A Library of Symbolic Functions

Introduction

We have formed a language constituent of Symbolic Expressions. Additionally, we have an array of useful and primitive Lambda Calculus functions at our disposal. Now in order to build expressive and powerful programs, it will be helpful to define a library of useful Symbolic Functions for manipulation of the various data-types we have formed by abstraction.

Predicates

We begin with a trivial example, a boolean inverter. Our definition is listed as an S-Expression because when we are ready to make use of it, we will include that pair in a call to `let*`. The name of the function is the first element, because this is the variable to which it will be assigned. The actual function definition is very familiar. We form a lambda of a single function that results in conditional behavior; if `x` is true, it results in `#f`, but if `x` is false, it results in `#t`.

$$(not\ (\text{lambda}\ (x)\ (if\ x\ \#\text{f}\ \#\text{t})))$$

Now we add to our current assortment of numeric predicates the functions `<` and `>`. These predicates are lambdas accepting two values, that they may be compared. With less-than-equal-to already defined as `leq`, both of these relations are trivial to define.

$$\begin{array}{ll}(< & (\text{lambda}\ (x\ y)\ (and\ (leq\ x\ y)(not\ (eq\ x\ y)))))\\ (> & (\text{lambda}\ (x\ y)\ (not\ (leq\ x\ y))))\end{array}$$

An important conveniency to note in the above forms, is their nature given their Lambda Calculus definitions. That is, since they compile down to a curried form of a function, in other words, a function returning a function, they are very nice to work with. Let's look at an example form and apply appropriate reductions to get a better view of this function's nature.

$$\begin{array}{ll} & (>\ 2)\\ \Longrightarrow & ((\text{lambda}\ (x\ y)\ (not\ (leq\ x\ y)))\ 2)\\ \Longrightarrow & (\text{lambda}\ (y)\ (not\ (leq\ 2\ y)))\end{array}$$

What we see is that when a single value is passed, we achieve a convenient function ready to compare in terms of that parameter. This may seem obvious, familiar, or redundant, but the convenience of this fact should not be taken for granted. This phenomenon is known as currying; it can prove very useful in providing clearness to your expressions, our look at `>` was only an example of what is going on in all of the functions we discuss.

This ability to supply the arguments of a function one at a time makes for very legible code. Below is an example of an inductive definition utilizing this functionality. The predicate is `<` with the first argument supplied as two, the step is the `pred` function, and the combinator is multiplication.

Generalized Inductive Calculator

```
(let
   induct
   (lambda
      (pred num step combo)
      (if
         (pred num)
         num
         (combo num (step num))))
   (induct (>  2) 6 pred *))
⟹  720
```

This reads very well, as "perform induction while greater than 2 from 6 by means of decrement combining with multiplication", i.e., find the factorial of 6. Let's look at another instance of this, taking advantage of the interchangeability of the definition, and once again of partial function application. This time we induce addition up to and at five, stepping by two in each step up from one.

```
(let
   induct
   …
   (induct (<  5) 1 (+ 2) +))
⟹  16
```

Our answer coincides with what you would expect, the sum of odd numbers one through seven. Hopefully you feel that our language displays complexity well. The above example handled a very generic problem type elegantly and concisely. It is thanks to our adding of abstraction as we go that we are able to make these creations both clear and versatile, and ones in which the creator can take pride.

Lastly we provide predicates to determine whether a given number is even or odd. This function is once again very easy given our convenient Lambda Calculus primitives. The following two functions simply check for a specific value of the modulus division by two applied to a number; this is the essence of parity.

```
(odd?        (lambda  (x) (eq (mod x 2) 1)))
(even?       (lambda  (x) (eq (mod x 2) 0)))
```

Use cases for these two predicates will arise later, but for now they are good at their simple duties, determining parity.

Higher-Order Functions

In writing clear and concise expressions, it is often useful to have at your disposal higher-order functions (*HOF*), that is, functions that (a) return functions, (b) accept functions as arguments, or (c) do both *a* and *b*.

Hopefully you caught something odd in what I just said, the fact that everything, hence any possible argument or resultant value, is a function! However since we have defined some primitive data-types, I am referring to non-data-symbolizing functions. This may seem a fine line, but you will often seen this terminology tossed around, so you may as well utilize it even in when in the purest of functional languages.

We begin with a couple of type (c) HOFs. The first of the following serves to flip the argument ordering of a given function, and the second composes two functions.

```
(flip                    (lambda (func a b) (func b a)))
(compose        (lambda (f g) (lambda  (arg) (f (g arg)))))
```

The function `flip` is very convenient when aiming to apply only the second argument of a function, leaving the other free. The design of `flip` is quite simple, it merely accepts a function, then two arguments, and returns application of them in reverse order. Despite the simplicity of its operation, it can really greatly reduce the complexity of an expression. As an example, look at the following definition of a singleton constructor, that is, a creator of a pair with a single element.

Singleton Constructor

```
((flip cons) nil)
```

`compose` allows the results of various manipulations to be piped from one to another. A beautiful example of this is a linear function creator. Below is the function accepting slope and y-intercept as its two arguments.

Linear Function Generator

```
(lambda
   (m b)
   (compose
      (+ b)
      (* m)))
```

One of the most important HOFs is defined next. `fold` serves to accumulate a list of values into a single resultant value, based on a function of combination and a starting value. Note that this function is recursive and will be provided using `letrec`. Other functions accepting their name as the first argument should be assumed to follow the same practice.

```
(fold (lambda (fold func accum lst)
   (if (null? lst)
      accum
      (fold func (func accum (car lst)) (cdr lst)))))
```

Our definition of `fold` is as a manipulator of a list returning an accumulated value at the end of a list, and at other points recursing with the `cdr` of the list and an accumulator as determined by the passed function. If the meaning of `fold` is still unclear to you, consider some of these examples.

$$(fold \quad + \quad 0 \quad '(1\ 2\ 3)) \implies 6$$

$$(fold \quad * \quad 0 \quad '(1\ 2\ 3\ 4)) \implies 24$$

As you can now see, the folding of an infix operation $a \bullet b$ over a sequence $a, b, c, \ldots$ is the nested application of the operation, or the following.

$$(\ldots((a \bullet b) \bullet c) \ldots)$$

As a complement to `fold` we define `reduce`. `reduce` is just like `fold` except right-associative; Hence the function applications are nested just like the `cons` basis of these lists.

```
(reduce (lambda (reduce func end lst)
  (if (null? lst)
    end
    (func (car lst) (reduce func end (cdr lst))))))
```

Our definition of `reduce` is as a manipulator of a list returning an accumulated value at the end of a list, and at other points returning a manipulation of a recursion with the `cdr`, manipulated by the passed function. If we do an expansion of an infix operator for `reduce` as we did for `fold` we achieve something like this following when dealing with a list $\ldots, x, y, z$

$$(\ldots (x \bullet (y \bullet z)) \ldots)$$

Together `reduce` and `fold` are sufficient basis for any iterative process. Now we will provide an inverse operation for constructing a list given a construction criterion. `unfold` serves to invert a folding.

```
(unfold (lambda (unfold func init pred)
  (if (pred init)
    (cons init nil)
    (cons init (unfold func (func init) pred)))))
```

To clarify the distinction between `fold` and `reduce`, we display the manner in which they can be thought of as opposites.

$$(fold \quad (flip\ cons) \quad nil \quad '(1\ 2\ 3)) \quad \Longrightarrow \quad '(1\ 2\ 3)$$
$$(reduce \quad cons \quad nil \quad '(1\ 2\ 3)) \quad \Longrightarrow \quad '(1\ 2\ 3)$$

This examples drives home that the difference between the two is in direction of association, `reduce` is the natural operation for right associative operations and `fold` for left associative operations.

Together our definitions of `fold` and `reduce` are sufficient for definition of any iterative process. `unfold` in addition provides us with a means of constructing arbitrary lists based on constructing rules. We will now implement a variety of derived iterative forms based on `fold` and `reduce`.

Reductive Forms

We begin with some extensions to our basic binary operators of arithmetic and boolean algebra. The structure of these definitions is similar to that of our early definitions of arithmetic, an iterative process on a base value; however, in this case the conditions and multitude of application are determined by a provided list.

All of the following forms, which we will refer to as *Reductive Forms* are dependent on either `fold`. `fold` provides the generic versatile power to combine a list in an arbitrary way; hence you will see a variety of operations used in folding, so you may want to think back to the examples of the nested operator.

We begin with some definitions of arithmetic and boolean manipulations. The definitions of these forms are intuitive, each with an infix operator which fits the role very intuitively.

```
(sum (lambda (lst)              (fold + 0 lst))
(product (lambda (lst)          (fold * 1 lst))
(and * (lambda (lst)            (fold and #t lst))
(or * (lambda (lst)             (fold or #f lst))
```

Now we expand our application field in defining some optimization functions, `min` and `max`. Both of these works by comparing each element with a running extreme value, swapping if a new extreme is found. The definition of `max` follows, a simple folding onto the higher value.

```
(max
  (list)
  (fold
    (lambda
      (old new)
      (if
        (> old new)
        old
        new))
  (car list)
  (cdr list)))
```

The application of this function to $'(1\ 5\ 3\ 4)$, for example, would return 5. Our implementation of `min` is nearly identical, simply changing the criterion of the sort.

```
(min
  (list)
  (fold
    (lambda
      (old new)
      (if
        (> old new)
        old
        new))
  (car list)
  (cdr list)))
```

Next we define some methods that aid in treatment of lists in their entirety, `length` and `reverse`. `length` is one of the simplest folds you could define, folding by increment. `reverse` on the other hand, is not as obvious in its means of operation; it folds by means of a swap operation, $(flip\ cons)$, in this way forming a fully reversed list.

```
(length     (lambda (lst) (fold (lambda (x y)(+ x 1)) 0 lst)))
(reverse          (lambda (lst) (fold (flip cons) nil lst)))
```

Now we provide a special function for determining associations in a list meant as a table. The setup of these lists is like the following, where each element is a list, with the first element serving as a key, and the second serving as a value.

```
'((apple      red)
  (pear       green)
  (banana     yellow))
```

In determining the association, we `fold` with the aim of reaching a value with a key matching that for which we are searching.

```
(assoc (lambda (x list)
  (fold
    (lambda
      (accum item)
      (if
        (equal? item (car x))
        (cdr x)
        accum)))
    #f
    list)))
```

`assoc` is very important in modeling hash-tables, and in general keeping track of named values. If `assoc` were applied to the table displayed prior with `banana` as a key, it would evaluate to `yellow`. Here is the full form, with `table` referring to the aforementioned table.

$$(assoc \quad 'banana \quad table) \implies 'yellow$$

List Manipulations

In manipulating a list, there are two basic classes of operations, (a) mapping a list to a value, and (b) converting one list to another. We have thoroughly covered the former, starting first with general forms and then implementing some useful examples. Now we will move on to the latter.

In mapping one list to another, we will provide two generic functions.

The first, `map`, will apply a single function to each element of a list; the second will filter out items based on a predicate. These functions are very useful, imagine for example finding a sum of squares or constructing a list of primes.

```
(map
  (lambda
    (func lst)
    (reduce
      (lambda
        (x y)
        (cons (func x) y))
      nil
      lst)))
```

Our map implementation works as a reduction with `cons`; if this were the extent of the function, the initial list would be returned. However, each element is passed through the provided function to result in a list with modified elements. `filter` takes advantage of the same aspect of `reduce`; however, in its definition it casts away values not matching a predicate.

```
(filter
  (lambda
    (pred lst)
    (reduce
      (lambda
        (x y)
        (if
          (pred x)
          (cons x y)
          y))
      nil
      lst)))
```

Let's look at some examples of `map` and `reduce`; the extent of their usefulness was alluded to earlier, but below are some examples to clarify their usage.

```
(map      (* 2)   '(1 2 3))          ⟹ '(2 4 6)
(filter   odd?    '(1 2 3 4 5 6))    ⟹ '(1 3 5)
```

The above were very clear in their meaning, as one would hope. Now that we have some strong ways of manipulating a list, we will move on to means of adding elements to a list. We provide some functions for appending to a list, either a single element of a list of elements, i.e., *concatenation*. These functions serve as nice complements to the two which were defined above, as they allow for expansion to supersets, and the earlier two allow only for constructing a subset.

```
(push     (lambda (a b) (reverse (cons b (reverse a)))))
(concat                 (lambda (a b) (fold push a b)))
```

These list manipulations will prove very useful, and given our prior functions, were very concise and clear in definition. Below are some examples of `push` and `concat` applications.

```
(push     4     '(1 2 3))     ⟹ '(1 2 3 4)
(concat   '(1 2)  '(2 4))     ⟹ '(1 2 3 4)
```

Conclusion

We have amassed a variety of useful and versatile functions of symbolic expressions. With these in hand, we are ready to build complex and useful programs.

by Matt Neary