ADVANCED PROGRAMMING LANGUAGES

Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

The λ Calculus

The development of functional languages has been influenced from time to time by many sources, but none is as paramount nor as fundamental as the work of Church on the λ calculus.

The λ calculus is usually regarded as the first functional language. Although it was certainly not thought of as programming language at the time (there weren't even computers on which to run the programs) modern functional languages can be thought of as (nontrivial) embellishments of the λ calculus.

Church's work was motivated by the desire to create a calculus – a syntax for terms and set of rewrite rules for transforming terms – to capture the intuition about the behavior of functions.

Unlike other approaches, in which functions are characterized as sets of (*argument*, *value*) pairs , the intent was to capture their computational aspects.

The Pure Untyped λ Calculus

Syntax. The abstract syntax of the pure untyped λ calculus (a name chosen to distinguish it from other versions we will develop later) embodies what are called λ expressions (or terms), defined as follows:

$$\begin{array}{rcll} M, N, P & ::= & x, y, z & \text{variable} \\ & \mid & \lambda x.M & \text{abstraction} \\ & \mid & M\ N & \text{application} \end{array}$$

A λ abstraction $\lambda x.M$ is an anonymous function with formal parameter x. In Scala, this corresponds directly to the function literal `x => M`. We say that the function abstraction binds the variable x, so that the scope of this bounded variable is M, the body of the function.

Notice that x may have multiple occurrences in M and all of them are bound by the abstraction (unless captured by an inner abstraction: see later). On the other hand, the occurrences of a variable x not bound are *free*.

Free variables. The free variables of a term M are noted $\mathrm{fv}(M)$, and defined by the following rules, *inductively* on the syntax of expressions:

$$\begin{array}{rcl} \mathrm{fv}(x) & = & \{x\} \\ \mathrm{fv}(M\,N) & = & \mathrm{fv}(M) \cup \mathrm{fv}(N) \\ \mathrm{fv}(\lambda x.M) & = & \mathrm{fv}(M) - \{x\} \end{array}$$

We say that x is free in M iff $x \in \mathrm{fv}(M)$.

A few additional remarks on λ abstractions.

- Functions and their arguments are in the same syntactic category, which implies in particular that it is possible to use functions as function arguments (higher-order functions). That is, **functions are first-class values.**
- Another noteworthy attribute of the λ calculus is its restriction to functions of one argument. There is no real restriction in that, however, as a function of multiple argument, say `f = (x,y) => M` may be represented by the λ expression $f \equiv \lambda x.\lambda y.M.$ and applied as $(f\ N)\ P$ as opposed to $f(N, P)$.
  This notation has become known as ***currying***, and f is said to be a *curried* function. As we will see, the notion of currying has carried over today as a distinguishing syntactic feature of modern functional languages. Indeed, we have the exact same feature in Scala:

```
def add(a:Int,b:Int) = a + b          // add: (Int,Int) => Int
def addCurry(a:Int) = (b:Int) => a+b  // addCurry : Int => (Int => Int)

val x = add(3,4)                      // x : Int
val f = addCurry(3)                   // f : Int => Int
val r = addCurry(3)(4)                // r : Int
```

By convention, function application is expressed by juxtaposition as opposed to the Scala form `M(N)`, and is left associative, so that $M\ N\ P$ is the same as $(M\ N)\ P$. Hence, if in $M\ N\ P$ we want M to be applied to $N\ P$ we have to force it with explicit parentheses: $M(N\ P)$.

Function application is the core of computation in the pure λ calculus, and is formalized in terms of the notion of substitution.

Substitution. Substitution is an intuitively simple concept, but formalizing it is subtle, as we need to be careful about name clashes. The substitution of an expression M for all the free occurrences of a variable x in an expression N, written $[M/x]N$, is defined inductively as follows:

$$\begin{array}{llcll} \text{(s-var)} & [M/x]y & = & M & \text{if } x = y \\ & & \mid & y & \text{if } x \neq y \\ \text{(s-app)} & [M/x](N\,P) & = & ([M/x]N)([M/x]P) & \\ \text{(s-abs)} & [M/x](\lambda y.N) & = & \lambda y.N & \text{if } x = y \\ & & \mid & \lambda y.[M/x]N & \text{if } x \neq y \text{ and } y \notin \mathrm{fv}(M) \\ & & \mid & \lambda z.([M/x]([z/y]N) & \text{with } z \neq x, y \text{ and } z \notin \mathrm{fv}(M) \cup \mathrm{fv}(N) \end{array}$$

The cases (s-var) and (s-app) should be self explained. As to (s-abs) we distinguish three sub-cases:

- the first tells us that substitution only affects the free variables of a term: thus, the substitution $[M/y](\lambda y.N)$ has no effect, as y is not free in $(\lambda y.N)$.
- in the remaining two cases the side conditions are so defined as to make sure no free variable in M is *captured* by the binder of the λ abstraction. To guarantee that we further distinguish to sub-cases
  - if y is not free in M we can freely replace each free occurrence of x with M in the body of the abstraction, and be guaranteed against any undesired name clash
  - otherwise we choose another name for y (a *fresh* name that does not appear free in M and N) and rename the λ abstraction consistently to make sure we have no clashes (and fall back to the previous sub-case)

A few examples to illustrate the definition at work.

- $[M/x](\lambda x.x) = \lambda x.x$, while $\lambda x.([M/x]x) = \lambda x.M$.

- Now consider the substitution $[y/x]\lambda y.(x\,y)$: if we disregard the side conditions, the resulting term would be $\lambda y.(y\,y)$ where y has been $captured$ by the λ abstraction, changing (at least intuitively) the behavior of the term. The problem is resolved by first renaming the bound variable y, as in $\lambda z.(x\,z)$ and then proceeding with the substitution: $[y/x]\lambda z.(x\,z) = \lambda z.(y\,z)$. As we will see shortly, this renaming is sound, as terms that differ only for the renaming the bound variables (α-renaming) may be considered the same term.

- As a further example we look at the following, more complex case: $[y/x]((\lambda y.x)(\lambda x.x)x)$.

$$\begin{array}{rcll} [y/x]((\lambda y.x)(\lambda x.x)x) & = & [y/x](\lambda y.x)[y/x](\lambda x.x)x) & \text{by (s-app)} \\ & = & [y/x](\lambda z.x)[y/x](\lambda x.x)x) & \text{by renaming} \\ & = & (\lambda z.y)[y/x](\lambda x.x)x) & \text{by (s-abs)} \\ & = & (\lambda z.y)[y/x](\lambda x.x)([y/x]x) & \text{by (s-app)} \\ & = & (\lambda z.y)(\lambda x.x)([y/x]x) & \text{by (s-abs)} \\ & = & (\lambda z.y)(\lambda x.x)y & \text{by (s-abs)} \end{array}$$

To complete the λ calculus, we define three equality laws on λ expressions:

$$\begin{array}{lll} \alpha\text{−conversion (renaming)} & \lambda x.M & = & \lambda y.[y/x]M \\ \beta\text{−conversion (application)} & (\lambda x.M)N & = & [N/x]M \\ \eta\text{−conversion (x} \notin \text{fv(M))} & \lambda x.(M\,x) & = & M \end{array}$$

Each of these equations can be justified based on the reading of λ terms as functions and by observing that – interpreted as functions – the left-hand and right-hand sides of each equation denote the same function. In fact:

- α-conversion formalizes the idea that the name of the parameters are irrelevant to the behavior of the function
- β-conversion is just parameter passing: it says that applying the function of an argument does not change the value computed by the function
- η-conversion can be justified by noting that the “functional behavior” of $\lambda x.(M\,x)$ and of M is the same if $\mathrm{x} \notin \mathrm{fv(M)}$. Indeed, applying either of the two functions to any argument N results into the same term, namely $M\,N$.

Together with the standard equivalence rules for reflexivity, symmetricity, and transitivity, the three laws induce a theory of equality on the λ calculus, which can be shown to be consistent as a mathematical system.

In general we say that an equality theory over a domain is consistent if it does not equate values in the domain that are not equal. For instance, and equality theory over the domain of boolean expressions is consistent if it does not equate a true and a false expression. In the λ calculus, the notion of consistency is defined in terms of reduction and normal form, introduced below.

Evaluation as reduction

The computational meaning of an expression is defined in terms of rewriting, or reduction. Reduction is the same as convertibility, but restricted so that β-conversion and η-conversion and only happens in one direction:

- β-reduction $(\lambda x.M)N \rightarrow [N/x]M$
- η-reduction $\lambda x.(M\,x) \rightarrow M \quad x \notin fv(M)$

We write $M \rightarrow N$ if N can be derived by applying a β- or an η-reduction (possibly after an α-conversion) to M or any of M's subexpressions, and we take $\rightarrow^*$ to be the reflexive and transitive closure of $\rightarrow$. I

In fact, **we will mostly focus on β reduction and leave η reduction on the background**, as in the following definition.

Definition - Redex and Normal Form. A *redex* in an expression$M$ is a subexpression of M (possibly M itself) which can be reduced using β reduction. An expression is in *normal form* if it has no redexes.

Another way of interpreting the reduction relation is to think of the terms M and N in $M \rightarrow N$ as the states of an abstract machine which evaluates M by successive reductions until it reaches a normal form.

Interestingly, there exist λ expressions that have no normal form. The paradigmatic case is the following expression with one redex which reduces to itself (thus generating a nonterminating reduction process):

$$\Omega \equiv (\lambda x.(x\,x))(\lambda x.(x\,x))$$

Nevertheless, the normal form provides an attractive canonical form for a term: it has a clear sense of finality in a computational sense, and is what we intuitively think of as the value of an expression. Obviously, we would like for that value to be unique; and we would like to be able to find it whenever it exists. The Church-Rosser theorems give us positive results for both of these desires.

Church-Rosser Theorem I If $M = N$, then there exists an expression P such that $M \to^* P$ and $N \to *P$.

In other words, if $M = N$, then there exists a third term P (possibly the same as M or N) to which they can both be reduced.

Corollary No λ expression can be converted to two distinct normal forms (ignoring differences due to α -conversion).

Evaluation order

One consequence of the first Chuch-Rosser Theorem is that *how* we arrive at the normal form does not matter; in other words, the order of evaluation is irrelevant, a property that has important consequences for parallel evaluation strategies.

Then , the question arises as to whether or not it is always possible to find the normal form (assuming it exists). We begin with a definition.

Definition A *normal-order reduction* is a sequential reduction in which, whenever there is more than one redex, the leftmost, outermost one is chosen first. In contrast, an *applicative-order reduction* is a sequential reduction in which the leftmost, innermost redex is chosen first.

Church-Rosser Theorem II If $M \to^* N$ and N is in normal form, then there exists a normal-order reduction from M to N.

Thus, if a normal form exists, we can always find it using normal-order reduction. To see why applicative-order reduction is not always adequate, consider the following example:

- Applicative-order reduction

$$\begin{array}{rcl} (\lambda x.y)((\lambda x.xx)(\lambda x.xx)) & \to & (\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \\ & \to & (\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \\ & \to & \ldots \end{array}$$

- Normal order reduction

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \quad \to \quad y$$

We will return to normal- and applicative-order reduction and the trade-offs between them later on. For now we simply note that the strongest completeness and consistency results have been achieved with normal-order reduction.

Recursion in the λ Calculus

One of the cornerstones of the pure λ calculus is its untyped nature: every term denotes a function, and functions can be applied to any term. As a result, function may be applied even to themselves as exemplified in the Ω term!

In fact, self-application is at the core of the computational power of the λ calculus. We have already seen the power of this mechanism with the Ω term, capturing the simplest form of infinite computation. A further, more interesting case for self-application is the following term, known as the Y *combinator*:

$$Y \equiv \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

As we show below, the Y combinator provides the fundamental basis for recursion in the λ Calculus.

Fixpoint Theorem Every λ expression M has a fixpoint , i.e. a term N such that $N = (M\,N)$.

Proof. Take N to be (YM), where Y is the combinator defined as above. Then we have

$$\begin{array}{rcl} (YM) & = & \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))\ M \\ & = & (\lambda x.M(x\,x))(\lambda x.M(x\,x)) \\ & = & M((\lambda x.M(x\,x))(\lambda x.M(x\,x))) \\ & = & M\,(Y\,M) \end{array}$$

The theorem implies that any recursive function may be written non-recursively, without having to resort to the equational definitions we are used to. To see how, consider a recursive function f defined by:

$$f = F ... f ...$$

This equation essentially says that f is a fixpoint of the λ expression $\lambda f.F ... f ...$. But that is exactly what Y computes for us, so f may be defined directly using the Y combinator as follows: $\hat{f} \equiv Y(\lambda f.F ... f ...)$. In fact, from what we just showed:

$$\begin{array}{rcll} \hat{f} & \equiv & Y(\lambda f.F ... f ...) & \text{(by definition)} \\ & = & (\lambda f.F ... f ...)(Y(\lambda f.F ... f ...)) & \text{(by the fixpoint theorem)} \\ & = & (\lambda f.F ... f ...)(\hat{f}) & \text{(by definition)} \\ & = & F ... \hat{f} ... & \text{(by } \beta\text{-conversion)} \end{array}$$

As a concrete example, consider the factorial function:

```
val fact = n => if (n = 0) then 1 else (n * fact(n - 1))
```

Using the Y combinator can be written (in a suitable extension of the pure λ calculus to include numerals with their operations, and conditionals) non-recursively as follows:

$$\text{fact} = Y(\lambda f.\lambda n.\ \text{if}\ (n = 0)\ \text{then}\ 1\ \text{else}\ (n * f(n-1)))$$

The ability of the λ calculus to simulate recursion in this way is the key to its power and accounts for its persistence as a useful model of computation.

The Church-Turing Thesis on computable functions

The work on λ Calculus was part of a wider effort that in the 1930s was directed to formalize a notion of computation and of computable functions.

The first formalization was proposed in 1933 by Gödel and Herbrand, who defined the class of *general recursive functions* (on natural numbers) as the smallest class of functions (with arbitrarily many arguments) closed under *composition*, *recursion* and *minimization* and including the *zero* function, the *successor* operator and all *projections*.

Three years later, in 1936, based on an encoding of the natural numbers (known as Church numerals), Church proposed his own formalization of computable functions as the class of functions on natural numbers that can be represented as terms of the λ-calculus.

In 1936, a further independent model of computation was proposed by Turing based on his tape machines (known as Turing Machines) leading to the definition of the class of computable functions as the class of functions (on encoded natural numbers) computable by Turing machines.

The three notions were soon proved to be equivalent, with the results by Church, Kleene and Turing that showed that the three classes of computable functions were in fact the same class: a function is λ-computable if and only if it is Turing computable, and if and only if it is general recursive.

This quickly led to the widely agreed upon thesis that the concept of computability is accurately characterized by these three equivalent characterizations, a thesis that was later synthesized in what is known as the Church-Turing Thesis, stating that
the three formally-defined classes of computable functions coincide with the *informal* notion of an effectively calculable function.

Exercises

Exercise 1 (Variable Bindings). Fully parenthesize each of the following expression based on the standard parsing of λ-calculus expressions, i.e. you should parenthesize applications and λ abstractions.

Then, draw a box around all binding occurrences of variables, underline all usage occurrence of variables, and circle all free variables. For each bound usage occurrence, draw an arrow to indicate its corresponding binding occurrence.

- λa. z λz. a y
- (λz. z) λb. b λa. a a
- λa. λb.(λa. a) λb. a
- x λx. λx. x (λx. x)

Exercise 2 (Alpha equivalence). Which of these three lambda-calculus expressions are alpha equivalent?

- λx. y λa. a x
- λx. z λb. b x
- λa. y λb. b a

Exercise 3. (Reduction) Put the following expressions into normal form, using β-reduction (and α conversion as needed). Remember we're assuming left-association (hence, the λ term in the first item is the same as ((λz.z) (λq.q q)) (λs.s a)).

1. (λz.z) (λq.q q) (λs.s a) = ((λz.z) (λq.q q)) (λs.s a)
2. (λz.z) (λz.z z) (λz.z q)
3. (λs.λq.s q q) (λa.a) b
4. (λs.λq.s q q) (λq.q) q
5. ((λs.s s) (λq.q)) (λq.q)

Exercise 4. (Recursion). Let *Y* be the recursion operator defined earlier, and let F be a normal-form term.
- exaluate (*Y* F) using applicative order
- evaluate (*Y* F) using normal order

CREDITS

Most of the material from this lecture is taken from this paper by Prof. Paul Hudak.