Functional Concepts

Introduction

In many ways, the functional programming paradigm is simpler than imperative or Object Oriented Programming (OOP) paradigms, but there are a few key concepts that may never have been considered by programmers who have never have been trained to think of them in this new way. These include concepts of what is the true meaning of expressions, how expressions are used to define variables (misnamed in the case of "pure" functional programming as these never "vary" being immutable constants) and as the body of functions, and what the variations of function definitions and the application of functions really mean.

The Key Building Block for Functional Programs - Expressions

A programming expression is a programmatical combination of expression "terms" that can be evaluated to produce a value, where each "term" can in turn be a sub-expression. Thus, an expression defines an evaluation of a set of input values to produce a single output value. The sub "terms" can be as simple as a single value such as a number, a text character, or a boolean value, to something more complex such as a string of characters, a tuple collection of Type'ed constants or a record collection of Type'ed constants (like a tuple with named fields), to tagged unions of possible groups of sub-terms (often known as Arithmetic Data Types - ADT's), with these last complex forms all forms of wrapped collections of "terms". Expressions are then combinations of "terms" with combinations of functions and "operators" (symbolic representations of functions that are often defined to be applied "infix" or between their arguments rather than "prefix" or before their arguments like functions; these also may have a defined application precidence and associativity in their application as to which argument is evaluated first and the order of operator application) applied to them as well as conditional expressions as in `if ... then ... else` and `case ... of ...` expressions. Thus `42`, `z`, and `"Hello"` are constant literal terms as are `[1, 2, 3]` (a list of numbers literal), `("Name", True)` (a two-tuple literal of a String and a boolean value), and `{ name =`

`"Me", age = 42 }` (a record literal containing the name fields for a String and a number). However, any combination of application of operators and function applications also form an expression such as `"This" ++ " and " ++ "that"` (appends String literals together to form a String), `(4 + 7) // 2` (an Int expression using parenthesis to control evaluation order different than default), and `sqrt (x * x + y * y)` (defines the hypotenuse calculation given `x` and `y` represent the adjacent sides). There are also conditional expressions defined with keywords such as `if <boolean expression> then <expression> else <expression>` where it can be considered that the evaluation of the "boolean expression" is the input value with the result the expression in the `then` branch if it evaluates as `True` or the `else` branch expression if `False` and both branches resulting in the same Type. There is usually also a multiway branch expression as in `case <expression> of <one or more target patterns that when matched evaluate an expression result>` where the result of the input expression is matched against each of the target patterns in turn and whichever pattern matches first causes the evaluation of that target expression as the returned value, thus all target expressions must also have the same result Type.

Lack of Statements in "Pure" Functional Programming

An imperative language "statement" differs from a term in that there is no "production" of an output result value but rather is an instruction to do something, such as assign (or re-assign, requiring mutation) the result of an expression to a variable, or to perform some "side effect" is in outputting text to a terminal or printer, waiting for input of text from a console, getting a time value from the external operating system hardware, or getting a random value as per some Type and range constraint. These last are all "side effects" in functional programming terms because in order to work they require some mutable change to the programming "state" or "environment", as in a location in memory is mutated, a printer or console "state" is changed (ie. line number, line contents, etc.) and/or, the result of the operation is not the same for every call with the same arguments (not "referentially transparent" - the function call and its arguments can't be replaced by a constant of the result).

The Distiction Between Functional Programming Definitions and Assignments

An assignment is a "statement" as defined above and when "statements" are allowed as in imperative languages, there may be many assignments to a given "variable" when such variables are mutable/changable. Pure functional programming definitions may look like assignments, but when immutability is enforced there can be only one definition for a given name binding, which then represents a constant and not a variable.

Type Definitions in Functional Programming, Expecially for Static Typing

Dynamically Typed languages don't do Type Checking/Resolution at compile time but rather defer it to run time and try to resolve the required matching of available Type's to as required for expression evaluation and function/operator application as required by the run time. Dynamic Typing has often been described as singular typing with every Type having a "super Type" with variations on the "super Type" to the required type identified with "tag" field values. This is similar to the Custom Type's/ADT's that can be optionally be defined and used in many Statically Typed functional programming languages but with the difference that Type Checking is performed at compile time with an error response if the Type's cannot be resolved, meaning that the definition of Type "Option's" is by programmer design and not just an artifact of the Dynamic Typing implementation. Most functional languages allow Type Aliases to be defined for otherwise already existing Type's (including built-in Type's) and some means of defining new Custom Type's as tagged union Type's/variant records/ADT's. Type aliases might look something like the following examples: `type alias MyInt = Int` (a custom Type Name representing the built-in Int Type), `type alias MyTuple = (Int, Bool)` (a Type Name representing a fixed variation of the generally defined built-in Tuple Type limited to containing only an integer and a boolean value), `type alias MyRecord = { name : String, age : Int }` (a custom Type Name representing a fixed variation of the built-in generally defined Record Type but containing only two named fields of a String and and Int). Custom Type definitions generally take the form of `type MyType = First Int Bool | Second String Bool` (not an alias but a new type with two variants in this case - one or more variants in the general case), tagged as "First" and "Second", with each variant carrying payloads of zero or more other Type's, in this case each variant has two different Type's.

The Definition of Constant Named Bindings in Functional Programming

In the case of definitions of named bindings, there can be only one definition per name (unless name shadowing is allowed, which is a poor programming practice as it leads to programming confusion as to what the latest definition of the name represents, in what scope it exists, etc.). Other than for the definiton of Type Names (capitalized in most functional languages) as in Type Aliases (Type Names representing other Type's already defined) or Custom Type definitions including the capitalized Name's of the Variant Tag's (which can act as Constructor's of the given Type where they are defined, as Value's when there is no payload and as functions when there is), there are two kinds of name binding definitions (whose names generally start with a lower case letter): constant definitons, and function definitions, with the only difference between these last that constant definitions have no parameters representing arguments where functions have at least one parameter. The value of a given constant binding is the result of the expression on the right hand side of the definition. The return value of a function name binding is again the result of the expression on the right hand side of the definition, but the expression may incorporate the named parameters in the expression, making a function just a named binding for an expression with the ability to inject different values for parts of the expression. Function parameters can be of any Type including functions and the Unit/"nothing" value of `()`, and the function expression can return any Type including a function (or a nothing value `()`, although if it did it would make a pure function into a useless function as it couldn't do anything); thus, constant definitions can actually be name bindings for expressions as in the following examples: `myAddFunction = (+)` (an alternate name for the built-in function that gets called by the addition operator), or `myfunc = \ x -> 2 * x + 3` (the expression is an "anonymous"/not named function that computes a number given a number), with the last callable just as for regular functions as in `myfunc 3` results in the number nine. This is both an example of a name binding that represents a function on the left side of the equals sign and a function expression on the right side of the equals sign. Function expressions can also be created by function composition as in the following: `expr = (+) 1 << (*) 2` where the name binding `expr` is defined as the composition of a function that first multiplies the "extra" single argument by two, then composes that result to add one to the result to get the final result. This is written with "the left composition" operator, but is exactly equivalent if writen with "right composition" as `expr = (*) 2 >> (+) 1`. This form of function definition using only function composition with no named parameters is often called a "point free form" function.

Partial Application of Function Arguments

This last example shows the use of partial application of arguments to a function as both the `(*)` function and the `(+)` function have exactly the same function Type signature of taking two arguments of numbers and producing a number result; however when each is used with only a single argument as above, the result is a function that takes a single number argument to produce a number result. Also this is written in "point free form" so rather than being written as `expr x = ((+) 1 << (*) 2) x` or `expr = \ x -> ((+) 1 << (*) 2) x` it is "eta converted" to drop both the `x` as a parameter and in the application to the function. The definition could also have been written as `expr x = (+) 1 <| (*) 2 x` to avoid the need for the extra brackets around the composed function, which is called "backward piping", with the `(<|)` operator defined as `(<|) f g x = f (g x)` but with a precedence lower than any other operator (also lower than function application which is the top precedent) and evaluating the right argument before the left (right associativity). The equivalent "forward pipe" operator, `(|>)`, could have been used as `expr x = x |> (*) 2 |> (+) 1` for the exact same meaning. The important take away from this is that any function can produce a function as a result when less than the full number of arguments are supplied to that function, with the resulting function having the signature of taking the remaining arguments and producing the result, with the arguments and result of the same type as they were for the "full" function.

Currying of Function Arguments

Conversely, all functions can be considered to take exactly one argument to produce exactly one return value and computations that need to take more than one input can then be considered to be an outer function that takes the first argument which returns a function that takes the next input to return a function ... etc. until the final result is obtained. One can see that this works in conjunction with the above ability to just partially apply arguments to functions to produce functions as results.

Other Considerations of Functions Versus Values

From the above definitions, it can be seen that a named constant definition of an expression can be considered to be an immediate evaluation of the given expression on the right hand side (unless in a non-strict/lazy language where all computations are deferred until the result is required for further computation); conversely, a function definition (or expression) can be considered to defer the computation of the expression that produces the result until the function is called, meaning when arguments are applied to that function as represented by its parameters. Thus, as well as being considered a packaging device to bind a name with parameters to the computation of an expression as is often the consideration in imperative languages, a function can also be considered to be a device to delay the computation of an expression result.

Defining "Local" Constant Values and Functions within an Expression Body

Name bindings for constant values and functions defined at the outermost level of a Module (generally one file) are often called topmost or "global" or "Module" definitions; there is a means of defining "local" name bindings to other constant values and functions within the body expression where the expression body definition is extended to this syntax: `let <one or more constant value and/or function declarations> in <expression>`. This definition thus has the "local" definitions within the `let ... in` block with the condition that the names of these definitions cannot be the same as any already defined names at any local or outer scope from the current scope if name "shadowing" is not allowed, and these names can be used freely within the expression following the `in`. This facility can be nested to any level with the same conditions, so any expression part of the local named bindings can have its own `let ... in` definitions, and so on and so on. This is often useful to make it clearly understood that particular constants and functions are only to be applied within an inner "scope".

Functions Capturing Outer Scoped Values as Closure Functions

Also, since most functional programming languages have "scoped functions" so inner functions enclosed as definitions inside outer functions see all defined name bindings from their "scope" outwards, this allows functions to actually use name bindings that aren't defined within their body "scope" nor one of their samed parameter inputs, and these are called a "closure functions" or "closures" for short. For example, in the following code:

```
expr x =
  \ y = x + y
```

the `x` value is used inside the returned anonymous function but comes from a parameter to the outer `expr` function, and can be read as `expr` is a function that takes a single value of `x` and returns a function that takes another `y` value to produce a result which is the sum of `x` and `y`. One can see that this is just a longer way of writing the definition for the `(+)` function (which is exactly what this `expr` does) and looks like it requires a lot of extra steps but we leave it to the compiler to optimize this into exactly the same code "underneath the covers". When the `expr` function is called with an argument for the `x` parameter and the capturing anonymous closure function is returned, the value of the `x` argument will be captured and preserved to be applied along with the argument for the `y` parameter when the returned closure function is called.

Closure Functions Whose Purpose is to Delay Execution

Given the explanation of closure functions as per the above, there is a special type of closure function often encountered in functional programming when it is desired to defer execution until later or indefinitely: the thunk function (almost always also a closure function). A thunk function is one that takes as an argument the "nothing" type of Unit or `()` (like an empty tuple) and returns something that may be the result of an expensive computation. A naive version of a thunk function is the non-capturing (and therefore non-closure) `thunk() = 42` but that is relatively useless as it just returns a value that is a number literal and easy to compute. More interesting uses of thunks is using them to defer a complex (or not) computation until the results of that complex computation is required or using them to represent an infinite stream such as the infinite stream of natural integers from zero up. To produce such a stream, we need to define a Type to contain and represent it, which might be a Co-Inductive Stream (CIS) with a head and a tail something like a singly linked List but with a deferred computation to produce the tail; the difference between a CIS and a lazy list is that a lazy list has a facility so that the computation is run just once and the result cached/"memoized" for any further references. The following CIS Type definition is generic meaning that it can contain a stream of any Type but every node contains the same Type:

```
type CIS a = CIS a (() -> CIS a) -- the tail function is a "thunk"!
```

where the first `CIS` in the Type definition is the defined Name of the new Custom Type and the second `CIS` is the Constructor Function for the variant of the CIS Type, of which there is only one variant (this stream doesn't have the possibility of being less than of infinite length, so doesn't have an `EmptyCIS` or some similiarly Name'ed variant.

Now we can define a natural number stream constant that is a CIS of all the natural whole numbers from zero up, as follows:

```
nats =
  let nxtnat i = CIS i (\ () -> nxtnat (i + 1))
  in nxtnat 0
```

where `nxtnat` is an internal scoped function that takes a number and produces a CIS stream of its argument and a "thunk" tail closure function (captures the external current value of `i`) that calls `nxtnat` with the next value to use as the head of the tail CIS and so on infinitely. To use the nats value, we just create a function that recursively "chases the tail" until some condition is met. For instance, if we provide the following `nthCIS` function:

```
nthCIS n (CIS hd tlf) =
  if n <= 0 then hd
  else nthCIS (n - 1) (tlf())
```

then one can obtain the trivial example of the twentieth natural number by `nthCIS 19 nats` (zero index based). Because the `CIS` Type only has one variant, we can use pattern matching for the `nthCIS` function cis parameter (more than one variant would be ambiguous as to what to do for the other variants) which destructures to obtain the head and tail (thunk function) of the current `CIS` variant "pair" which if counted down to zero, outputs the current head else calls itself recursively (from tail call position, meaning the last thing it does, so that it doesn't build a return stack).

The point of this exercise is to show the use of "thunk's" to permit building a data structure representing an "infinite" sequence (infinite until the range of the integer representation may be exceeded, whatever limit that is if any).

Advanced Means of Expressing Recursion

The above `nats` definition uses direct recursion in the definition of the internally scoped `nxtnat` function which calls itself with new arguments as the stream progresses. Some languages (very few these days) do not allow direct recursion so as to accomplish this goal and the idea must be expressed in other ways. One of the most common ways is through the use of a "Y combinator" which has many different means of expression. In non-strict/lazy languages such as Haskell, this can be easily expressed directly as in the following pair of combinators, with the first one a sharing recursion (the recursive value is re-used) and the second non-sharing (the recursion uses a separate instance for the input and output):

```
fix :: (a -> a) -> a
fix f = let x = f x in x
fixy :: (a -> a) -> a
fixy f = f (fixy f)
```

where both have the Type signature that they take a function argument that takes a Type and produces the same type that when applied to the `fix`/`fixy` function produces just a value of the same type. As mentioned, this works fine in Haskell where everything is lazy, but in a "strict"/non-lazy-by-default (most other languages), it recurses infinitely until it blows the return stack unless there are guards built into the use of the `f` function (can't be used generally). However, we can inject some deferred laziness into the definitions in languages such as Elm, so these can be defined as follows:

```
fix : ((() - a) -> a) -> a
fix f = let xf() = f xf in xf()
fixy : ((() - a) -> a) -> a
fixy f = f (\ () -> fixy f)
```

Both of these definitions work using some direct recursion, which both Haskell and Elm support; however, once defined, these combinators allow one to express recursion without the use of further direct recursive calls. All one needs is a `f` function defined to pass to the combinator function and the recursion will be accomplished. For instance, for the `nats` recursion, expressing the recursion using the combinator is a little more complex as we need to proceed in two steps, first using recursion to define a function that takes a `CIS` to produce the next step of `CIS`, then use recursion from an initial condition to produce the final `nats` sequence. The recursive advance function can be defined as follows:

```
advfunc : CIS number -> CIS number
advfunc= fix (\ fn -> \ (CIS hd tlf) -> CIS (hd + 1) (\ () -> fn ()
(tlf())))
```

where `fn : () -> (CIS number -> CIS number)` or `fn : () -> CIS number -> CIS number` (same thing by partial function application). The `a` in the `fix`/`fixy` type signature here represents `CIS number -> CIS number`, which is the signature for the `advfunc` function.

Given the above is a function that advances the CIS one step when called, we can then define `nats` as follows:

```
nats = fix (\ tailf -> advfunc (CIS -1 tailf))
```

where the initial head is here used as `-1` instead of `0` because the advfunc is applied once before the first element of the desired sequence is produced.

Now there is no real need to go through these hoops when direct recursion is so available and easy, but it shows that one can move the direct recursion away so that the recursion is expressed in terms of function parameters. There are other more complex forms of "y combinator" available to be used when there is not direct recursion available at all, even as used in the definitions of `fix` and `fixy` above.

Advanced Handling of State in Pure Functional Programming

There are many types of problems that involve an evolution of a state as a computation progresses. In an imperative or OOP language, the programmer would usually handle this state with a mutable/changeable data structure with the mutable/changeable value(s) mutated as the computation progresses. An example of this might be a decoder or encoder of a JSON file encoded as a text string, where an imperative programmer would either mutate the string as the computation progresses or at least mutate an index value that records the current place in the text string to which decoding has progressed. A pure functional language doesn't have mutation, so what do we do? The answer is generally that we will use an combinator that carries the current state with it as it proceeds with the computation and produces a new result value along with a new state at each computation step. For example, suppose we want to decode what we believe should be a Comma Separated Values (CSV - a text/"String" format) of integers plus optional separation "fill" spaces that should end when the `String` does, this might be how we would do it in Elm:

```
type alias State = (Int, String) -- contains the current location in the
string and the input string
type DStep a
  = Succeed State a
  | Error String -- the error encapsulates an error message
type Decoder a = Decoder (State -> DStep a) -- permits a chain of
computations

succeed : a -> Decoder a
succeed a = Decoder <| \ s0 -> Succeed s0 a

fail : String -> Decoder a
fail msg = Decoder <| \ s0 -> Error msg

chain : Decoder a -> (a -> Decoder b) -> Decoder b
chain (Decoder stf) chf = Decoder <| \ st0 ->
  case stf st0 of
    Error msg -> Error msg
    Succeed nst a ->
      let (Decoder nstf) = chf a in nstf nst

type Step a b = Done a | Loop b

loopDecoder : b -> (b -> Decoder (Step a b)) -> Decoder a
loopDecoder init stpf = Decoder <| \ st0 ->
  let loopb b st =
        let (Decoder stf) = stpf b in
        case stf st of
          Error msg -> Error msg
          Succeed nst rslt ->
            case rslt of
              Done a -> Succeed nst a
              Loop nb -> loopb nb nst
  in loopb init st0

runDecoder : Decoder a -> String -> Result String a
runDecoder (Decoder stf) input =
  case stf (0, input) of
    Error msg -> Err msg
```

```
      Succeed (pos, _) a ->
        if pos == String.length input then Ok a
        else Err "Didn't process to the end of the input string!"

-- just moves position ahead for spaces, possibly repeated, never errors
chompSpace : Decoder ()
chompSpace = Decoder <| \ (pos, str) ->
  let loopi i tstr =
        if String.isEmpty tstr then Succeed (pos + i, str) () else
        let (chr, ntstr) = String.uncons tstr |> Maybe.withDefault ('z',
"") in
        if chr /= ' ' && chr /= '\r' && chr /= '\n' && chr /= '\t'
        then Succeed (pos + i, str) ()
        else loopi (i + 1) ntstr
  in loopi 0 (String.dropLeft pos str)

chompComma : Decoder Bool -- tests for comma, advancing one if found
chompComma = Decoder <| \ ((pos, str) as st) ->
    let (chr, _) = String.uncons (String.dropLeft pos str)
                     |> Maybe.withDefault ('z', "")
    in if chr == ',' then Succeed (pos + 1, str) True
       else Succeed st False

chompValue : Decoder String
chompValue = Decoder <| \ (pos, str) ->
    let loopi i tstr =
          if String.isEmpty tstr
          then Succeed (pos + i, str) (String.slice pos (pos + i) str) else
          let (chr, ntstr) = String.uncons tstr |> Maybe.withDefault ('z',
"") in
          if chr == ' ' || chr == ','
          then Succeed (pos + i, str) (String.slice pos (pos + i) str)
          else loopi (i + 1) ntstr
    in loopi 0 (String.dropLeft pos str)

intDecoder : Decoder Int
intDecoder =
  chain chompValue <| \ vstr ->
    case String.toInt vstr of
      Nothing -> fail ( "Not a valid integer string of '" ++ vstr ++ "'!" )
      Just i -> succeed i

intListDecoder : Decoder (List Int)
intListDecoder =
  loopDecoder [] <| \ olst ->
    chain chompSpace <| \ _ ->
    chain intDecoder <| \ i ->
    chain chompSpace <| \ _ ->
    chain chompComma <| \ notdone ->
      if notdone then succeed <| Loop (i :: olst)
      else succeed <| Done (List.reverse (i :: olst))
```

now calling `runDecoder intListDecoder "1, 2, 3, 4, 5"` will produce as a result `Ok [1,2,3,4,5]` with many other encodings producing a result of `Err "some error message"`.

It works by defining a `Decoder` Type that carries the continuations of the `State` Type Alias through the chained computations of multiple integer strings separated by commas, with variable amounts of white space before and after each integer text grouping. Each computation can succeed or fail, and if it fails then that failure message is flowed through to the result of the entire chain of pending computations. The `loopDecoder` function enables decoding of very long lists of integers by "lifting" the computation loop into Tail Call Position so as not to build state for the recursive loop. The "runner" starts the whold Decoder chain with the initial `State` and checks that the entire input String has been processed.

Of course, rather than write the decoder for oneself as above, one might use a package that already can do most of it for use such as the "elm/parser" package, but internally it (and the "elm/json" encoder/decoder package) works as per the above code.

The Completeness of Functional Programming Concepts (Very Advanced)

In the 1930's, Alonzo Church and Alan Turing proved that anything that was mechanically computable on the theoretical "Turing machine" (a system doing computations through the use of numerical encodings on a tape that could be played and re-played as necessary for the computation) could be computed by Church's "the Lambda Calculus", which is formalization of the definition, application, and composition, of functions and is still used to prove the validity of programming computations; added to this proof was a proof at about the same time from Steven Kleene that these were also equivalent to "general recursive functions" which could also be used to do the same computations. One of the somewhat practical outcomes of this work on functional algorithms was the invention of Church encoding which included Church Numerals using "the untyped Lambda Calculus" (and later the more restricted simply typed and typed Lambda Calculus, which produces more exact proofs) which permits no other primitive values other than pure functions. Church numerals can express all numbers as sequences of function applications and thus all numerical operations on such numbers as compositions and applications of nothing but functions. This work can also be extended to (statically) Type'd functional languages through the use of the definition of Type wrappers for the wrapped pure functions, although for strict (non-lazy) functional languages, the wrapping Type needs to be extended slightly so that both functions and values can be represented by the same Type if it is required to be able to convert from a Church numeral representation to a conventional value.

The use of Church encoding/Church Numbers is never very practical as it requires multiple levels of function application per Church number that increases with the numbers absolute size and is often expressed recursively in a way that increases a return "stack" size with more and more complex operations on larger "values" to possibly use more memory than is available on a given machine. However, writing the code for the basic Church Number operations is instructive and proves the completeness of a given computing language's Type system (if statically typed) and its ability to manipulate pure functions. The ideas behind these implementations are actually quite simple: a Church zero representation is nothing but a function that returns the identity function for any input (thus a so-called constant/`always` function), a Church one representation is nothing but an identity function that takes any function and returns that function (so therefore itself an identity function), a Church two representation is a function that is just the composition of a function with itself as in `f << f`, a Church three applies one more level as in `f << f << f`, and so on... For untyped (or dynamically typed) languages where type resolution occurs at runtime just sufficient to satisfy the requirements of the operators, the above is very easy to implement; however, for statically typed non-lazy languages (where values are not in turn functions), some extra Type trickery needs to be used in order for "higher rank functions" (functions taking functions taking functions as arguments, for instance), being compatible with "lower rank functions" (functions taking levels of less "rank" than this). As well, if there needs to be a capability to convert back to normal non-Church numeric values, the system must be able to represent both values-that-are-not-functions as well as the above multi-ranked functions.

The definition of the Church encoding Type in GHC Haskell is as follows:

```
newtype Church = Church { unChurch :: forall a. (a -> a) -> a -> a }
```

which can be read as "a Church Type is defined as a wrapper around a function that takes a function `a -> a` and returns a function of the same function type, but GHC Haskell has a extension feature used here that says that Type `a` can be of any `forall` Type (meaning that it can take functions of any Rank/Arity including lazy "thunks" for constant values as described in the previous section, functions, functions that take functions as arguments, etc.), where many functional languages such as Elm don't have that feature. One can attempt to define the Elm equivalent to the above using just a Type Alias as follows:

```
type alias Church a = (a -> a) -> a -> a
```

however, this definition runs into problems in generality for higher order Church functions that require applying functions of higher Rank/Arity (as in functions taking functions). The following Elm recursive definition solves part of that problem in using a wrapper Type as in the Haskell implementation:

```
type Church a = Church (Church a -> Church a)
```

in that the recursive definition then allows functions of any Rank/Arity to be nested indefinitely and Type Check; however, the wrapped value must always be a function and not a value, which containing values is required if it is desired to convert the Church Numeral back to conventional numerics. The following "tagged union" definition then allows for the Church Type to be able to both descibe wrapping functions or values:

```
type Church a = Church (Church a -> Church a)
              | ArityZero a -- treat a value as a function
```

Now one can proceed to provide "helper functions" for this last definition for the application of Church numbers to Church numbers and composition of Church numbers as functions, as follows:

```
applyChurch : Church a -> Church a -> Church a
applyChurch ch charg = case ch of Church chf -> chf charg
                                  ArityZero _ -> charg -- never happens!
```

```
composeChurch : Church a -> Church a -> Church a
composeChurch chl chr =
  case chl of Church chlf ->
                  case chr of Church chrf -> Church (chlf << chrf)
                                 otherwise -> chr -- never happens!
                 otherwise -> chl -- never happens!
```

Note that the "never happens" conditional branches are there because all pattern matches must be defined as "total" or covering all possible cases, but those particular cases will never be used by the conversion functions that use `ArityZero` variants as defined as follows:

```
-- conversion functions...
intToChurch : Int -> Church a
intToChurch i = List.foldl (\ _ ch -> succChurch ch) churchZero (List.range
1 i)
churchToInt : Church Int -> Int
churchToInt ch =
  let succInt = Church <| \ ach -> case ach of ArityZero v -> ArityZero (v
+ 1)
                                               otherwise -> ach -- never
happens!
  in case applyChurch (applyChurch ch succInt) <| ArityZero 0 of
       ArityZero r -> r
       otherwise -> -1 -- never happens!
```

Now all of the Church numeral values and functions can be defined with combinations of these definitions such as the following, which were discussed above:

```
churchZero : Church a
churchZero = Church <| always <| Church identity
churchOne : Church a
churchOne = Church identity
succChurch : Church a -> Church a
succChurch ch = Church <| \ chf -> composeChurch chf <| applyChurch ch chf
churchTwo : Church a
churchTwo = succChurch churchOne -- or "intToChurch 2"
```

where using a combination of these provided tool functions, it is possible to define a complete set of integer, boolean, and even floating point (which is composed of combinations of integer operations) on both positive and negative numbers (where the sign is a Church boolean value). Some of the more basic Church functions have been implemented in Elm here, but there are online sources for many many more such as on Wikipedia.

As said at the outset, the use of Church numerals in real commercial programs isn't really practical as compared to using non-functional constant values, which all modern function languages support, but it is an interesting exercise in proving that a language's Type system is complete and therefore compatible with the typed Lambda Calculus, which has applications in proof of correctness for programming languages and systems. However, programmers need not be able to apply the Lambda Calculus (and general category theory) to write functional programs, although understanding it may help better understand the principles involved. A large part of The Lambda Calculus difficulty is learning the usual notation and vocabulary used to express it, which is different than other notations used for other fields of mathematics and computer science.

Conclusion

It should be clear from these examples and explanations that there are very few core functional concepts, being limited to the use of expressions bound to constant values or expressions bound as the body of functions with parameters that can be injected into the function body expression, and that functions can be used as values both as parameters/arguments to functions and as return values from functions and constant values (and therefore expressions), but that the application of these very simple concepts can be used to solve any programming task. Understanding the advanced sections on the "combinator" handling of state and especially the relationship of the Lambda Calculus to functional programming aren't absolutely essential to being able to use the functional programming paradigm, but may aid in understanding it more completely.