Y in Practical Programs*
Extended Abstract

Bruce McAdam
bjm@dcs.ed.ac.uk
Laboratory for Foundations of Computer Science,
The University of Edinburgh,
EH9 3JZ UK

July 27, 2001

Introduction

For typical working programmers, the Y combinator for finding the fixed point of higher order functions is seen, at best, as an idiosyncratic example of the features of functional programming languages or, worse, not understood at all. We are going to see that it is actually useful in programming development.

If we program recursive functions in a form that uses Y instead of the recursive constructs built in at the language level then we gain control over and information about how programs are executed. The examples we will investigate include creating memo functions (in languages with mutable data), providing dummy or default results for failed function calls and building call trees annotated with arguments and results.

As well as demonstrating the practical properties of this programming technique, we will see an interesting property relating to the theory of *sequential realisability* [Lon99].

Y in programs

The Y combinator as it is usually expressed in the λ-calculus is not suited to most programming languages as it will not type check. There are two ways around this: either use the programming language's recursion construct, or (more esoterically) its recursive type construct. In Standard ML [MTHM97], Y is most easily written using `fun`, the recursive function construct

```
fun Y f x = f (Y f) x
```

This definition is frequently given to students on undergraduate programming courses, and some even deal with the λ-calculus version [FF95]. There does not, however, appear to be any evidence that this is widely known to computing professionals.

To use Y to define a recursive program, we must write a function to fix. We will call this the *seed*. Here is the seed for the factorial function

```
fun fact_ fact x = if x = 0 then 1 else x * (fact (x-1))
```

`fact_` is the name of the seed. Its first argument, `fact`, will be the fixed function (which will be the factorial function) and the final argument, `x`, is the value to find the factorial of. The 'recursion' in the seed comes from the use of `fact`.

The seed is fixed to create the working factorial function as follows

*This is based on material in [McA97].

```
val fact = Y fact_
```

This will yield a function equivalent to

```
fun fact x = if x = 0 then 1 else x * (fact (x-1))
```

Note the similarly between the definitions of `fact` and `fact_`. This indicates that there is no extra work created for the programmer in using the technique.

Wrappers

Now that we have a seed, we will see what else we can grow from it. We can do this by using wrappers. Here is a simple wrapper to print out intermediate results (using Standard ML's side effects).

```
fun printerWrapper f_ f x =
 let
  (* Find the result *) val result = f_ f x
  (* Print it on a new line *)  val _ = printInt result
 in
  (* Return the result *) result
 end
```

We apply the wrapper as follows

```
val factPrint = Y (printerWrapper fact_)
```

Running this function will print a list of intermediate results to the screen.

It is not possible to directly code this wrapper in a programming language (such as Haskell) lacking side effects. Instead monads would have to be used. The difficulties of using Y with monads are discussed in [EL00].

Applications

We have seen one wrapper for printing intermediate results. It is important to realise that the effect was achieved without recompilation of the seed. This makes the programmers task easier by separating the printing (which is usually temporary and intended to help debug programs during development) with the actual action of the program. Because the seed does not need to be recompiled, we can even apply wrappers to precompiled seeds without needing to access or understand the original source code.

Memo functions

We can write a wrapper to add memory to a seed.

```
fun memoWrapper domain f_ f x =
 let
   (* Create mutable memory *) val mem = newMemory domain
 in
   case lookupMemory x of
     SOME result => result (* already computed *)
   | NONE =>
      let
        (* compute result *) val result = f_ f x
        (* store in memory *) val _ = store (memory, x, result)
      in
```

```
        (* return result *) result
      end
 end
```

To create a memo factorial function, use `Y (memoWrapper 100 fact_)` (where 100 is to be the limit of the memory domain). You can check that it really works by using `Y (printerWrapper (memoWrapper 100 fact_))`.

Filling in missing results

A function may not always be able to give an answer. In this case it may return an optional result (`NONE` or `SOME` *value*). We can write a wrapper to replace `NONE` with a default value (e.g. in unification, when a recursive call fails we may wish to continue with the identity substitution.).

```
fun supplyDefault default f_ f x =
  case f f_ x of
    NONE => default (* optionally print an error message here *)
  | SOME result => result
```

Note that the result returned by `f_` is different from the result returned by `supplyDefault f_`. Before wrapping the seed cannot be fixed as the type it returns is different from the type is expects on 'recursion'.

I have used this technique extensively with unification algorithms for type inference as an alternative to exceptions.

Gathering Call Trees

We can record call trees (with arguments and results) in the following ML datatype

```
datatype ('arg, 'result) call_tree =
  NODE of ('arg * 'result * (('arg, 'result) call_tree list))
```

This wrapper gathers call trees

```
fun callTreeWrapper f_ f callTreeRef x =
 let
  val myRef = ref [] (* empty list of children)
  val result = f_ (f myRef) x
  val _ = callTreeRef := ( (NODE(x, result, !myRef)) :: (!callTreeRef) )
 in
  result
 end
```

The wrapped seed takes one more argument than the original seed. After evaluation, the extra argument (a reference) contains the call tree. For example if we execute

```
val factCallTree = Y (callTreeWraper fact_) ;
val treeRef = ref []
val factOf10 = factCallTree treeRef 10
```

then `!treeRef` will be a list containing a single tree representing the computation of `fact 10`.

Some Theory — Sequential Realisability

The function `callTreeWrapper` cannot be written in a pure functional language (without changing the type of the seed so it takes a state monad). It is however a pure function in the sense that given the same pure functional argument, it will always return the same result.

Functions which are pure functions but which require non-functional features to be implemented lie in the interesting set of *sequentially realisable functions* [Lon99].

Conclusions

We have had a brief introduction to programming with Y and wrappers. How they work, some of their applications and some interesting properties. Overall, this demonstrates that fixed points are of more than theoretical interest in Computer Science. It is hoped that these ideas should be understandable to anyone with a knowledge of functional programming.

References

[EL00] Levent Erkök and John Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.

[FF95] Matthias Felleisen and Daniel P. Friedman. *The Little Schemer.* MIT Press, 3rd edition, dec 1995.

[Lon99] John Longly. When is a functional program not a functional program. In *International Conference on Functional Programming*, pages 1–7. ACM Press, 1999.

[McA97] Bruce J. McAdam. That about wraps it up — Using FIX to handle errors without exceptions, and other programming tricks. Technical Report ECS–LFCS–97–375, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, UK, November 1997.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised).* MIT Press, 1997.