Defining Symbolic Expressions

Introduction

You have been told that the Lambda Calculus is computationally universal, capable of expressing any algorithm. However, this most likely seems intangible. We will begin to define a layer of abstraction over the Lambda Calculus which makes design of programs begin to seem conceivable.

Our layer of abstraction will be a uniform language of Symbolic Expressions which is a dialect of the language called Lisp. These symbolic expressions are parentheses enclosed arrays of symbols, taking on different meanings based on their matching of patterns which we will define.

Here's an example:

$$(+\ 2\ 3)$$

The above evaluates to 5. In this case our expression is a function application receiving two numbers as arguments. This syntax is very simple, uniform and legible. Additionally, as you will see later on in this book, it is very easily interpreted by program.

Symbolic Expressions

The language into which we are entering is one of symbolic expressions. All of our expressions will take the form defined by the following grammar. This uniformity will make its definition in terms of Lambda Calculus far easier, and simplify its later interpretation or compilation.

A Symbolic Expression

$$\begin{aligned} \text{<expr>} &::= \text{<sexpr>} \mid \text{<atom>} \\ \text{<sexpr>} &::= (\text{<seq>}) \\ \text{<seq>} &::= \text{<dotted>} \mid \text{<list>} \\ \text{<dotted>} &::= \text{<expr>} \mid \text{<expr>}.\ \text{<expr>} \\ \text{<list>} &::= \text{<expr>} \mid \text{<expr>}\ \text{<exp>} \end{aligned}$$

A Symbolic Language

Primitive Forms

Now, these Symbolic Expressions or *S-Expressions* can take any of a multitude of forms. Of these, we will define meaning for forms of interesting patterns. We begin, unsurprisingly, with an S-Expression which serves to create lambdas. All forms matching the patterns which we discuss will be converted to the provided form, labeled as the consequent.

$$\begin{aligned} (\text{lambda}\ (var)\ expr) &\implies \lambda var\ expr \\ (\text{lambda}\ (var\ rest\ldots)\ expr) &\implies \lambda var\ (\text{lambda}\ rest\ expr) \end{aligned}$$

Essentially, we are saying that any expression of the form `(lambda args expr)` should be a function of the provided arguments bearing the provided expression.

Additionally, we provide a default case for our S-Expressions. Should no other mentioned pattern be a match to a given expression, we will default to function invocation. In other words, the following is a pattern we will match, with `fn` being some foreign form not selected for elsewhere.

$$\begin{gathered} (fn\ val) = (fn)val \\ (fn\ val\ rest\ldots) = ((fn)val\ rest) \end{gathered}$$

The Lambda Calculus has now been fully implemented in our symbolic forms; however, we will add many more features for the sake of convenience. After all, our goal was to add abstraction, not move a few symbols around!

Evaluation of Symbolic Forms

Before we continue, we'll look at some examples of our syntax as implemented so far. In evaluating a Lambda Calculus expression, or in this case, derived form, the single operation necessary is known as reduction. Reduction is conversion of an expression of function application to a new expression, one derived by substitution of the argument value.

To begin gaining familiarity with our language, we look at a function of two variables. The function that follows performs `f` on a value `x`, and then `f` once more on the resultant value.

$$(\text{lambda}\ (f\ x)\ (f\ (f\ x)))$$

Now we will look at a similar form, a function of three variables. Below is a function of values `g`, `f`, and `x`. The result is similar to that of the one above, but this time replacing `(f x)` with `(g f x)`.

$$(\text{lambda}\ (g\ f\ x)\ (f\ (g\ f\ x)))$$

So far we have covered some examples of Symbolic Expressions, but all of them have been of the lambda definition form. Furthermore, they have lacked any concrete meaning. To explore the additional notation we have defined, we will apply the former expression to the latter, which expands into the following.

$$((\text{lambda}\ (g\ f\ x)\ (f\ (g\ f\ x)))\quad (\text{lambda}\ (f\ x)\ (f\ (f\ x))))$$

This expression appears quite complex, so let's use the aforementioned reduction operation to simplify it. Recall from our definition of `lambda` that a function of multiple variables evaluated for one results in a function of one-less variable than the initial form. Hence we begin by substituting our argument for `g` throughout the expression; later steps are of a similar nature.

$$\begin{aligned} &((\text{lambda}\ (g\ f\ x)\ (f\ (g\ f\ x)))\quad (\text{lambda}\ (f\ x)\ (f\ (f\ x)))) \\ \Longrightarrow\ &(\text{lambda}\ (f\ x)\ (f\ ((\text{lambda}\ (f\ x)\ (f\ (f\ x)))\ f\ x))) \\ \Longrightarrow\ &(\text{lambda}\ (f\ x)\ (f\ (f\ (f\ x)))) \end{aligned}$$

That looks much better! What we have arrived at is only slightly different than our initial function of `f` and `x`. The only change was the number of times that `f` was applied. Hopefully these examples have given you a feel for how this syntax can work, and maybe even an early sense of how useful functions will emerge from the Lambda Calculus.

Foundations in Lambda Calculus

To accompany our syntactic constructs, we will need to define some forms in the Lambda Calculus, especially data-types and their manipulations. Our definitions will be illustrated as equalities, like $id = \lambda xx$; however, syntactic patterns will be expressed as implications. Recall that in Lambda Calculus there are only functions, no literals or primitive data-types. To combat this apparent shortcoming of the language, we will need to give data-types of interest a functional form. The examples of the prior section were a preview into how our conceptualization of numbers will behave.

Numbers

Numbers are a rather fundamental data-type, especially in modern computing. Additionally, they are in most other languages seen as atomic and primitive. However, we must provide a definition for the behavior of numbers in our lanuguage constituent of functions. We begin with a means of defining all natural numbers inductively, via the successor.

$$\begin{aligned} &\text{Numbers} \\ 0 &= \lambda f \lambda x x \\ succ &= \lambda n \lambda f \lambda x (f)((n) f) x \end{aligned}$$

Our definition of numbers is just like the examples from the previous section. Notice that `1`, for example, could be easily defined as $1 = (succ\ 0)$, as could any positive integer with enough applications of `succ`. Later on when we return to syntactic features we will define all numbers in this way; the numbers will take on their usual form as a string of decimal digits.

Numbers are our first data-type. Their definition is iterative in nature, with zero meaning no applications of the function `f` to `x`. We now will define some elementary manipulations of this data-type, i.e., basic arithmetic. The following definitions are pretty straightforward; nearly all of them consist exclusively of iterative application of a more primitive function to a base value.

$$\text{Arithmetic}$$
$$\begin{aligned} + &= \lambda n \lambda m((n)succ)m \\ * &= \lambda n \lambda m((n)(sum)m)0 \\ pred &= \lambda n \lambda f \lambda z((((n)\lambda g \lambda h(h)(g)f)\lambda uz)\lambda uu) \\ - &= \lambda n \lambda m((m)pred)n \end{aligned}$$

Addition merely takes advantage of the iterative nature of our numbers to apply the successor `n` times, starting with `m`. In a similar manner, multiplication applies addition repeatedly starting with zero. The predecessor is much more complicated, so let's work our way through its evaluation.

We'll begin our exploration of the `pred` function by looking at the value of two. Since two equals `(succ)(succ)0` we can work out its Lambda form, or simply take as a given that is the following.

$$2 = \lambda f \lambda x(f)(f)x$$

Now we can evaluate `pred` for this value. `pred` has been defined already, but let's briefly render it in the following more succinct form. The form below is very easily translated back to Lambda Calculus and should serve to cut through at least a portion of the complexity of the definition.

$$pred = \lambda n \lambda f \lambda z((\lambda g \lambda h(h)(g)f)^n\ \lambda uz)\ \lambda uu$$

With the above rendering of the definition in mind, we aim to reduce an application of `pred` to `2` to a result.

$$(\lambda n \lambda f \lambda z((\lambda g \lambda h(h)(g)f)^n\ \lambda uz)\ \lambda uu)\ 2$$

$$(\lambda f \lambda z((\lambda g \lambda h(h)(g)f)^2\ \lambda uz)\ \lambda uu)$$

Now that we have reduced the expression to the form of our prior rendering of `pred`, we will expand it into a true Lambda Calculus form and continue our reduction.

$$(\lambda f \lambda z(((\lambda g \lambda h(h)(g)f)\ (\lambda g \lambda h(h)(g)f))\ \lambda uz)\ \lambda uu)$$

In the following conversions, as in all, our reductions will need to take place in a right-to-left direction when evaluating expressions of the form `(f)(g)x`. Recall that our goal here is to reduce a complex form to simplistic result, and we have already made significant progress.

$$(\lambda f \lambda z((\lambda g \lambda h(h)(g)f)\ (\lambda h(h)(\lambda uz)f))\ \lambda uu)$$

$$(\lambda f \lambda z((\lambda g \lambda h(h)(g)f)\ (\lambda h(h)z))\ \lambda uu)$$

$$(\lambda f \lambda z(\lambda h(h)(\lambda h(h)z)f)\ \lambda uu)$$

$$(\lambda f \lambda z(\lambda h(h)(f)z)\ \lambda uu)$$

The above were all mere substitutions, as should be expected. If any were unclear, try working those steps out in a notebook. We are now finally ready to reduce the application of the identity (λuu) and achieve our final result.

$$\lambda f \lambda z(\lambda uu)(f)z$$

$$\lambda f \lambda z(f)z$$

Our result was a single application of `f` to `z`, i.e., one. Hence you have seen that at least in this case, the `pred` function did its job. Achieving an intuitive grasp of how it works is unfortunately not as straight-forward. If you wish to, keep in mind that λuz maps a value to the numeric starting point, and λuu leaves an expression alone. So the decrement occurs by the setting of the origin later than it would normally occur.

With our complex definition of the predecessor complete, subtraction is trivial. Once again we perform an iterative process on a base value, this time that process is `pred`.

Booleans

Having defined numbers and their manipulations, we will work on booleans. Booleans are the values of true and false, or in our syntax, `#t` and `#f`. Booleans are quite necessary in expressing conditional statements; thus the concomitant `if` function. These values will give us great power in their ability to branch results to a function, in a sense constructing piece-wise functions. It is by this ability that we are able to form a multitude of inductive definitions, as well as other important forms.

Booleans

$$\begin{aligned} \#\mathrm{t} &= \lambda a \lambda b(a)id \\ \#\mathrm{f} &= \lambda a \lambda b(b)id \\ if &= \lambda p \lambda t \lambda f((p)\lambda\_t)\lambda\_f \end{aligned}$$

The key to our booleans is that they accept two functions as parameters, functions that serve to encapsulate values, of which one will be chosen. Once chosen, that function is executed with the arbitrarily-chosen identity as an argument. This method of wrapping the decision serves as a means of lazy evaluation, and is fully realized in the lambda-underscores wrapping the branches of an `if` statement.

Now, since of course no boolean system is complete without some boolean algebra, we define `and` and `or`. These functions perform the operations you would expect; `(and a b)` is true only when both `a` and `b` is true, but `(or a b)` is true if either argument is true. Their definitions follow easily from our `if` function. Keep in mind that both of these functions operate only on booleans.

Boolean Algebra

$$\begin{aligned} and &= \lambda a \lambda b(((if)a)b)\#\mathrm{f} \\ or &= \lambda a \lambda b(((if)a)\#\mathrm{t})b \end{aligned}$$

With boolean manipulation and conditionals in hand, we need some useful predicates to utilize them. We define some basic predicates on numbers with the following. `eq` will be very useful in later developments; it is one of McCarthy's elementary functions.

Numerical Predicates

$$\begin{aligned} zero? &= \lambda n((n)\lambda x\#\mathrm{f})\#\mathrm{t} \\ leq &= \lambda a \lambda b(zero?)((-)m)n \\ eq &= \lambda a \lambda b(and\ (leq\ a\ b)(leq\ b\ a)) \end{aligned}$$

The above predicates serve to identify traits of a given number or given numbers. `zero?` is true when a number is zero, `leq` is true when the first number is less than or equal to the second, and `eq` determines whether two numbers are equal.

Pairs

Finally we reach the most important part of our S-Expressions, their underlying lists. That is to say, every Symbolic Expression is innately a list of other expressions, whether atomic or symbolic, and these lists serve as an analog to that data-type. To construct lists we will opt for a sort of linked-list implementation in our lambda definitions. We begin with a pair and a `nil` definition, each readily revealing their type by opting for either the passed `c` or `n` function.

Pairs

$$\begin{aligned} cons &= \lambda a \lambda b \lambda c \lambda n((c)a)b \\ nil &= \lambda c \lambda n(n)id \end{aligned}$$

`cons` constructs a pair when given two values, and accepts a function which will receive the two items to manipulate. `nil` on the other hand serves as a sort of empty pair, and instead fires the second provided function to identify itself as such.

Now, once again we follow a defined data-type with its manipulations. Just as did McCarthy, we will provide `car` and `cdr` as additional elementary functions, with `pair?` and `null?` serving as complements to each other in determining the end of a list. `car` return the first value of a pair, and `cdr` the second. Their names are quite historical and refer to address access of a pair in memory, but you can just think of them as $/k\alpha r/$ and $/k\mho der/$.

Pair Operations

$$\begin{aligned} car &= \lambda l(((l)\lambda a\lambda ba)id) \\ cdr &= \lambda l(((l)\lambda a\lambda bb)id) \\ pair? &= \lambda l(((l)\lambda\_\lambda\_\#\mathrm{t})\lambda\_\#\mathrm{f}) \\ null? &= \lambda l(((l)\lambda\_\lambda\_\#\mathrm{f})\lambda\_\#\mathrm{t}) \end{aligned}$$

`car` provides that pair with a pair handling function that returns the first element, and an arbitrary `nil` handling function. Similarly, `cdr` provides a pair handling function returning the second element. `pair?` and `nil?` are logical opposites to each other, each provides a pair- and nil-handling function, returning either `#t` or `#f` as is appropriate.

Together, these functions are sufficient for designing a list implementation. The implementation that comes naturally is known as a linked-list. A linked-list is essentially either a pair of an element and a linked-list or `nil`. If that is unclear, think of a tree with a fractal structure. The tree consists of a leaf and a child tree, which in turn has both leaf and child tree, until the tree ends with `nil` for a child tree.

Recursion

Our last definition will be a bit more esoteric, or at least complex. We define a *Y Combinator*. This function, `Y`, will allow another to be executed accepting itself as an argument.

Self-Reference by a Combinator

$$Y = \lambda f(\lambda x(f)(x)x)\lambda x(f)(x)x$$

You need not delve into its innerworkings; rather, let's work through an example. Let's say you want to define a function that will evaluate the factorial of a number `n`. Well then the fundamental idea would be to do something like the following.

$$fact = (\mathrm{lambda}\ (n)\ (*\ n\ \ \ldots))$$

Well the question remains, what should be present instead of the dots? Nothing we have discussed would give any help in answering that, besides the Y Combinator. If you recall from Mathematics, the factorial has an inductive definition like the following.

Inductive Definition of Factorial

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \end{aligned}$$

Our task is to translate this into our Symbolic Language; however, we wish to generalize this idea a bit, not to add a special expression type for every inductive definition we think of. Hence our duty is to make a factorial function which is self-aware, if you will. The following is a rough form of the concept.

$$fact = (\mathrm{lambda}\ (fact\ n)\ (*\ n\ (fact\ (pred\ n))))$$

There remains one issue! We have not handled the base case present in our prior definition. To achieve this in the Lambda Calculus we will utilize an if statement; this use case was alluded to earlier.

$$fact = (\mathrm{lambda}\ (fact\ n)\ (if\ (zero?\ n)\ (succ\ 0)\ (*\ n\ (fact\ (pred\ n)))))$$

This is a complete realization of the definition, but there remains one problem. How are we to pass `fact` the value of `fact`? This is when the Y Combinator comes in. The invocation of `(Y fact)` will form a factorial function, aware of itself for the sake of recursion, accepting the single variable `n`.

Conclusion

We have now laid a good foundation upon which our Symbolic Expressions can exist. As should be expected, lists will be our primary data-structure in our language of S-Expressions.

Special Forms of S-Expressions

Numbers

Returning to our prior definition of numbers, we will now define arbitrarily long strings of decimal digits. As you can see, the following defines numbers by either matching single digits and defining them as a successor, or by matching leading digits and a final digit and evaluating them separately.

$$\begin{aligned}
1 &= (\mathit{succ}\ 0) \\
2 &= (\mathit{succ}\ 1) \\
&\dots \\
9 &= (\mathit{succ}\ 8) \\
\mathit{ten} &= (\mathit{succ}\ 9) \\
d\ldots 0 &= (\mathit{mul}\ d\ldots\ \mathit{ten}) \\
d\ldots 1 &= (\mathit{sum}\ (\mathit{mul}\ d\ldots\ \mathit{ten})\ 1) \\
&\dots \\
d\ldots 9 &= (\mathit{sum}\ (\mathit{mul}\ d\ldots\ \mathit{ten})\ 9)
\end{aligned}$$

Hopefully the above is a clear embodiment of our decimal number system. Place value is achieved by two measures, (a) inductive definition, and (b) multiplication by ten. This pattern of recursive definition and symbolic pattern matching will be at the heart of our language constructs.

List Literals

We define our lists inductively based on the pair-constructing `cons` function we defined earlier. We choose to name this function `quote` because it is treating the entire expression as a literal, rather than as a symbolic expression. More importantly, the syntax of passed lists is indistinguishable from a regular S-Expression, hence we are utilizing the *quoted* form of such an expression.

The following definition has a rather sensitive notation. Quotes show that an atomic value, that is, a value referred to in our grammar as `<atom>` or more importantly, a value referred to as `<var>` in our grammar of the Lambda Calculus, is being matched. This is unique from most cases in which a portion of a pattern is being labeled by a variable. Additionally, the italicized *ab...* is meant to label the first letter and rest of a string as `a` and `b`, respectively.

$$\begin{aligned}
(\text{quote}\ (a)) &\implies \mathit{cons}\ a\ \mathit{nil} \\
(\text{quote}\ (a\ \mathit{rest}\ldots)) &\implies (\mathit{cons}\ (\text{quote}\ a)\ (\text{quote}\ (\mathit{rest}\ldots)) \\
(\text{quote}\ a\ \mathit{rest}\ldots) &\implies (\mathit{cons}\ (\text{quote}\ a)\ (\text{quote}\ (\mathit{rest}\ldots))) \\
(\text{quote}\ \ \text{" 0 "}) &\implies 0 \\
&\dots \\
(\text{quote}\ \ \text{" 99 "}) &\implies 99 \\
&\dots \\
(\text{quote}\ \ \text{" } \mathit{ab}\ldots\text{"}) &\implies (\mathit{cons}\ a\ (\text{quote}\ b\ldots)) \\
(\text{quote}\ \text{"a"}) &\implies (\mathit{cons}\ 97\ \mathit{nil}) \\
&\dots \\
(\text{quote}\ \text{"z"}) &\implies (\mathit{cons}\ 122\ \mathit{nil})
\end{aligned}$$

In addition to defining the `quote` function, we will provide a shorthand for the operation. So often will we need to define list literals that it makes perfect sense for us to make it as brief as possible.

$$'a = (\text{quote}\ a)$$

Quoted forms will come up often in writing list literals, atomic values derived from strings, i.e., *atoms*, and forming more complex data-structures from lists such as tables.

Equality

We have built up an array of atomic values, and a way of keeping them literal. Now we need a way of recognizing them, by means of equivalence. `eq` already solves this problem for numbers, but not for other quoted atoms. We generalize `eq` to all expressions in our definition of `equal?`.

$$\begin{aligned}(\text{equal? } (a\ b\ldots)\ (c\ d\ldots)) &\implies (\mathit{and}\ (\text{equal? } a\ c)\ (\text{equal? } (b\ldots)\ (d\ldots)))\\ (\text{equal? } a\ b) &\implies (\mathit{eq}\ a\ b)\end{aligned}$$

The above definition is inductive in nature. It provides a base case in which equivalence is determined by the Lambda Calculus definition of `eq`, and an inductive step in which lists are equivalent only if their constituents are equal.

Variable Definition

Now we add some *syntactic sugar* that will make it easier to store values that will be used in an expression. `let` and `let*` set a single value and a list of values, respectively, to be utilized in a given expression. `letrec` takes this idea in another direction, performing the Y-Combinator on a passed function to prepare it for recursion in the passed expression.

$$\begin{aligned}(\text{let } \mathit{var}\ \mathit{val}\ \mathit{expr}) &\implies ((\text{lambda } (\mathit{var})\ \mathit{expr})\ \mathit{val})\\ (\text{let* } ((\mathit{var}\ \mathit{val}))\ \mathit{expr}) &\implies (\text{let } \mathit{var}\ \mathit{val}\ \mathit{expr})\\ (\text{let* } ((\mathit{var}\ \mathit{val})\ \mathit{rest}\ldots)\ \mathit{expr}) &\implies ((\text{lambda } (\mathit{var})\ (\text{let* } (\mathit{rest}\ldots)\ \mathit{expr}))\ \mathit{val}\\ (\text{letrec } \mathit{var}\ \mathit{fn}\ \mathit{expr}) &\implies (\text{let } \mathit{var}\ (Y\ (\text{lambda } f\ \mathit{fn}))\ \mathit{expr})\end{aligned}$$

Once again we provide an inductive definition, and here we finally utilize the Y Combinator we discussed with regard to recursive functions.

Conclusion

We have formed a basic language consisting of Symbolic Expressions defined by the Lambda Calculus. All of our expressions are reducible to Lambda forms, yet clearer or more concise given their symbolic form. This language will be utilized for expression of all sorts of computational ideas, including algorithms, simulators, and interpreters. Our choice of syntax was purely aesthetic; Lambda Calculus is sufficient for communication with machine, however, our language of Symbolic Expressions is far friendlier to a human reader. This motivation reveals the additional motivation for our construction of this language, to form a clear, formal, extensible, and uniform means of communicating ideas.

by Matt Neary