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.

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

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.

Let's begin with a simple function of two variables. The following performs a function `f` on a value `x`, and then `f` once more on the value of that.

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

The following is a function similar to the above, but now accept another argument, `g`, which changes the inner `(f x)` to `(g f x)`. This is a bit convoluted, so we'll evaluate it with an example.

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

Now we combine our prior two examples into a single Symbolic Expression in the following.

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

This is really beginning to appear complex, but let's step our way through the evaluation of this expression.

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

That looks much better! What we arrived at was 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.

Numbers

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. 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.

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

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. In doing this we will return temporarily to our symbolic forms.

We'll begin with 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.

$$\lambda f \lambda x (f)(f)x$$

Now we can evaluate `pred` for this value. Let's, however, briefly reflect on a simpler expression of `pred`.

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

Now we wish to, as an example, evaluate this function for two.

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

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

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

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

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

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

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

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

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

With the predecessor defined, however, 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 quite necessary in expressing conditional statements; hence we will define an `if` function as well.

Booleans

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

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 definitions follow easily from our `if` function.

Boolean Algebra

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

With boolean manipulation and conditionals in hand, we need some useful predicates to make use of 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

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

Pairs

Finally we reach the most important part of our S-Expressions, their underlying lists. 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`.

Pairs

$$cons = \lambda a \lambda b \lambda c \lambda n((c)a)b$$
$$nil = \lambda c \lambda n(n)id$$

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?` complements to each other in determining the end of a list.

Pair Operations

$$car = \lambda l(((l)\lambda a \lambda b a)id)$$
$$cdr = \lambda l(((l)\lambda a \lambda b b)id)$$
$$pair? = \lambda l(((l)\lambda\_\lambda\_\#t)\lambda\_\#f)$$
$$null? = \lambda l(((l)\lambda\_\lambda\_\#f)\lambda\_\#t)$$

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 perform recursion accepting itself as an argument.

Recursion by a Combinator

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

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.

A Language of S-Expressions

TODO: dotted lists

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 single final digit and evaluating them separately.

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

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.

The following has a rather sensitive notation. Quotes show that an atomic value is being matched rather than a portion of a pattern being labeled. Additionally, the italicized *ab...* is meant to label the first letter and rest of a string as `a` and `b`, respectively.

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

In addition to defining this 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.

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

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?`.

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

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.

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

by Matt Neary