Useful programming with FIX
Presentation to the ML-Club

Bruce J. McAdam

11th December 1997

Title Slide

Hullo!

Today we're going to look at a programming technique. I want to show how we can do fancy things in our programs without needing fancy new constructions in the language (exceptions, continuations, tools built into compilers). Higher-order programming lets us do a lot more than we may expect, and we probably don't make as much use of it as we should.

Background

I wanted my type-checker to recover from failure nicely and I wanted to be able to easily change how it recovers at a later date (without making too many changes to source code). Following the example of Ramsey (*Eliminating Spurious Messages*) I decided to add combinators to handle possible lack of information. Using `FIX` was the way to separate the algorithm from handling failure.

Introduction

We already know. . .

We can write higher-order functions.

This is useful, and one thing we especially like about ML.

We can write `FIX`.

We ask. . .

Is there any point in writing `FIX`?

We will discover. . .

This is useful as a practical programming technique.

It is interesting to compare the 'power' of different constructions.

Slide 1: Introduction

We can use `FIX` to customise and modify functions and we can simulate other program constructions.

It's good to know that we can make some use of unusual constructions. a lot of non-ML-programmers might say that ML is too powerful, but this is a vindication its abilities.

I always like to be able to say why something is of practical interest and why it is of interest from a theory perspective.

Getting functions in a FIX

Define FIX

```
fun FIX f x = f (FIX f) x
```

Define a ‘functional’

```
val fact_ = fn fact => fn i =>
  if i = 0 then 1 else i * (fact (i-1))
```

FIX the functional

```
val fact = FIX fact_
```

It’s just the same as

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

Slide 2: Getting functions in a `FIX`

This slide is just to remind people how to do this.

Alternatively we can define `FIX` without using `fun`.

I use the word 'functional' to mean a function I am going to fix and end functionals' names with a underscores. I write these without using `fun` to emphasise that they are not recursive.

The key thing `FIX` lets us do it to capture a single step of the computation, then turn it into a complete function later. This will allow us to make extensions to the single steps which will then happen inside the finished function.

It is not much harder to define functions this way then using `fun` (though mutual recursion is tricky).

Wrap up

This is a wrapper

```
val printer = fn f_ => fn f => fn i =>
  let
    val r = f_ f i
    val _ = print ((Int.toString i)^"\n")
  in
    r
  end
```

Wrap the functional and fix it

```
val fact' = FIX (printer fact_)
```

Try it out

```
> fact' 3 ;
1
1
2
6
val it = 6 : int
```

Slide 3: Wrap up

In general a wrapper is a function

- which creates a new functional from an old one

- which performs some secondary task (the main computation is still in the functional)

- which can be used on different functionals

Simple wrappers return a functional with the same type as before. The example is a simple wrapper of this type. I call these homogenous wrappers.

We can also make wrappers which change results from one type to another (heterogenous wrappers). See the slides on simulating exceptions for an example.

We can write wrappers which pass extra information (no examples here).

We cannot make the `fun` definition print intermediate results unless we edit the source.

Remember this!

Memoisation

Make functions remember previous results to save recomputing them.

We want a combinator to change a non-memo function into a memo function.

The obvious implementation doesn't remember recursive calls.

Use a wrapper

A wrapper can intercept recursive calls.

So we can use a wrapper to memoise recursive functions (as long as they are writing `FIX` style).

Other advantages

We can combine the memo wrapper with others.

Slide 4: Remember this!

Memoisation is an application of wrappers.

We can't memoise recursive functions because recursive calls will not be remembered. But using a `FIX` style definition allows the memoisation combinator to see inside the operation of the function.

The sample wrapper will memorise the result for parameters in the range 0–99. It would be possible to write a similar wrapper for any range of integers (or unlimited range), or for any other type (e.g. by using hash-tables).

Memoisation program

```
val memoise = fn f_ =>
  let
    val t = Array.array (100, NONE)
    val lookup = fn x => Array.sub (t, x)
    val store = fn (x, y) => (Array.update(t, x, SOME y); y)
  in
    fn f' => fn x =>
      if x>=0 andalso x<100 then (* if x in this range *)
        case lookup x of
          SOME y => y
        | NONE => store (x, f_ f' x)
      else
        f_ f' x
  end ;
val fact'' = FIX (memoise (printer fact_)) ;
```

Try it out. . .

```
> fact'' 3 ;
1
1
2
6
val it = 6 : int

> fact'' 3 ;
val it = 6 : int

> fact'' 4 ;
24
val it = 24 : int
```

Slide 5: Memoisation program

This is another homogenous wrapper, but unlike the previous example it does not always follow the execution path of the original fixed functional.

Consider what would happen if

```
printer (memoise fact_)
```

was used instead of

```
memoise (printer fact_)
```

The test shows that results are never recomputed.

I was really chuffed when I wrote this example, I knew that I needed a way to print newly computed results (to demonstrate that it really works) and it was reassuring to see that the technique I was writing about could solve my problem. I didn't even need to do any work as the printer wrapper was already compiled and loaded.

Exceptional!

We use exceptions a lot.

Sometimes this causes us to make mistakes

- We can forget handlers
- Handler code can't be sure where the exception originated
- It is harder to analyse code
- The type-system doesn't tell us anything about what exceptions can occur

(Of course the point of exceptions is that we are allowed to do these things, it's just that we don't want to do them accidently).

Simulate exceptions

We can simulate exceptions with datatypes.

Simulate handlers with wrappers (all 'exceptions' will be handled).

I tried this in a type-checker. . .

Slide 6: Exceptional!

The point of exceptions is that they can propagate anywhere, but often we want our programs to be more controlled, e.g. some exceptions should only be used between recursive calls, they should not appear outside the function.

Describe how to write the datatype and the wrapper. The wrapper is a heterogenous wrapper, it turns a type representing the union of valid results and possible exceptions into a valid result.

There is a similarity between the datatype and one way for giving semantics for exceptions. Also, recall thread on comp.lang.ml about using options.

The type-checker assures us that the exceptions are handled.

Wrappers help, it would be messy having all the code in one definition

We can try different wrappers (handlers).

The wrapper can *retry* if it receives an exception.

Important idea: avoid exceptions as they make it hard to understand your code (the require a better understanding of the semantics), instead use plain higher-order programming (so we only use a subset of the semantics).

Type-checker (edited highlights)

```
val W_ = fn W =>
    fn (A, Exp.ID i) =>
    let
        val tau = Option.map StatObjects.genInst (Assume.lookup i A)
    in
        case tau of
            SOME tau => OK(StatObjects.id, tau)
          | NONE => FAIL "Unbound identifier"
    end
     | (A, Exp.APP(e1, e2)) =>
           let
               val (S1, tau1) = W (A, e1)  val A' = assumeSubst S1 A
               val (S2, tau2) = W (A', e2) val tau1' = StatObjects.subst S2 tau1
               val beta = StatObjects.mkVarTy (StatObjects.newTyvar ())
               val funTy = StatObjects.mkFunTy(tau2, beta)
               val Vopt = StatObjects.U (tau1', funTy)
           in
               case Vopt of
                   SOME V =>
                       OK(StatObjects.compose
                           (V, (StatObjects.compose (S2, S1))),
                          StatObjects.subst V beta)
                 | NONE => FAIL ("Cannot apply\n  "^
                                 (StatObjects.stringTy tau1)^
                                 "\nto\n  "^(StatObjects.stringTy tau2))
           end
     | (A, Exp.ABS(x, e)) => ...
     | (A, Exp.LET(x, e1, e2)) => ...

val wrap = fn W_ => fn W' => fn p as (A, e) =>
        case W_ W' p of
            OK (S, tau) => (S, tau)
          | FAIL m => (print ("FAILURE in\n"^(Exp.toString e)^
                                     "\n"^m^"\n");
                              (StatObjects.id,
                               StatObjects.mkVarTy (StatObjects.newTyvar ())))

val W = Fix.fix (wrap W_)
```

Slide 7: Type-checker (edited highlights)

First I defined a datatype for the return type of the functional.

```
datatype 'a result =
    OK of 'a
  | FAIL of string
```

This represents the union of a valid result, or an 'exception' carrying a string. This could be extended, e.g. to have two types of failure.

Note that recursive calls do not process 'results', but that the functional returns a result. The functional cannot be fixed as it stands.

The wrapper inserts a *safe assumption* of the identity substitution and a new type variable so that type-checking can continue. I may want to change the wrapper to make a different assumption, or to ask the user if they want to continue (or something else).

Abstraction and declaration cases are missing from the functional but fairly obvious (look in the tech-report for the full program).

It didn't take very long to write this program (which was based on an earlier version) — about an afternoon.

Continuing. . .

Some uses of continuations can be expressed using wrappers.

I implemented an example of a producer-consumer pair.

This technique is suitable for tasks with a looping pattern (but can't replace all uses of continuations).

Program using NJ/SML continuations

```
datatype state = STATE of state cont ;

fun resume (STATE k : state) : state =
  callcc (fn k' : state cont => throw k (STATE k')) ;

val buf = ref 0 ;

fun produce(n : int, cons : state) =
  (buf := n ; produce(n+1, resume(cons))) ;

fun consume(prod : state) =
  (print ((Int.toString (!buf))^"\n") ; consume(resume(prod)))

fun pInit(n : int) : state =
  callcc(fn k : state cont => produce (n, STATE k))

fun pRun () = consume(pInit(0))
```

Slide 8: Continuing. . .

Lots of people like using continuations (we saw some last week).

I think that if we make better use of higher-order programming then we can largely do without them. This will make programs simpler.

The example is taken from a paper no continuations for SML *Typing first-class continuations in ML* Duba, Harper, MacQueen.

Before you look at the next slide, think about what the producer and consumer should be like if they are represented as functionals.

Example without continuations

```
val buf = ref 0 ;

val produce_ =
  fn produce => fn i =>
  (buf := i; produce (i+1)) ;

val consume_ =
  fn consume => fn () =>
  (print ((Int.toString (!buf))^"\n"); consume ()) ;

fun schedule p_ c_ pP cP =
      p_ (schedule' p_ c_ cP) pP
and schedule' p_ c_ cP pP =
      c_ (schedule p_ c_ pP) cP ;
```

Run it. . .

```
> (schedule producer_ consumer_ 1 ()) : unit
  (* will never terminate *) ;
1
2
3
4
```

Slide 9: Example without continuations

The producer and consumer should be exactly as you guessed.

I kind of cheat though, since I combined the wrapper and fixer into one function (which needed a mutually recursive partner).

The producer and consumer each have a 'state': for the producer this is an integer, for the consumer it carries no information. When calling for the next iteration the producer/consumer passes on the new state. This is why we need the polymorphic scheduler, it can fix a producer/consumer pair with any combination of types for states. If it was a single function, the producer and consumer would need the same types (since we don't have polymorphic recursion).

There is a further discussion of this in my tech-report.

How powerful?

I have demonstrated that plain higher-order programming can be used in place of some uses of other constructions.

This raises some questions

- Can we do away with some constructions?
- What techniques are more/less powerful than others?
- How do we express one construction in terms of another?

These are the sorts of issues that both language designers and programmers should consider.

Slide 10: How powerful?

(This slide is just something to think about).

Questions of theoretical interest.

There are a number of papers on subjects like these, e.g. showing that in a simply typed λ-calculus adding exceptions makes it Turing complete.

Programmers should consider questions like these so that they write better programs. Designers should consider these so that the languages they produce are more compact and usable.

Just because a technique gives us more expressibility, we don't need to use it all the time. (But, for example, continuations are useful as we saw last week.)

Conclusions

We often use fancy techniques where simpler ones are sufficient

- We rewrite and recompile where we don't need to
- We use exceptions where we should be more controlled
- We use continuations where they are not needed
- We use compiler tools when we can just add to the program

I think we should make better use of higher-order programming.

It is also intriguing to consider how many uses of other programming techniques can be replaced.

Read my tech-report (ECS–LFCS–97–375).

Slide 11: Conclusions

Byeee!

You can download the tech-report from

http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/97/ECS-LFCS-97-375/index.html

Papers mentioned here are in the bibliography. At some point I will get round to making a web page with links to related papers.