International Journal of Advanced Innovation and Research
Vol. 3, Issue 06, June 2025
ISSN: 2584-2129

Behaviour of Recursive Function in Lambda Calculus

Dr. Rajesh Kumar
Department of Computer Science & Applications, Chhaju Ram Memorial Jat College, Hisar, Haryana, rajtaya@kuk.ac.

Keywords

Calulus, Theory, Programming Language

ABSTRACT

The aim of this paper is to link the λ-calculus with the today computational machine using the exploit mathematical properties of the calculus for the parallel evaluation of program. This paper is about the use of both the theory and practice of functional programming. The theory consists of the calculus and the practice will be illustrated using the programming language Standard ML. The practical parts of this paper almost exclusively emphasize the recursive functions but the material on the λ-calculus underlies both approaches.

Introduction

The recursive functions form an important class of numerical functions. Shortly after Church invented the lambda λ-calculus Kleene proved that every recursive function could be represented in it [1]. This provide evidence for Church thesis the hypothesis that any intuitively computable function could be represented in the calculus [2]. It has been shown that many other models of computation define the same class of functions that can be defined in the calculus [3].

In this section it is described what it means for an arithmetic function to be represented in the calculus two classes of functions the primitive recursive functions and the recursive functions are defined and it is shown that all the functions in these classes can be represented in the λ-calculus.

This section explains how a number n is represented by the λ-expression n. A λ-expression f is said to be represent a mathematical function f if for all number x1, xn :

f (x1, …. xn) = y iff f (x1, …. xn) = y (i)

Syntax And Semantics

The λ-calculus is a notation for defining functions. The expressions of the notation are called λ-expressions and each such expression denotes a function [4].

Using BNF, the syntax of λ-expressions is as under:

< λ-expression> ::= <variable>
| (< λ-expression> < λ-expression>)
| (λ<variable>. < λ-expression>)

If V has a range over the BNF syntax class < variable > and E, E1, E2, . . . etc. range over the BNF [5]

syntax class < λ-expression >, then the BNF simplifies as:

E ::= V | (E1 E2) | λ V[E] (ii)

where V represents variable
(E1 E2) is applications combinations
λ V[E] represents abstraction on scope of
λV
For example (λx. x) denotes the 'identity function' [5]:

$$(\lambda x.\ x) = (\lambda x\ [x]\ .\ E\ ) = E. \quad \text{(iii)}$$

Representation in Lambda-Calculus
The true and false λ-expressions denote the truth-value true and false, not represents the negation function and (E E1|E2) represents the conditional 'if E then E1 else E2'. There are countless different ways to denote the truth-value [6] and negation that work; the methods used here are traditional and have been developed by logicians over the years.

$$\text{true} \approx \lambda\ xy[x] \quad \text{(iv)}$$
$$\text{false} \approx \lambda\ xy[y] \quad \text{(v)}$$
$$\text{not} \approx \lambda\ t[(t\ \text{false}\ \text{true})] \quad \text{(vi)}$$

A function is called primitive function [7] if it can be constructed from 0 and the function S(successor function) and U (projection function) be a finite sequence of applications of the operations of substitution and primitive recursion. Let's consider the below given function which have the several arguments.

$$(\lambda<x1, \ldots, xn>[E]<E1, \ldots, En>) = \{E1/x1, \ldots, En/xn\}E \quad \text{(vii)}$$

Where {E1/x1, . . ., En/xn}E is called the simultaneous substitution. If this has the value {E1/x1}{ E1/x2}, . . ., {En/xn}E then this is called the sequential substitution.

Referential Transparency
An expression is referentially transparent [8] if it can be replaced by its value in a program without changing the behaviour of the program in any way.
In a pure functional language all functions are referentially transparent.

Hence
• programs are mathematically more tractable than imperative ones.
• compiler optimizations [9] such as common sub-expression elimination, code movement etc. can be incorporated without any static analysis.
• Any expression anywhere may be replaced by another expression which has the same value.
In most imperative languages (because of assignment, and side-effects [10] to non-local variables) there is no (guarantee of) referential transparency.

Higher-Order Function& Modularity
Higher Order: Higher order functions characterise most functional programming. It leads to compact and concise code [3, 11].
Modularity: Modularity can be built into a pure functional language
Objected-Orientedness: Object-oriented features require state updation and can be obtained only by destroying referential transparency. So a pure functional programming language cannot be object-oriented, though it can be modular [12].

Imperative Features
Input/Output: All input-output and file-handling (esp. in the Von Neumann framework [13]) is inherently imperative.
Object-Orientation: Object oriented features require updation of state and are hence better served by imperative features.
So most functional languages need to have certain imperative features [14, 15].

Algorithms and Findings

Let isqrt(n) of a non-negative integer n is the integer k ≥ 0 such that k2 ≤ n < (k + 1)2

That is,

isqrt(n)={ if & n<0@k &otherwise)┤ (viii) where

$0 \le k2 \le n < (k+1)2$ (ix) This value of k is unique!

$0 \le k2 \le n < (k+1)2$ (x)

$\square(\Rightarrow\top(\ )\ )\ 0 \le k \le \sqrt{n} < k+1$ (xi)

$\square(\Rightarrow\top(\ )\ )\ 0 \le k \le n$ (xii)

Use this fact to close in on the value of k. Start with the interval [l ,u]= [0,n] and try to shrink it till it collapses to the interval [k, k] which contains a single value.

If n = 0 then isqrt(n)= 0.

Otherwise with [l, u]= [0, n] and

$l2 \le n < u2$ (xiii)

use one or both of the following to shrink the interval [l, u].

if (l + 1)2 ≤n then try [l + 1,u]

otherwise l2 ≤n <(l + 1)2 and k= l

if u2 >n then try [l,u−1]

otherwise (u−1)2 ≤n <u2 and k= u−1

where:

isqrt(n)= { if n<0@0 if n=0@shrink(n,0,n) if n>0)┤ (xiv)

isqrt(n)= { if l=u@ shrink(n,l+1,u) if l<u and ⟦(l+1)⟧ ^2≤n@shrink(n,l,u-1) if l<u and u^2 >n)┤

In the above algorithm the function isqrt uses the function shrink which is recursively defined. Beginning with an initial closed interval [0,n], shrink reduces the size of the interval by 1 in each i recursive call. The complexity of the algorithm is therefore O(n) where n is the input to the function isqrt.

```
*Program for generating primes upto some number * fun primeWRT (m, [ ] ) = true
| primeWRT (m, h : : t ) = if m mod h= 0 then false else primeWRT (m, t )
fun generateFrom ( P , m, n ) = if m > n then P
else if primeWRT (m, P ) then (generateFrom ( (m:P), m+2 , n ))
else generateFrom ( P, m+2, n )
fun primesUpto n = if n < 2 then
[ ] else if n=2 then [ 2 ]
else if ( n mod 2= 0) then primesUpto ( n−1)
else generateFrom ( [ 2 ] , 3 , n ) ;
```

Curiously a systematic notation for functions is lacking in ordinary mathematics. The usual notation 'f(x)' does not distinguish between the function itself and the value of this function for an undetermined value of the argument.

Example 1. Let y= x2 be the squaring function on the reals. Here it is commonly understood that x is the "independent" variable and y is the "dependent" variable when we look on it as plotting the function f (x)= x2 on the x−y axis.

Example 2. Often a function may be named and written as
f (x)= xn to indicate that x is the independent variable and n is understood (somehow!) to be some constant. Here f , x and n are all names with different connotations. Similarly in the quadratic polynomial ax2 + bx + c it is somehow understood that a, b and c denote constants and that x is the independent variable. Implicitly by using the names like a, b and c we are endeavouring to convey the impression that we consider the class {ax2 + bx + c | a,b,c ∈R}of all quadratic polynomials of the given form.

These above examples should clearly convince the reader that there is a need to disambiguate between a function definition and its application.
The notation f (x), which is interpreted to refer to "the value of function f at x', will be replaced by f (x)
to denote an application of a function f to the (known or unknown) value x.
In other notation of the applied λ-calculus the functions and their applications in the examples would be rewritten as follows.

Example 1. Squaring. λx[x2] is the squaring function.
Example 2. q = λ a b c x[ax2 + bx + c] refers to any quadratic polynomial with coefficients unknown or symbolic. To obtain a particular member of this family such as 1x2 + 2x + 3, one would have to evaluate (((q 1) 2) 3) which would yield
λ x[1x2 + 2x + 3].

Conclusions
Since the λ-calculus only has variables and expressions and there is no place for names themselves (we use names such as K and S for our convenience in discourse, but the language itself allows only (untyped) variables and is meant to define functions anonymously as expressions in the language). In such a situation, recursion poses a problem in the language.
Recursion in most programming languages requires the use of an identifier which names an expression that contains a call to the very name of the function that it is supposed to define. This is at variance with the aim of the lambda calculus wherein the only names belong to variables and even functions may be defined anonymously as mere expressions.

References

1. Alves, Sandra, Maribel Fernández, Mário Florido, and Ian Mackie. 2011. "Linearity and Recursion in a Typed Lambda-Calculus." In Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, 173–82.
2. BIRD, R, and P WALDER. 1988. "Introduction to Functional Programming, Programming Research Group." Oxford University, Department of Computer Science, University of Glasgow.
3. Bruijn, Nicolaas G de. 1994. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." In Studies in Logic and the Foundations of Mathematics, 133:375–88. Elsevier.
4. Burge, William H. 1975. "Recursive Programming Techniques."
5. Burge, William H .1993. "Restricted Partition Pairs." Journal of Combinatorial Theory, Series A 63 (2): 210–22, 1993, https://doi.org/https://doi.org/10.1016/0097-3165(93)90057-F.
6. Darlington, John, Peter Henderson, and David A Turner. 1982. Functional Programming and Its Applications: An Advanced Course. CUP Archive.
7. Endrullis, Jörg, Dimitri Hendriks, and Jan Willem Klop. 2012. "Highlights in Infinitary Rewriting and Lambda Calculus." Theoretical Computer Science 464: 48–71.
8. Geuvers, Herman, and Rob Nederpelt. 2013. "NG de Bruijn's Contribution to the Formalization of Mathematics." Indagationes Mathematicae 24 (4): 1034–49.
9. Jones, Simon L Peyton, Graham Hutton, and Carsten Kehler Holst. 2013. Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming 13–15 August 1990, Ullapool, Scotland. Springer Science & Business Media.

10. McCarthy, Jay. 2014. Trends in Functional Programming. Springer.
11. Oostrom, Vincent van. 2024. “Confluence by the z-Property for de Bruijn’s λ -Calculus with Nameless Dummies, Based on PLFA.” In 13th International Workshop on Confluence, 56.
12. Tarau, Paul. 2015. “A Logic Programming Playground for Lambda Terms, Combinators, Types and Tree-Based Arithmetic Computations.” arXiv Preprint arXiv:1507.06944.
13. Tranquilli, Paolo. 2011. “Intuitionistic Differential Nets and Lambda-Calculus.” Theoretical Computer Science 412 (20): 1979–97.
14. Tromp, John. 2006. “Binary Lambda Calculus and Combinatory Logic.” Kolmogorov Complexity and Applications 6051.
15. Uustalu, Tarmo, and Varmo Vene. 2005. Advanced Functional Programming. Springer. Vial, Pierre. 2017. “Non-Idempotent Typing Operators, Beyond the Lambda-Calculus.” PhD