Recursive Functions of Symbolic Expressions and Their Application, Part I

JOHN MCCARTHY

Review: Amit Kirschenbaum

Seminar in Programming Languages

Historical Background

- **LISP** (**LIS**t **P**rocessor) is the second oldest programming language and is still in widespread use today.
- Defined by John McCarthy from M.I.T.
- Development began in the 1950s at IBM as FLPL - Fortran List Processing Language.
- Implementation developped for the IBM 704 computer by the A.I. group at M.I.T.

Historical Backgroung (Cont'd)

" The main requirement was a programming system for manipulating expressions representing formalized declerative and imperative sentences so that the Advice Taker could make deductions."

Many dialects have been developed from LISP: Franz Lisp, MacLisp, ZetaLisp . . .

Two important dialects

- Common Lisp - ANSI Standard
- Scheme - A simple and clean dialect. Will be used in our examples.

Imperative Programming

- Program relies on modfying a $state$, using a sequence of $commands$.
- State is mainly modified by $assignment$
- Commands can be executed one after another by writing them sequentially.
- Commands can be executed conditinonally using **if** and repeatedly using **while**
- Program is a series of instructions on how to modify the state.

Imperative Prog. (Cont'd)

- Execution of program can be considered, abstractly as:

$$s_0 \longrightarrow s_1 \cdots \longrightarrow s_n$$

- Program starts at state s_0 including inputs
- Program passes through a finite sequence of state changes,by the commands, to get from s_0 to s_n
- Program finishes in s_n containing the outputs.

Functional Programming

A functional program is an *expression*, and executing a program means *evaluating* the expression.

- There is no state, meaning there are no variables.
- No assignments, since there is nothing to assign to.
- No sequencing.
- No repetition but recursive functions instead.
- Functions can be used more flexibly.

Why use it?

- At first glance, a language without variables, assignments and sequencing seems very impractical
- Imperative languages have been developed as an abstraction of hardware from machine-code to assembler to FORTRAN and so on.
- Maybe a different approach is needed i.e, from human side. Perhaps functional languages are more suitable to people.

Advantages of functional programming

- Clearer sematics. Programs correspond more directly to mathematical objects.
- More freedom in implementation e.g, parallel programs come for free.
- The flexible use of functions we gain elegance and better modularity of programs.

Some Mathematical Concepts

- Partial Function - function that is defined only of part of its domain.
- Propositonal Expressions and Predicates - Expressions whose possible values are T (truth) and F (false).
- Conditional Expressions - Expressing the dependence of quantties on propositional quantities. Have the form

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

  Equivalent to
  “If p_1 then e_1, else if p_2 then e_2, $\cdots$ else if p_n then e_n”

Mathematical Concepts (Cont'd)

- Conditional expression can define **noncommutative** propositional connectives:

$$p \wedge q = (p \to q, T \to F)$$

$$p \vee q = (p \to T, T \to q)$$

$$\neg p = (p \to F, T \to T)$$

- Recursive function definitions - Using conditional expressions, we can define recursive functions

$$n! = (n = 0 \to 1, T \to n \cdot (n-1)!)$$

- Functions are defined and used, using λ-notation.

Brief intro to λ-calculus

- A formal system designed to investigate
  - function definition
  - function application
  - recursion
- Can be called the $smallest\ universal\ programming\ language$.
- It is universal in the sense that any computable function can be expressed within this formalism.
- Thus, it is equivalent in expressive power to Turing machines.
- λ-calculus was developed by $Alonzo\ Church$ in the 1930s

λ-notation

Defining a function in mathematics means:

- Giving it a name.
- The value of the function is an expression in the formal arguments of the function.
- e.g., $f(x) = x + 1$

Using λ-notation we express it as a λ-expression

- $\lambda x \,.\, (+\ x\ 1)$
- It has no name.
- prefix notation is used.

λ-notation (Cont'd)

- The function f may be applied to the argument 1 :

$$f(1)$$

- Similarly, the λ-expression may be applied to the argument 1

$$(\lambda x\ .\ (+\ x\ 1))1$$

- Application here means
  - Subtitue 1 for x: $(+\ x\ 1) \Rightarrow (+\ 1\ 1)$
  - Evaluate the function body: make the addition operation .
  - Return the result: 2

Syntax of λ-calculus

Pure λ-calculus contains just three kinds of expressions

- variables (identifiers)
- function applications
- λ-abstractions (function defintions)

It is convinient to add

- predefined constants (e.g., numbers) and operations (e.g., arithmetic operators)

$$
\begin{array}{lll}
\langle exp \rangle & ::= & \\
 & & \text{var} \\
 & | & \text{const} \\
 & | & (\langle exp \rangle\ \langle exp \rangle) \\
 & | & (\lambda\ \text{var}\ .\ \langle exp \rangle)
\end{array}
$$

Function Application

- Application is of the form $(E_1\ E_2)$
- E_1 is expected to be evaluated to a function.
- The function may be either a predefined one or one defined by a λ-abstraction.

λ-abstractions

- The expression

$$(\lambda x \,.\, (*\, x\; 2))$$

  is the function of x which multiplies x by 2
- The part of the expression that occurs after λx is called the $body$ of the expression.
- When application of λ-abstraction occurs, we return the result of the body evaluation.
- The body can be any λ-expression, therefore it may be a λ-abstraction.
- The parameter of λ-abstraction can be a function itself

λ-abstractions

- In mathematics there are also functions which return functions as values and have function arguments.
- Usually they are called $operators$ or $functionals$
- For example: the differentiation operator

$$\frac{d}{dx}x^2 = 2x$$

.

Constants

- Pure λ-calculus doesn't have any constants like $0, 1, 2, \ldots$ or built in functions like $+, -, *, \ldots$, since they can be defined by λ-expressions.
- For the purpose of this discussion we'll assume we have them.

Naming Expressions

- Expressions can be given names, for later reference:

$$\text{square} \equiv (\lambda x\ .\ (*\ x\ x))$$

Free and bound variables

- Consider the expression

$$(\lambda x\,.\,(*\;x\;y))2$$

- x is bound: it is just the formal parameter of the function.
- y is free: we have to know its value in advance.

- A variable v is called $bound$ in an expression E if there is some use of v in E that is bound by a decleration λv of v in E.
- A variable v is called $free$ in an expression E if there is some use of v in E that is not bound by any decleration λv of v in E.

Reduction rule

- The main rule for simplifiying expressions in λ-calculus is called β-reduction.
- Applying a λ-abstraction to an argument is an instance of its $body$ in which $free$ occurences of the formal parameter are substituted by the argument.
- parameter may occur multiple times in the body

$$(\lambda x\ .\ (*\ x\ x))4 \longrightarrow (*\ 4\ 4) \longrightarrow 16$$

Reduction rule

- Functions may be arguments

$$(\lambda f\,.(f\ 3))(\lambda x\,.(-\ x\ 1))$$

$$\begin{aligned}
&(\lambda f\,.(f\ 3))(\lambda x\,.(-\ x\ 1))\\
&\rightarrow (\lambda x\,.(-\ x\ 1))3\\
&\rightarrow (-\ 3\ 1)\\
&\rightarrow 2
\end{aligned}$$

Expressions for Recursive Functions

- The λ-notation is inadequte for defining functions recursively
- the function

$$n! = (n = 0 \rightarrow 1, T \rightarrow n \cdot (n-1)!)$$

should be converted into

$$! = \lambda((n)(n = 0 \rightarrow 1, T \rightarrow n \cdot (n-1)!))$$

- There is no clear reference from '$!$' inside the λ-clause, to the expression as a whole.

Expressions for Recursive Functions

- A new notation: $label(a, \mathcal{E})$ denotes the expression $\mathcal{E}$, provided that occurences of a within $\mathcal{E}$are to be referred as a whole.
- For example, for the latter function the notation would be

$$label(!, \lambda((n)(n = 0 \rightarrow 1, T \rightarrow n \cdot (n-1)!)$$

- (There is a way to describe recursion in λ-calculus, using Y-combinator, but McCarthy doesn't use it.)

S-Expressions

A new class of **S**ymbolic expressions.
S-Expression are composed of the special characters

- (- start of composed expression
-) - end of composed expression
- $\bullet$ - composition
- and "an infinite set of distinguishable **atomic symbols**".
  e.g.,

A

ABA

APPLE-PIE-NUMBER-3

S-Expression : Definition

- Atomic symbols are S-expression.
- if e_1 and e_2 are S-expressions then so is $(e_1 \cdot e_2)$
- examples

AB

(A· B)

((AB· C) · D)

- S-expression is then simply an ordered pair.

S-Expression : Lists

- The list

$$(m_1, m_2, \ldots, m_n)$$

is represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))$$

- NIL is an atomic symbol, used to terminate lists, also known as the $empty\ list$.
- (m) stands for $(m \cdot NIL)$
- $(m_1, m_2, \ldots, m_n \cdot x)$ stands for $(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot x) \cdots)))$

M-expressions

- **M**eta-expressions are functions of S-expressions, also called S-functions.
- Written in conventional functional notation.
- There are some elementry S-functions and predicates

M-expressions

- **atom** - atom[x] has the value T or F according to whether x is atomic symbol.
  - atom[X] = T.
  - atom[(X·A)] = F.
- **eq** - eq[x;y] is defined iff both x and y are symbols. eq[x;y] = T if x and y are the same symbol and eq[x;y] = F otherwise
  - eq[X;X] = T.
  - eq[X;A] = F.
  - eq[X;(A · B)] is undefined

M-expressions

- **car** - car[x] is defined iff x is not atomic.
  car[$(e_1 \cdot e_2)$]=e_1
  - car[(X·A)] = X.
  - car[(X·A)· Y)]= (X·A).
- **cdr** - cdr[x] is also defined iff x is not atomic.
  cdr[$(e_1 \cdot e_2)$]=e_2
  - cdr[(X·A)] = A.
  - cdr[(X·A)· Y)]= Y.
- **cons** - cons[x;y] is defined for any x and y. It is the list constructor
  cons[$(e_1$;$e_2)$]=$(e_1 \cdot e_2)$
  - cons[X;A] = (X·A).
  - cons[(X·A);Y]= ((X·A)· Y).

M-expressions

- Compositions of car and cdr arise very frequently.
- Many expressions can be written more concisely if we abbreviate.
- cadr[x] $\equiv$ car[cdr[x]]
- caddr[x] $\equiv$ car[cdr[cdr[x]]]
- cdadr[x] $\equiv$ cdr[car[cdr[x]]]
- expressions are not defined for every x. depends on the list structure.

Recursive S-functions

- Forming new functions of S-expression by conditional expression and recursive definition gives us much larger class of functions.
- In fact all computable functions.

Recursive S-function examples

- ff[x] - returns the first atomic symbol of the S-expression x, ignoring the parentheses.
- ff[x] = [atom[x]$\rightarrow$x ; $T \rightarrow$ ff[car[x]]]

ff[(A·B)]
= [atom[(A·B)]$\rightarrow$ (A·B) ; $T \rightarrow$ ff[car[(A·B)]]]
= [$F \rightarrow$(A·B);$T \rightarrow$ ff[car[(A·B)]]]
= ff[car[(A·B)]]
= ff[A]
= ff[atom[A]$\rightarrow$A ; $T \rightarrow$ff[car[A]]]
= [$T \rightarrow$A ; $T \rightarrow$ff[car[A]]]
= A

Transform M-expressions to S-expressions

There is a transformation mechanism that translate an M-expression $\mathcal{E}$ into S-expression $\mathcal{E}^*$

- if $\mathcal{E}$ is an S-expression, $\mathcal{E}^*$ is (QUOTE $\mathcal{E}$).
- M-expression $f[e_1; \ldots; e_n]$ is translated to $(f^*\ e_1^* \ldots e_n^*)$. Thus, $\{cons[A; B]\}^*$ is (CONS (QUOTE A) (QUOTE B))
- $\{[p_1 \rightarrow e_1]; \ldots; [p_n \rightarrow e_n]\}^*$ is (COND $(p_1^*\ e_1^*) \ldots (p_n^*\ e_n^*))$
- $\{\lambda[x_1; \ldots; x_n]\mathcal{E}\}^*$ is (LAMBDA$(x_1\ \ldots\ x_n)\ \mathcal{E}^*$.
- $\{label[a; \mathcal{E}]\}^*$ is (LABEL a $\mathcal{E}^*$)

What do we gain?

- Unifying Symbol-level and Meta-level, gives us a way to treat expressions over symbols exactly the same as symbols.
- Functions and data are **the same**.
- Thus we can write a program, which write another program and evaluating it.
- This is useful in AI.
- Furthermore, we can expand the language with new features.
- LISP interpreters are easily implemented in LISP.

S-function $apply$

- "Plays the theoretical role of a universal Turing machine and the practical role of an interpreter".
- Formally,
  - If f is an S-expression for an S-function f'
  - and $args$ is a list of arguments of the form $(arg_1, \ldots, arg_n)$ where $arg_1, \ldots, arg_n$ are S-expressions,
  - Then $apply[f; args]$ and $f'[arg_1, \ldots, arg_n]$ are defined for the same values of $arg_1, \ldots, arg_n$ and are equal when defined.
- example: $\lambda[[x; y]; cons[car[x]; y]]\ [(A, B); (C, D)] \equiv apply[(\text{LAMBDA}, (X, Y)(\text{CONS}(\text{CAR}X)Y))((A\ B)(C\ D))] = (A\ C\ D)$

S-function *eval*

- serves both as a formal definition of the language and as an interpreter
- Before *apply* applies the function f on the list of arguments $(arg_1, \ldots, arg_n)$, it sends them to *eval* for evaluating the S-expressions which represents them.

```
> (eval '(lambda (x) (+ x 1)))
#<procedure>
```

```
'(lambda (x) (+ x 1)))
```

is an S-expression which repersents a function. *eval* evalutes it and return its value, which is indeed a function

Implementing *eval*

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                          env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
          (apply (eval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Strength of the mechanism

- Extending the language is done easily by adding required forms to $eval$.
- Just add syntax and evaluation rules.
- Paraphrasing Oscar Wilde: LISP programmers know the value of everything but the cost of nothing.

The cost

- Performance of LISP systems became a growing issue
  - Garbage Collection.
  - Representation of internal structures.
- Became difficult to run on the memory-limited hardware of that time.

LISP Machines

- The solution was LISP machine - a computer which has been optimized to run LISP efficiently and provide a good environment for programming in it.
- Typical optimizations to LISP machines
  - Fast function calls.
  - Efficient representation of lists.
  - Hardware garbage collection.

LISP in the real world

- de-facto standard in AI
- NLP
- Modelling speech and vision

Some more

- AutoCAD
- Yahoo Store
- Emacs
- Mirai, the 3d animation package was used to create Gollum in Lord Of The Rings.

The End

```
((lambda(x)(x x))
   (lambda(x)(x x)))
```