On Fixed Point Combinators and Similar Amenities

Luca Abeni

January 11, 2023

1 Fixed Point Combinators

In general, a combinator is a higher-order function (function that takes other functions as arguments and/or returns functions as a result) with no free variables (that is, all variables used in the combinator are bound in its local environment: they are therefore local variables or parameters)[1].

The *fixed point combinators* (fixpoint combinators) are particularly important in the context of functional programming (and its theoretical foundation, λ calculus). A fixed point combinator is a combinator that computes the *fixed point* of its argument. In other words, if g is a function, a fixpoint combinator is a function Fix with no free variables such that $Fix(g) = g(Fix(g))$.

Note that defining a generic higher order function Fix that computes the fixed point of the function passed as an argument is quite easy: by definition

$$Fix(g) = g(Fix(g))$$

and this expression can also be seen as a definition of Fix if we consider a small extension of the λ calculus that allows us to associate names with λ expressions

$$Fix(g) = g(Fix(g)) \Rightarrow Fix = \lambda g.g(Fix(g)). \tag{1}$$

As a first consideration, it is interesting to note that if we try to evaluate the equation 1 using an *eager* strategy (evaluation by value), we get

$$Fix(g) = (\lambda g.g(Fix(g)))g \rightarrow_\beta g(Fix(g)) = g((\lambda g.g(Fix(g)))g) \rightarrow_\beta g(g(Fix(g))) = ...$$

and the reduction diverges. This happens because an eager evaluation strategy will always evaluate the innermost expression "$Fix(g)$" by expanding it to "$g(Fix(g))$" and so on... Using instead a *lazy* (evaluation by name) strategy, "$Fix(g)$" is evaluated only when g actually invokes it recursively (hence, it is not evaluated when arriving at the inductive basis... This guarantees that if if the various recursive invocations lead to the inductive basis then the evaluation of $Fix(g)$ does not diverge). This observation is important when trying to implement a fixed point combinator in eager languages Standard ML.

Another important observation is that the "Fix" function defined in Equation 1 allows (by construction) to compute the fixed point of its argument g, but it is not a combinator: in particular, the definition of Fix uses the "Fix" variable which is free, not being bound by any λ (while the definition of a combinator should contain only bound variables). This is especially relevant because "Fix" is just the name of the function being defined, so the definition of "Fix" is recursive. In other words, we just moved the recursion from the definition of g to the definition of Fix.

There are many different fixed point combinators that can be used to compute the fixed point of a function without using explicit recursion either in the definition of the function or in the definition of the combinator. The existence of fixed point combinators has a remarkable theoretical importance (in practice, it shows that "pure" λ-calculus - without environment or extensions that allow to associate names to expressions - can implement the recursion and is Turing complete). From a practical point of view, it means instead that a functional language that does not implement "`val rec`" (or the equivalent "`fun`"), "`let rec` ' ' or similar...) may still allow the implementation of recursive functions!

The most famous of the fixed-point combinators is the **Y combinator**, developed by Haskell Curry (yes, the names are always the same at the end...), whose definition (using the λ-calculus) is:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{2}$$

[1] Remember that in the particular case of λ calculus a combinator is defined as a λ expression which does not contain free variables, consistently with this more generic definition.

It is very important (especially when searching for information on the internet!) to notice that in the literature there are some small terminological inconsistencies: while in general the Y combinator is *a particular* fixed point combinator, some people tend to use the term "Y combinator" to identify a generic fixed point combinator (and thus write that there is an infinity of Y combinators).

2 Haskell Implementation

The Y Combinator is defined as

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

Trying a simple conversion from the λ-calculus syntax to the Haskell syntax ("λ" becomes "\" and "." becomes "`->`") we obtain

```
y = \f -> (\x -> f (x x))(\x -> f (x x))
```

However, this expression is not accepted by a Haskell compiler or interpreter. For example, `ghci` generates the following error:

```
Prelude> y = \f -> (\x -> f (x x))(\x -> f (x x))

<interactive>:1:23: error:
    o Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
        Actual type: (t0 -> t) -> t
    o In the first argument of 'x', namely 'x'
      In the first argument of 'f', namely '(x x)'
      In the expression: f (x x)
    o Relevant bindings include
        x :: (t0 -> t) -> t (bound at <interactive>:1:13)
        f :: t -> t (bound at <interactive>:1:6)
        y :: (t -> t) -> t (bound at <interactive>:1:1)
...
```

The meaning of this error becomes clear when trying to figure out the type of "`x`". Assuming that "`x`" has type "`t0`", we have:

- Since "`(xx)`" indicates that "`x`" is *a function* applied to an argument, the type "`t0` " of "`x`" must be a function. Let us consider the more generic case: a function $\alpha \to \beta$, therefore $\texttt{t0} = \alpha \to \beta$
- On the other hand, the function "`x`" is applied to the argument "`x`". So the type α of the function argument "`x`" must be equal to the type of "`x`" ("`t0`"): $\texttt{t0} = \alpha$.

Putting it all together you get

$$\texttt{t0} = \alpha \to \beta \wedge \texttt{t0} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

which defines the type "`t0`" recursively (as indicated by the error message, "`t0`" must be equivalent to "`t0 -> t`"). Since Haskell supports recursive data types, one would think that it should be able to support the "`t0`" type we are talking about. However, a recursion like $\alpha = \alpha \to \beta$ doesn't make much sense in a language performing strict type checks like Haskell. To understand why, it is necessary to go into a little more details about recursive data types.

When faced with a definition such as $\alpha = \alpha \to \beta$, we speak of *equi-recursive data types* (equi-recursive data types) if α and $\alpha \to \beta$ represent the same data type (same values, same operations on values, etc...). We speak instead of *iso-recursive data types* (iso-recursive data types) if the type α and the type $\alpha \to \beta$ are not equal, but there is an isomorphism (function $1 \to 1$, invertible, which for each value of α associates a value of $\alpha \to \beta$ and vice versa) from one to the other. This isomorphism (which establishes that the two types are essentially equivalent) is the "constructor" function of the algebraic data types. Understanding that Haskell supports iso-recursive types (a value of α can be generated using a special constructor starting from a value of $\alpha \to \beta$), but not equi-recursive (α and $\alpha \to \beta$ **cannot** be the same type!) tt then becomes clear why the definition of `Y` given above generates a syntax error (in particular, a type error).

This error does not mean that the definition of the Y combinator (Equation 2) is "wrong", but simply that it is not directly implementable in Haskell (or in any other language that uses strong typing). The Y combinator is defined using the λ-calculus, where each identifier is bound to a function, whose type is not important. Using languages with "less strict" type checking than Haskell (for example, Lisp, Scheme, Python, Javascript, etc...) or languages that support equi-recursive types (for example, OCaml with appropriate options), Equation 2 can be implemented without issues.

To better understand how to solve this problem, let us consider the "problematic part" of the previous expression (the application of function "`(xx)`"), focusing on the function "`\x -> (xx)`".

Since in Haskell a recursive data type `t0` can be defined using the `data` construct and (at least) a constructor mapping values of a type that depends on `t0` into values of `t0` (iso-recursive type definition), one can define something like $T = F(T \to \beta)$ to "simulate" the (equi-recursive) type $\alpha = \alpha \to \beta$ of function `x`. The resulting type `T` will then be a function of the type β and in Haskell this is denoted by "`T b`". Remebering the syntax of the "`data`" keyword, we can write

```
data T b = F (T b -> b)
```

where (as mentioned) "`T b`" is the name of the type and "`F`" is the name of the constructor that maps values of `T b -> b` into values of `T b`.

At this point, it is possible to use "`T b`" to correctly type "`x`" by writing "`x`" as a function `T b -> b` from `T b` to `b`. The argument (actual parameter) of this function must therefore be of type `T b`, which can be generated from "`x`" using the constructor `F`. Instead of "\x -> x x" we can then write:

```
\x -> (x (F x))
```

so that Haskell is able to understand and compile this definition.

Note how the impossibility of using equi-recursive types forced us to use the `F` constructor. The type of the function defined above is `(T b -> b) -> b` (function that maps values "`x`" of type "`T b-> b `" to values of type "`b`").

Now that we have seen how to solve the "`x`" type issue, it is possible to go back to the original Y combinator expression, which still has a similar problem: "`\x - > f(x (F x))`" applies to itself, so its type is recursive like the type of "`x`"! Again the problem can be solved by using the previously defined "`T b`" datatype: "\f -> (\x -> f (x (F x)))" has type "`(b -> c) -> (T b -> b) -> c`", so assuming "`b -> c `" as type for "`f`" we have that "`\x -> f (x (F x))`" has type "`(T b - > b) -> c`"... The value "`\x -> f (x (F x))`" to which it is applied must therefore have type "`Tb -> b`". Hence, "`(T b -> b)`" (type of "`x`") must be replaced with "`T b`" (obtainable from "`x`" by applying the constructor "`F`"). The argument will then be "\f -> (\ (F x) -> f (x (F x)))".

As a result, the expression of the Y combinator should be:

```
y = \f -> (\x -> f(x (F x)))(\ (F x) -> f(x (F x)))
```

and in theory this expression should be accepted by Haskell! Unfortunately, however, some `ghc` (and therefore `ghci`) versions have problems correctly inferring data types[2]. Other programs, such as the Hugs interpreter (`https://www.haskell.org/hugs`) are able to parse and evaluate this definition without problems.

The problem encountered by some versions of `ghc` can be overcome by somehow "helping" the compiler to correctly infer the types of the various subexpressions. For example, you could replace "`F x`" with a variable "`z`" gaving the right type (`T b`). To do this, however, you need a function that allows you to extract "`x`" from "`F x`"; in practice, the inverse function of the "`F`" constructor:

```
invF (F x) = x
```

At this point it is possible to replace "`\(F x)`" with "`\z`" and the following "`x`" with "`invF z`", obtaining

```
y = \f -> (\x -> f(x (F x)))(\z -> f((invF z) z))
```

and this expression is accepted by all versions of `ghci`!

Summing up, the first part of the expression ("`\x -> f(x (F x))`") has type "`(T b -> b) -> c`' ', while the second part ("`\z -> f((invF z) z)`") has type "`T b -> c`" and is therefore usable as an argument to the former simply by setting $\beta = \gamma$.

The type of the resulting function is "`(b -> b) -> b`", where "`b`" is clearly a function type (for the factorial, for example , is a function "`Int -> Int`").

[2] This is probably related to a bug described in `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bugs.html`.

The above discussion, which shows how it is possible to implement the fixed point combinator Y in strictly typed languages (like Haskell) that do not support equi-recursive data types, allows us to intuit one important thing: Y "eliminates recursion " by the fixed point operator (Equation 1 uses explicit recursion) by moving the recursion to the data type (indeed it requires using recursive data types of some kind). This is not immediately visible in Equation 2 or scheme (or similar) implementations of Y, but it becomes clearer trying to implement Y in (for example) Haskell.

Once Haskell has "accepted and understood" our definition of Y, one can, for example, use this Y combinator to define the factorial function. First of all, let's define a "closed" version `fact_closed`[3] of the factorial function as

```
fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
```

At this point the factorial function can be defined as

```
fact = y fact_closed
```

Note that the type of "`fact_closed`" is "`(Eq p, Num p) => (p -> p) -> p -> p`", therefore, since Y has type "`(b -> b) -> b`" "`y fact_closed`" has type "`(Eq p, Num p) => p -> p`".

So, a possible definition of the Y combinator in Haskell (there are many others) can be:

```
data T b = F (T b -> b)
invF (F x) = x
y = \f -> (\x -> f(x (F x)))(\y -> f((invF y) y))
```

Based on this definition it is possible to use `ghci` to do:

```
Prelude> fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
Prelude> :t fact_closed
fact_closed :: (Eq p, Num p) => (p -> p) -> p -> p
Prelude> fact = y fact_closed
Prelude> :t y
y :: (b -> b) -> b
Prelude> :t fact
fact :: (Eq p, Num p) => p -> p
Prelude> fact 3
6
Prelude> fact 4
24
...
```

[3]This function is called "closed" because it has no free variables. The traditional recursive definition of the factorial function instead uses a free variable (its name) to call itself recursively.