow e_2 \equiv_\beta e_1$
  - $e_1 \equiv_\beta e_2 \equiv_\beta e_3 \Rightarrow e_1 \equiv_\beta e_3$

Normal Forms

- Normal form: expression without any redex $\rightarrow$ cannot be β-reduced
  - $\lambda x.\lambda y.x$ is a normal form, $\lambda x.(\lambda y.y)x$ is not ($(\lambda y.y)x \rightarrow_\beta x$, so $\lambda x.(\lambda y.y)x \equiv_\beta \lambda x.x$)
- β-reductions can bring to a normal form...
- ...Or can continue forever!
  - $(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (xx)[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx)$...
- This is like endless recursion (or endless loops)...

Confluence Theorem

- Consider β-reductions of expressions with multiple redexes...

  "If e reduces to e_1 after some (β-)reduction steps and e reduces to e_2 after some (β-)reduction steps, then it exists an expression e_3 so that both e_1 and e_2 reduce to e_3 after some (β-)reduction steps"

- If e reduces to a normal form, then such a normal form does not depend on the reduction order

λ Calculus: What can it Do?

- λ calculus as just defined can look "not powerful enough"
  - Expressions are composed only by variables, abstractions and applications...
  - Something like $\lambda x.x + 2$ is not a valid λ-expression
    - 2 and $+$ are not variables
- However λ calculus is Turing complete!
  - Can code all the "useful" algorithms
  - So, it must allow to encode constants, mathematical operations, ...
    - How???

Example: Encoding Natural Numbers

- Encoding based on Peano's definition:
  - 0 is a natural number
  - If n is a natural number, then its next (succ(n)) is also a natural number
- Alonso Church did something similar...
  - 0 is encoded as $\lambda f.\lambda x.x$ (f applied 0 times to x)
  - succ(n): apply f to n
- in practice : 0 = function applied 0 times to a variable, 1 = function applied 1 time, ...
- n: function applied n times to a variable
- So, what's the formal definition of "succ()"?

Natural Numbers: Computing the Next — 1

- succ(n) = $\lambda n.\lambda f.\lambda x.f((nf)x)$
  - It should simply add an f to n...
- Informally, n is encoded as $\lambda f.\lambda x.$ followed by n times f and by x
  - "Body" of this function: $\overbrace{f(\ldots f(}^{n} x) \ldots)$
  - Must be "extracted" from n (removing $\lambda f.\lambda x.$), then an "f" can be added, and the expression can be abstracted again respect to f and x
- How can we do this, more formally?
  - Using abstractions and applications

Natural Numbers: Computing the Next — 2

- We saw how to increase a natural number (remove $\lambda f.\lambda x$, add an "f" on the left, add $\lambda f.\lambda x$ again...):
- Let's see how to do it in practice:
  - "Exctracting" the function body: apply n to f and then to $x \rightarrow ((nf)x)$
  - Add "f": easy... $\rightarrow f((nf)x)$
  - Abstract again: $\lambda f.\lambda x.f((nf)x)$
- All this depends on n: $\lambda n.\lambda f.\lambda x.f((nf)x)$

Encoding Natural Numbers - 1, 2, ...

- $1 =$ succ(0): $(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x)$
  - $(\lambda n.\lambda g.\lambda y.g((ng)y))(\lambda f.\lambda x.x)$
  - $\lambda g.\lambda y.g(((\lambda f.\lambda x.x)g)y)$
  - $\lambda g.\lambda y.g((\lambda x.x)y) = \lambda g.\lambda y.gy$
  - $\lambda g.\lambda y.gy = \lambda f.\lambda x.fx$
- $2 =$ succ(1): $(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.fx)$
  - $(\lambda n.\lambda g.\lambda y.g((ng)y))(\lambda f.\lambda x.fx)$
  - $\lambda g.\lambda y.g(((\lambda f.\lambda x.fx)g)y)$
  - $\lambda g.\lambda y.g((\lambda x.gx)y)$
  - $\lambda g.\lambda y.g(gy) = \lambda f.\lambda x.f(fx)$
- Similarly, $3 =$ succ(2) $= \lambda f.\lambda x.f(f(fx))$, etc...

Summing Natural Numbers

- As said, $n \equiv f$ applied n times to x
- So, $2 + 3 =$ "Apply 2 times f to 3"
  - Apply 2 times f to "apply 3 times f to x"...
- $n + m$: apply n times f to m
  - Extract the bodies of n and m
  - In n body, replace x with m
  - Abstract again respect to f and x
  - Abstract respect to m and n
- How to do this:
  - m body : $(mf)x$
  - n body with x replaced by m body: $(nf)((mf)x)$
  - So, $\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x)$

Example of Sum

- $2+3$: $\lambda f.\lambda x.f(fx)$ + $\lambda f.\lambda x.f(f(fx))$
  - +: $\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x)$
- $(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$
  - $(\lambda n.\lambda m.\lambda g.\lambda y.(ng)((mg)y))(\lambda h.\lambda z.h(hz))(\lambda f.\lambda x.f(f(fx)))$
  - $\lambda g.\lambda y.((\lambda h.\lambda z.h(hz))g)(((\lambda f.\lambda x.f(f(fx)))g)y)$
  - $\lambda g.\lambda y.(\lambda z.g(gz))((\lambda x.g(g(gx)))y)$
  - $\lambda g.\lambda y.(\lambda z.g(gz))(g(g(gy)))$
  - $\lambda g.\lambda y.(g(g(g(g(gy)))))$
- This is equal to $\lambda f.\lambda x.f(f(f(f(fx))))$
  - f applied 5 times to x: 5!
  - So, $2+3=5$...

Yes We Can

- Lambda calculus can encode everything needed to be Turing-complete (not only natural numbers and arithmetic operations)
  - Boolean, conditionals (`if ... then ... else`), ...
- However, some encodings are everything but simple!
  - $2 + 3 \equiv (\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$
- $\lambda x.x + 2$ is not a valid λ-expression...
  - But $\lambda x.((\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))x(\lambda f.\lambda x.f(fx))$ is!
  - And it has the same meaning...

A Possible Extension

- Going beyond “pure” lambda calculus, it is possible to use natural numbers, operators, conditionals, and so on...
  - All these things can be implemented using “pure” λ-expressions (only variables, abstractions and applications)
- Things like $\lambda x.(x + 2)$ or $\lambda x.$`if x = 1 then 0 else ...` become valid!
  - Symbols like 2, $+$, `if ...` are like macros, that can be replaced with the appropriate encoding...
- “Extended” λ calculus (can be reduced to pure λ calculus by... Replacement!)

Iteration and Recursion

- How to encode iteration in λ expressions?
  - Functional paradigm: use recursion!
  - So the question is: how to encode recursion???
- This would need to "name" $\lambda x. ...$
  - ...But this would require a non-local environment! λ calculus does not have it
- How to implement recursion using abstraction and application only?
- Let's try a stupid example:

```
int f(int n) {return n == 0 ? 0 : 1 + f(n - 1);}
```

  - Yes, this is really stupid... But is just an example
  - It implements the identity function

```
int f(int n) {return n;}
```

Recursion in λ Calculus: an Example

- $f = \lambda n.\texttt{if}\ n == 0\ \texttt{then 0 else 1 + } f\ \texttt{(n - 1)}$
- "$f =$" is not a definition, this is an equation...
    - $f = G(f)$... $G()$: higher-order function
        - Takes a function as an argument
        - Returns a function as a result
    - Solving the equation, we can find f... But, what does "$=$" mean?
- How can we solve this equation?
- First, define G by abstracting respect to f:
- $G = \lambda f.\lambda n.\texttt{if}\ n == 0\ \texttt{then 0 else 1 + } f\ \texttt{(n-1)}$
- So, we need to find $h : h \equiv_\beta Gh$
    - Applying G to h we obtain something equivalent to h, again (using β-equivalence!)

Recursion - Example Continued

- $f = \lambda n.\texttt{if}\ n == 0\ \texttt{then 0 else 1 +}\ f\ \texttt{(n-1)} \to \lambda f.\lambda n.\texttt{if}\ n == 0\ \texttt{then 0 else 1 +}\ f\ \texttt{(n-1)}$
  - See? The Recursion Disappeared!!!
  - The function to be invoked recursively is passed as a parameter!
- Example:

```
std::function<int(int)> f = [&f](int n){return n == 0 ? 1 : n * f(n - 1);};
```

$\Rightarrow$

```
auto g = [](std::function<int(int)> f, int n){return n==0 ? 1 : n*f(n-1);};
```

- We need `f1` such that `f1 = g f1`...
- Notice: `[&f]` is not needed, here

- Back to the problem: given a function G, find $f : f \equiv_\beta Gf$
  - Here, "$=$" after some β-reduction on left or right side... β-equivalence!
- This requires to find the *fixed point* (fixpoint) of G...
- How? Y combinator! $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
  - Uh??? And WTH is it??? Consider e and try to compute $Y e$...

Y!!!

- $Ye = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))e$
- $(\lambda x.e(xx))(\lambda x.e(xx)) =_\alpha (\lambda y.e(yy))(\lambda x.e(xx)) \rightarrow_\beta$
- $\rightarrow_\beta e(\lambda x.e(xx))(\lambda x.e(xx))$
- But $(\lambda x.e(xx))(\lambda x.e(xx))$ can be the result of a β-reduction...
  - $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ applied to e
- $e(\lambda x.e(xx))(\lambda x.e(xx)) \leftarrow_\beta$ $e(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))e) = e(Ye)$
  - Note: some of the steps did not happen by direct β-reduction! Hence, $Ye \equiv_\beta e(Ye)$
- $Ye \equiv_\beta e(Ye) \Rightarrow YG \equiv_\beta G(YG)$: interpreting "$\equiv_\beta$" as "$=$", YG is a fixed point for G!!!

Y... Combinator???

- Y Combinator: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- Combinator: λ-expression without free variables
  - $\lambda f.$ ...
  - It is a higher-order function: an argument (G) is a function and the result is a function
  - No free variables: all the symbols are bound through some λ
- Y is an expression $\lambda f.$ ... without free variables $\rightarrow$ it is a combinator!
- It is a special combinator: given a function f, it computes its fixed point (fixed point combinator)
  - Y is not the only fixed point combinator... Many other exist!
  - Y works with β-equivalence

Fixed Point Combinators

- Importance: allows to implement recursion in λ calculus
  - In a programming language, allows to implement recursion without naming a function
  - WTH???
- Y Combinator: works with evaluation by name
  - With evaluation by value (eager), infinite recursion...
- Other fixed point combinators can work with evaluation by value
  - Z Combinator: $\lambda f.((\lambda x.(f(\lambda y.(xx)y)))(\lambda x.(f(\lambda y.(xx)y))))$
  - H Combinator: $\lambda f.((\lambda x.xx)(\lambda x.(f(\lambda y.(xx)y))))$

Simplifying Even More

- λ calculus: only few features
  - Variables
  - Function application
  - Abstraction
- Are they all needed? Can we do without some of them?
  - They are all needed if there are not “predefined functions”
  - But if we provide some smart combinators...
  - ...Then we can work without abstractions!!!
- This looks funny... Let’s look at some more details!

Combinator Calculi

- Combinator: expression without free variables
- Combinator calculus: based only on variables, some pre-defined combinator, and function application!
  - Multiple different combinator calculi are possible
  - Depending on the pre-defined combinators
- Pre-defined combinators: calculus *basis*
- Appropriate basis: the calculus can be Turing-complete!!!
- How does an “appropriate basis” looks like?
  - SK (or SKI) calculus!

SK Calculus

- Two basic combinators: S and K
  - S: $Sxyz = xz(yz)$
  - K: $Kxy = x$
  - Sometimes, the *identity* combinator I is also considered... But $I = SKK$
- The resulting SK calculus is equivalent to the λ calculus
  - All possible λ-expressions can be encoded as SK expressions
- But it does not use abstractions!
- Used in some esoteric functional programming languages (unlambda, ...)

Lambda and Types

- λ calculus: very low-level programming language
- Expressions are basically untyped (everything is a function)
- Like Assembly (everything is a sequence of bits)
  - $\mathcal{E}$: set of λ-expressions
  - A function f is a λ-expression $\Rightarrow f \in \mathcal{E}$
  - All functions have the same domain and codomain $\mathcal{E} \Rightarrow \mathcal{E} \rightarrow \mathcal{E} \subset \mathcal{E}$
- This does not compromise the language expressivity... But can cause bugs!.
  - Example: $\lambda x.x + 2$ is not a function $\mathcal{N} \rightarrow \mathcal{N}$
  - Can be applied to every function, not only to encodings of natural numbers!

Specifying the Types of Functions

- We would like to enforce that $(\lambda a.a + 2) \in \mathcal{N} \to \mathcal{N}$...
- But $\lambda a.a + 2$ really means $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$...
- Specifying the type of this function is not easy at all!
- Alternative: let's specify the type of the bound variables
- Yes, but... What is a type?
    - First of all, we need to formally define types

Types

- $\mathcal{P}$: set of *base types* (or *primitive types*); $\mathcal{T}$: set of all possible types
- A primitive type is a type
  - $\alpha \in \mathcal{P} \Rightarrow \alpha \in \mathcal{T}$
- Functions from a type to another have a valid type
  - $\alpha, \beta \in \mathcal{T} \Rightarrow \alpha \to \beta \in \mathcal{T}$
- These types can be associated to λ-expressions
  - As usual, consider the three possible types of λ-expression: variable, application and abstraction
  - Variables: the type of a free variable must be known

Associating Types to Expressions

- If E_1 has type $\alpha \to \beta$, $E = E_1 E_2$ is valid only if E_2 has type α
  - As a result, E has type β
- If E has type β, then $\lambda x.E$ has type $\alpha \to \beta$
  - Moreover, x has type α
- For abstractions $\lambda x.E$, explicit typing can also be used: $\lambda x : \alpha.E$ means that x has type α
- Some λ-expressions cannot be correctly typed
  - What's the type of $\lambda x.xx$? If x has type α, then $\lambda x.xx$ has type $\alpha \to \beta$, where β is the type of xx
  - But, what's the type of xx? If x has type α, then xx has type β and x has type $\alpha \to \beta$???

The Effect of Types

- So, $\lambda x.xx$ does not type-check...
- It can be proved that the β-reduction of every correctly-typed λ-expression terminates in a finite number of steps
  - No divergent computations / infinite recursion?
  - The typed λ calculus is not Turing-complete!!!
- So, adding a feature (types) reduces the expressive power of the language... Funny!
- The Y combinator also contains an "xx", which does not type-check...
  - Typed λ calculus $\rightarrow$ no recursion???
  - A more complex type system is needed... (recursion in the type system!)

Fixed Point Combinators in a Programming Language

- Implementing the Y combinator is possible, but... Not always easy!
- A first issue is with eager evaluation...
  - In this case, a different fixed point combinator must be implemented
- Issues with strict type checking (Y does not type check!)
  - Recursive data types must be used to eliminate recursion from functions
- The details are not simple...