THAT ABOUT WRAPS IT UP

Using FIX to Handle Errors Without Exceptions, and Other Programming Tricks

BRUCE J. MCADAM*

Technical Report ECS–LFCS–97–375
Department of Computer Science
University of Edinburgh

November 28, 1997

Abstract

The Y combinator for computing fixed points can be expressed in Standard ML. It is frequently used as an example of the power of higher-order functions, but is not generally regarded as a useful programming construction.

Here, we look at how a programming technique based on the Y combinator and *wrappers* can give programmers a level of control over the internal workings of functions not otherwise possible without rewriting and recompiling code.

As an experiment, the type-inference algorithm W is implemented using this technique, so that error messages are produced with minimum interference to the algorithm. The code for this example program illustrates the genuine usefulness of the concepts and the ease with which they can be applied. A number of other implementation techniques and possible applications are also discussed, including the use of higher-order functions to simulate the use of exceptions and continuations.

*e-mail: bjm@dcs.ed.ac.uk, WWW: http://www.dcs.ed.ac.uk/home/bjm/

Contents

1 Introduction

The Y-combinator for computing fixed points of higher-order functions can be defined in Standard ML (SML) and other higher-order programming languages. This provides one way for programmers to define recursive functions. The ability to represent this combinator is an indication of the expressive power of these languages, but because these programming languages generally have an explicit recursive declaration it is not generally regarded as anything more than a curiosity with little practical value. In this report, I demonstrate a programming technique which makes use of the ability to define functions using Y.

I first describe how to implement and use this combinator in Standard ML in Section 2, and the general principles of how to use it in conjunction with *wrappers* to modify the workings of recursive functions in Section 3. At the end of this section I demonstrate how to use this technique to simulate some uses of exceptions and continuations.

The first application of the ideas described here was in implementing an error-reporting mechanism for Hindley-Milner type-inference [3] without using SML exceptions. Following the example of Ramsey [6] I chose to implement the error-reporting algorithm by adding combinators to an implementation of the algorithm without any mechanism for handling failure. The implementation of this application is described in Section 4. A summary of my conclusions appears in Section 5.

2 Defining and using the combinator in SML

The Y combinator was defined in the λ-calculus by Church [2] as shown in Figure 1.

Figure 1 The Y combinator defined for the λ-calculus.

$$Y \equiv \lambda f.(\ (\lambda f'.f(\lambda x.f'f'x))\ (\lambda f'.f(\lambda x.f'f'x))\ )$$

An explanation of why Y works (from a programmer's perspective) can be found in [5].

2.1 SML implementations

We will denote the SML version of the fix-point combinator by `FIX`, this will emphasise its use in programs and remind us that it is not exactly the same as Y — it will only be able to fix a certain form of function (which I call a 'functional') and is strictly evaluated.

`FIX` has the type shown in Figure 2.

Figure 2 The type of the fixed point combinator in SML.

```
val FIX : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

`FIX` can be applied to a 'functional' with a type of the form `(A -> B) -> A -> B` to return a function of type `A -> B`. The functional first takes the function whose least fixed point is to be computed, and then the parameter.

`FIX` can be defined using either a recursive definition or a non-recursive definition (which uses a recursive datatype). The latter allows us to write any recursive function in SML without using any explicit recursion (i.e. declarations of the form "`val rec` ..." or "`fun` ...").

2.1.1 FIX recursively

Defining `FIX` using SML's explicit recursive function declaration is quite simple as shown in Figure 3.

Figure 3 Defining `FIX` as a recursive function.

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

The function fixes the parameter `f`. The parameter becomes fixed when it receives its own fixed point as a parameter — so applying `(FIX f)` to `f` gives the required result.

The extra parameter to `FIX`, `x`, is the final parameter to the function. We must include this here as SML is a strictly evaluated programming language and so we must delay the evaluation of `FIX f` until it is applied to a value.

2.1.2 FIX non-recursively

Defining `FIX` without recursion is a much more complicated affair (it appears to be impossible to many SML programmers). The definition is based on Y for the λ-calculus which appeared in Figure 1. A direct implementation of the simple λ-calculus version will fail to type-check with an "unification occurs check" error — which happens when the unification function which forms the basis of ML type-checking finds that a circular substitution is necessary. The solution is to use a datatype to introduce a legal circular type and to add datatype construction and deconstruction to change values to and from this type. The implementation shown in Figure 4 is taken from [1].

Figure 4 Defining `FIX` with a recursive datatype.

```
local
  datatype 'a t = T of 'a t -> 'a
in
  val FIX = fn f => (fn (T x) => (f (fn a => x (T x) a)))
                    (T (fn (T x) => (f (fn a => x (T x) a))))
end
```

2.2 Using FIX

It is much more practical to use the directly recursive version of `FIX`. This is because just as it is easier for a programmer to see that it is a recursive function and that it finds fixed points, it is easier for an SML compiler to exploit this and optimise the resulting code. A discussion of optimisations for functions defined using the λ-calculus Y combinator in Scheme can be found in [7] (This discusses a scheme compiler, but the techniques apply equally well to SML).

To use `FIX` to define a recursive function we must first define a non-recursive version of the function which expects the fixed version as a parameter. As noted previously, I refer to these non-recursive higher-order functions as *functionals* and write them as in Figure 5.

Figure 5 Example *functional* for the factorial function.

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

I have written this using `fn` notation in order to emphasise that it is not recursive, and I have adopted the convention of giving such functionals names ending in underscores. The functional shown will be used to define the factorial function, as it takes a function called `fact` and integer `i` and if this parameter function can compute the factorial of smaller values then the final result will be the factorial of `i`.

To fix the functional `fact_`, simply apply the `FIX` combinator to it as in Figure 6.

Figure 6 Create the factorial function by fixing `fact_`.

```
val fact = FIX fact_ ;
```

The final function `fact` follows the same algorithm and computes the same results as the directly recursive definition in Figure 7.

Figure 7 The factorial function expressed using an SML `fun` declaration

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

3 Wrappers

The essence of the programming technique described here is to provide *wrappers* for the unfixed versions of functions which do some extra work for the function. Once the wrapped function is fixed the wrapper effectively intercepts every recursive function call.

I will describe two types of wrapper, the first is quite simple but the second provides us with a substantial level of programming power by allowing us to fix functionals which would not otherwise have well defined fixed-points because these would not have defined types under the language's type-system.

3.1 Simple wrappers

Unfixed functionals have types of the form `(A -> B) -> A -> B`. A simple wrapper takes a functional (perhaps for specific instances of `A` and `B`), and returns a new functional of the same type.

For example, we can write a wrapper for the `fact_` function which will allow us to see how it recurses. I define this wrapper in Figure 8 using the `fn` notation to make it clear that it is not recursive.

Figure 8 A simple wrapper.

```
val wrap = fn f => fn f' => fn p =>
  let
    val result = f f' p
  in
    print (Int.toString result); print "\n"; result
  end ;
```

Note that the use of `Int.toString` means that this wrapper is specific to functionals returning integers.

To make use of the wrapper we need only add one extra line to the previous program from Figure 6 (to apply the wrapper to `fact_`). The required definition is in Figure 9.

Figure 9 Using the simple wrapper on `fact_`.

```
val fact_' = wrap fact_ ;
val fact' = FIX fact_' ;
```

The resulting function `fact'` has the same type as the original `fact`, and was created without making *any* changes to `fact_` (or even needing to recompile the definition). Applying `fact'` gives the same answer as for `fact`, and it also prints a list of intermediate values, as shown in Figure 10.

Figure 10 An SML session showing use of `fact'`.

```
> fact' 10;
1
1
2
6
24
120
720
5040
40320
362880
3628800
val it : int = 3628800
```

It is clear that a wrapper like this could be useful when debugging functions since it provides an easy mechanism for tracing execution without having to insert `print` expressions inside the actual function definition.

3.2 Simple wrapper for memoisation

A popular programming technique is to write *memoised functions*. These functions store previous values they have calculated and return the memorised value instead of recomputing it when given the same parameter a second time.

Figure 11 show a wrapper for creating memoised functions.

Figure 11 Wrapper to memoise integer functionals. Only the results for the domain 0–99 are memorised but this could easily be changed.

```
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 (* Only use table if x in this range *)
        case lookup x of
          SOME y => y
        | NONE => store (x, f_ f' x)
        (* The new value is only computed if the table does not contain
           an entry.  The new value is stored in the table, and returned
         *)
      else
        f_ f' x (* Do not use the table as x is not in range *)
  end ;
```

Figure 12 creates a memoised version of the factorial functional and fixes it.

Figure 12 Memoised factorial function.

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

In order to find out whether or not this really works, we will use another neat trick — using two wrappers, one on top of the other. Figure 13 shows how to print the intermediate results of the memoised function.

Figure 13 Printing intermediate results shows that the memoisation wrapper really does work (output from an ML session). The wrapper `wrap` is the result printer from Figure 8.

```
> val fact''_ = memoise (wrap fact_) ;
val fact''_ : (int -> int) -> int -> int = fn
> val fact'' = FIX fact''_ (* This will print newly computed results *) ;
val fact'' : int -> int = fn
> fact'' 10 ;
1
1
2
6
24
120
720
5040
40320
362880
3628800
val it : int = 3628800
> fact'' 10 (* This should use previous results *) ;
val it : int = 3628800
> fact'' 12 (* This should start where fact'' 10 left off *) ;
39916800
479001600
val it : int = 479001600
```

Consider what would happen if the order of the wrappers was reversed such that `fact''_` was defined as `wrap (memoise fact_)`.

3.3 Complex wrappers — Heterogenous wrappers

It is possible to write wrappers which can turn some functionals which do not have fixed points into functionals with fixed points, the particular case here is that of wrappers which deal with functionals returning one type of data but expecting another from their 'recursive' calls.

3.3.1 Types of wrappers

The form of the type of a simple wrapper is shown in Figure 14[1].

[1]The example wrapper for printing integers actually has a weaker type than this, but we are only interested in applications of it to functionals.

Figure 14 The form of a simple wrapper's type. `P` is the type of the parameter (of the final fixed function), and `R` is its result.

```
((P -> R) -> P -> R) -> (P -> R) -> P -> R
```

A simple wrapper takes a functional and returns a functional of the same type as the original. The simple wrapper shown earlier was homogenous — the data passing into it had the same type as the data it returned. A heterogenous wrapper converts data from one type into another. This will allow us to manipulate types in such a way that a wrapped functional will have a fixed-point, but the original functional did not. The form of the type of the wrapper in this case is in Figure 15.

Figure 15 The form of the type of a *heterogenous wrapper*. The result of the functional is `R2`, which is different from that of the wrapped functional `R1`.

```
((P -> R1) -> P -> R2) -> (P -> R1) -> P -> R1
```

Note that the original functional passed to a heterogenous wrapper cannot have a fixed point (as it would not have a valid type), but the wrapped functional will have a fixed point.

3.3.2 Writing a heterogenous wrapper

One application of heterogenous wrappers is to convert a 'no answer' result from a function into a default or dummy response. The large example described in Section 4 uses such a wrapper. There I implement a type-checker which may return no result (if it cannot infer a type for its parameter), the wrapper announces the problem and provides a default result known not to interfere with the type-checking of other parts of the input.

3.4 Complex wrappers — adding extra information

It is possible to write a wrapper which can send information to recursive calls lying directly below it in the call tree. To do this, we add a parameter to the wrapper, e.g. as in Figure 16.

Figure 16 General form of a wrapper which adds extra information to recursive calls.

```
val wrap = fn f_ => fn f′ => fn extra => fn x =>
    f_ (f′ (foo extra)) x
```

The extra parameter is not seen by the functional, but is available to future instances of the wrapper. When we fix the wrapped functional, we must start it with an initial value as in Figure 17.

Figure 17 Using a wrapper which adds extra information.

```
val f_' = wrap f_
val f' = (FIX f_') initialExtra
```

Typically we might use this to pass information about branch of the call tree down, as shown in Figure 18.

Figure 18 Passing the call tree between instances of a wrapper.

```
val wrap = fn f_ => fn f' => fn callTreeBranch => fn x =>
    let
      val result = f_ (f' (x::callTreeBranch)) x
      (* We add the parameter to the call tree branch *)
    in
      (if isUnusual result then
         printMessage (callTreeBranch, result)
         (* The message can describe where in the computation the
            unusual result was found *)
       else ());
       result
    end
```

The wrapper in the figure prints a message if it encounters an unusual result. This message can include a list of the nested calls which led up to the result.

We could also use this method to detect looping in simple integer functions: the extra information would be a set of previous parameters and the wrapper can detect a loop by checking if the current parameter is already on the call-tree.

3.5 Alternative implementations for wrapped functions

It is possible to implement wrapped functions using more conventional programming techniques. As an example have a look at the definition in Figure 19.

Figure 19 `fact'` (of Figure 9) defined without a wrapper.

```
fun fact' i =
  let
    val result = if i = 0 then 1 else i * fact (i-1)
  in
    print (Int.toString result); print "\n"; result
  end
```

This function behaves identically to the first wrapped factorial function shown earlier in Figure 9. The code contains all the main constructions of the wrapped functions and maintains the separations between the computation and the printing routine so the workings of it should be equally clear to a programmer.

One advantage in my approach over this definition of `fact'` is that, given the definition of `fact_`, it is possible to use different wrappers without editing and recompiling the function body. Another advantage was shown in Figure 13 where I used two wrappers and a functional which I had previously compiled, the wrappers gave me an easy way to try different combinations of features. Where a function is extended in a complex way (for example if memoisation was implemented for more complex parameter types by using hash tables or search trees) the same wrapper can be used for many functions without the need to make many changes to the program (and it should be remembered that changing source code is an easy way to introduce errors). SML also allows us to change between many different wrappers (or no wrapper) and to generate wrappers on the fly — something which would not be possible with the monolithic definition above.

3.6 Simulation of other program constructions

Programming languages typically provide constructions so that programmers can control their functions' behaviour. Examples of such constructions include recursive function definitions, continuations and exceptions. We have already seen how `FIX` can give programmers the same functionality as SML's "`fun` ..." declarations. In this section we will see how `FIX` and wrappers can simulate some uses of exceptions and continuations.

3.6.1 Exceptions

In the next section, I will show how to use wrappers to simulate a possible use of exceptions. Figure 20 show the general form of a wrapper simulating exception handlers and the datatype associated with it.

Figure 20 Simulating exception handlers with a wrapper. We create a datatype representing the possible results (a value or one of a number of exceptions), the wrapper converts the result into a value and performs housekeeping (e.g. reporting errors to the user).

```
(* We know in advance all the exceptions used inside the functional,
   so we can define these (and OK) as a datatype. *)

datatype 'a result =
  OK of 'a
| EXN_1 of exn_1_data
| EXN_2 of exn_2_data
| ...
| EXN_n of exn_n_data

(* The handleWrapper uses a number of functions each of which
   can transform `exception' data into a dummy result.  The
   handleExn functions will also inform the user or do any other
   housekeeping necessary. *)

val handleWrapper = fn f_ => fn f' =>
  fn OK r => r (* no `exception', return result *)
   | EXN_1 e => handleExn1 e (* returns data the same type as r *)
   | EXN_2 e => handleExn2 e
   | ...
   | EXN_n e => handleExnn e

(* Define the functional so it expects a value as the result of
   recursive calls, but returns data using one of the constructors
   for the result *)
```

If there are no exceptions then it is easier to reason about the program since we do not need to consider the complications caused when the order of execution is disrupted. A particular advantage is that we can be sure that the 'exception' never travels further than to the handler in the caller (represented by the wrapper), whereas if actual exceptions were used it would be easy to make a mistake in the program leading to the exception either not being caught or being caught by the wrong handler. This second reason is important as we often need an exception to be caught by a particular handler (especially if the exception carries information). We cannot make the mistake of forgetting a handler (represented by the wrapper) in this case as it would prevent the program from type-checking.

While wrapping functionals cannot capture every possible use of exceptions, it can simulate a wide range of applications. By using this technique, we can save exceptions for *exceptional* circumstances where they really are required (such as terminating a program completely because of an operating system error).

3.6.2 Continuations

Wrappers can control execution paths similarly to some uses of continuations. It is possible to simulate some applications of continuations on languages without continuations (e.g. SML without the extensions added to Standard ML of New-Jersey (NJ/SML)) using this technique. I will show how it is possible to implement an example from the paper introducing continuations for SML (as implemented in NJ/SML) [4]. The example written using continuations is shown in Figure 21, it consists of a producer and consumer pair communicating through a shared reference.

Figure 21 A producer-consumer program written using continuations for NJ/SML, taken from [4]. The functions `produce` and `consume` will alternate when `pRun` is called. Note that the producer and consumer are ignorant about what a state actually is.

```
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))
```

Besides the ‘state’ (containing a continuation), the producer has a value which it can pass to the future calls to itself. The consumer has no data, but the program could be changed so it can pass a value to its next incarnation.

In my version of this communicating system, the producer and consumer are defined as a pair of functionals. The producer will pass an integer when it makes a recursive call, the consumer will pass the unit value written as ‘`()`’ in SML. I will write my scheduler in such a way that it is polymorphic on the types of values passed on by the producer and consumer. The implementations of the producer and consumer appear in Figure 22.

Figure 22 A one place buffer and the producer and consumer functionals.

```
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 ()) ;
```

The functional `produce_` expects to receive a function which will do any other scheduled tasks (in this case this is the consumer) and then return to the producer with the new value. Similarly for the consumer functional, though it does not pass on a meaningful value.

The scheduler appears in Figure 23, instead of using `FIX` a pair of mutually recursive functions perform both the wrapping and fixing of the functionals.

Figure 23 A scheduler. For clarity we combine wrapping and fixing in a pair of mutually recursive function definitions. SML's type system prevents us from using a single function definition as there is no polymorphic recursion (since the values associated with the producer and consumer differ).

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

A pair of functions is required to fix the two functionals as these pass on two different types. The scheduler works for any pair of functionals with types of the form `(A -> A) -> A -> A` where `A` is the type of value passed on to the next instance.

To run the scheduled processes pass the two functionals and two starting values to the scheduler, as shown in Figure 24.

Figure 24 SML session running the producer and consumer. The type constraint is required as otherwise this (non-terminating) function would produce a result of polymorphic type, SML's type system does not allow this.

```
> (schedule producer_ consumer_ 1 ()) : unit ;
1
2
3
4
```

...the program never terminates.

To demonstrate the flexibility of this way of scheduling processes, I define a different consumer in Figure 25, in this example both the producer and consumer pass on information (of different types), only the new process and the compound process need to be compiled to run the new system.

Figure 25 An alternative consumer which can be scheduled using `schedule` of Figure 23. This gathers up the values from the producer into a list. When running, the final parameter to the scheduler should be the empty list instead of `()`.

```
val consumer2_ = fn consumer2 => fn l =>
  consumer2 ((!buf)::l)
```

A criticism which may be levied against my version of the producer-consumer program is that it relies on a mutually recursive pair of functions and so cannot easily be extended to handle any number of processes. The same argument holds against the continuation based program in Figure 21, in fact the `pRun` and `pInit` functions of the scheduler would have to be rewritten if we wished to change the type of the consumer. In general however it is easier to deal with large numbers of processes using continuations, and the lack of polymorphic recursion in SML proves a limitation when writing such programs as functionals.

4 Application to a type-checker

I developed the idea of using wrappers while writing an extension to the Hindley-Milner type-inference algorithm, W. In this section I will describe my program, and how and why it was written. A listing of the final program and examples of its use can be found in Appendix A.

4.1 Starting point

4.1.1 Algorithm W

W is a type inference algorithm for Hindley-Milner type systems such as that of SML. I started with an implementation of algorithm W(written to be as close as possible to the description in [3]). The algorithm takes a λ+`let`-calculus expression together with a set of type assumptions for free identifiers in the expression, and produces an infered type for the expressions together with a set of type assumptions (represented as a substitution from type-variables to types). The algorithm recursively calls itself on subexpressions and can fail if the results from two subexpressions do not fit together (this is generally called a 'unification' error because of the unification subroutine used to combine two types).

4.1.2 My Implementation

The original version of my function was defined using an SML `fun` declaration. When the function failed to find a type for some subexpression, it simply raised an exception causing it to terminate.

4.2 Required extension

My objective was to modify the program so that when encountering a subexpression with no type, instead of raising an exception a message is printed to inform the user and type-checking continues on the remainder of the program (possibly printing more error messages).

I wanted to implement this in a flexible way, for example so that I could substitute a more complex function in place of the simple error reporter (perhaps to print more information about the error, or to select only a subset of the errors to be reported). I also wanted to implement this in a pure functional style (with no exceptions or references). I wish the program to avoid imperative features as I intend to extend it further and to be able to study the algorithm without complications.

4.3 Modified implementation

I chose to implement the changes to my program using combinators as much as possible, so that the underlying algorithm could still be clearly seen and the combinators can be removed to retrieve the original code. For example, if my implementation involved the `option` type to handle missing information, I would have used combinators such as `Option.map` and `valOf` to keep the code as clean as possible. In this respect, I follow the example of Ramsey [6].

I decided to change my code to a functional version suitable for the fixed point combinator (this was a simple change to make, essentially involving only a change from "`fun W...`" to "`val W_ = fn W =>...`"). This allowed me to insert other code (a wrapper) to deal with errors, I realised this would help because there should be a maximum of one error reported for every call to W.

My first attempt used a homogenous wrapper. The wrapper only printed messages, and the main functional had to deal with a datatype which could represent missing information. This implementation seemed unsatisfactory as there were many changes to the main functional.

In the second version (shown in Appendix A along with examples of its results), `W_` expected to receive a value on recursive calls but returned an option (e.g. it could return `NONE` if it failed to unify two types). The heterogenous wrapper turns the type-checker into a *total* function by returning the pair $\langle Id, \alpha \rangle$ (α is new) when no type can be derived. This pair is a 'safe' assumption as it will never create a conflict with another type (it can be unified with any type).

The function does not produce better error reporting than implementations of this algorithm found in compilers, but it should be easier to modify in the future.

A further change I may make is to split the wrapper in two. One wrapper could provide the assumed value, and the other could print the message. This would make it easy to study different ways of doing each of these tasks.

4.4 Relevance to other applications

I had very specific aims when I implemented this variation of my type-inference function. While my own situation may differ from that of a typical programmer working on a software engineering project, the ability to make changes to code in such a flexible way while producing code which is possible to reason about is important to everyone.

5 Conclusions

I have demonstrated that by implementing a recursive function using the fixed-point combinator we can easily write 'wrappers' to intercept recursive calls and perform useful operations on the data flowing in and out (e.g. printing trace messages or replacing null results with assumed values). Moreover, we can use a wrapper to change the data (and even the type of data) returned from 'recursive' calls and to make modifications to the algorithm (such as memoisation).

Wrappers are easy to implement and use, and can be repeatedly applied to functionals without recompiling the functional or wrapper. We can therefore easily test different wrappers and combinations of wrappers without the time consuming (and possibly error-prone) tasks of rewriting and recompiling programs. We can also use the same wrapper on a number of different functionals, for example to write a number of different memoised functions. Because of this flexibility, wrappers are ideal not only for adding to the functionality of programs, but also as software engineering tools.

The technique has been shown to be useful and easy to use by the conversion of a simple type-checker into one which prints error messages, and its flexibility has been shown through a series of examples in which the factorial function is modified in different ways (such as making it print intermediate results or memorise results).

Though more work is needed to devise a consistent method of providing total control over the execution of recursive functions (e.g. to allow backtracking; upwards propagation of information about the call tree; and to simulate some complex applications of continuations) and the lack of polymorphic recursion in SML could prove to be a major limitation, examples included here demonstrate that basic higher-order programming techniques (wrappers and the Y combinator) are powerful enough to simulate a wide range of situations in which exceptions or continuations might be used. Avoiding the used of these control structures and using only higher-order functions produces programs which are easier to analyse.

A Implementation of a type-checker

A.1 Program code

The original version of `W` was a simple recursive function defined using `fun`. Whenever the algorithm failed in the original, an exception was raised.

In the new version, `W` is defined via a non-recursive functional `W_` which expects a *total* function as its first parameter but is itself only *partial* (possibly returning only a description of why it has failed in place of infered information). The wrapper turns the partial function into a total function by providing a 'safe' assumed value.

A.1.1 The main functor

```
functor Infer (structure StatObjects : STAT_OBJECTS
                     structure Assume : ASSUME
                     structure Exp : EXP
                     sharing type StatObjects.ty = Assume.ty
                         and type Assume.id = Exp.id
                         and type Assume.scheme = StatObjects.scheme )
    : INFER =
  struct

    type exp = Exp.exp
    type scheme = StatObjects.scheme
    type assume = Assume.assume

    (* Instead of returning options, we return a result which could
       contain a message. *)
    datatype 'a result =
        OK of 'a
      | FAIL of string

    val assumeSubst = fn S => Assume.map (StatObjects.substSch S)

    (* W_ expects (subst * ty) option as the result of recursive calls,
       but returns (subst * ty, string) result. *)

    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)) =>
           let
               val beta = StatObjects.mkVarTy (StatObjects.newTyvar ())
               val (S, tau) =
                   W(Assume.append(A, (x, StatObjects.forall ([], beta))),
                     e)
           in
               OK(S,
                         StatObjects.subst S
                         (StatObjects.mkFunTy(beta, tau)))
           end
     | (A, Exp.LET(x, e1, e2)) =>
           let
               val (S1, tau1) = W(A, e1)
               val tau1' = Assume.closure (assumeSubst S1 A) tau1
               val (S2, tau2) = W(assumeSubst S1
                                  (Assume.append (A, (x, tau1'))),
                                 e2)
           in
               OK(StatObjects.compose (S2, S1), tau2)
           end

(* The wrapper takes the result of W_ and
```

```
      Prints failure message (if necessary),
      returns an option. *)
   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 ())))

   (* It would be possible to use a more complex wrapper to do something
      more complex here, and my intention is to improve this wrapper
      at a later date (e.g. to calculate a better assumption and/or
      provide different information to the user). *)

   val W_' = wrap W_
   val W = Fix.fix W_'

   (* infer is just a 'user interface' wrapper, which converts the
      (S, tau) pair into a single type-scheme *)

   val infer = fn (A, e) =>
                 let
                   val _ = StatObjects.reset()
                 in
                     (fn (S, tau) => Assume.closure (assumeSubst S A) tau)
                     (W (A, e))
                 end

  end
```

A.1.2 Signatures

These signatures are required by the functor.

```
signature STAT_OBJECTS =
  sig

    eqtype tyvar
    val newTyvar : unit -> tyvar
    val reset : unit -> unit (* Reset tyvars produced *)

    eqtype tycon
    val tycon : string -> tycon
```

```
    eqtype ty
    val mkVarTy : tyvar -> ty
    val mkFunTy : (ty * ty) -> ty
    val mkConTy : ty list * tycon -> ty

    eqtype scheme
    val forall : (tyvar list * ty) -> scheme
    val genInst : scheme -> ty

    val tyvars : ty -> tyvar list
    val freeTyvars : scheme -> tyvar list

    val stringTy : ty -> string
    val stringScheme : scheme -> string

    type subst

    val id : subst

    val subst : subst -> ty -> ty

    val substSch : subst -> scheme -> scheme

    val compose : (subst * subst) -> subst

    val U : ty * ty -> subst option

  end


signature ASSUME =
  sig

    type assume

    val empty : assume

    type id

    type ty
    type scheme

    val lookup : id -> assume -> scheme option

    val append : (assume * (id * scheme)) -> assume
```

```
    val closure : assume -> ty -> scheme

    val map : (scheme -> scheme) -> assume -> assume

    val basis : assume

  end

signature EXP =
  sig

    eqtype id

    val idToString : id -> string
    val idFromString : string -> id

    datatype exp = ID of id
                 | ABS of (id * exp)
                 | APP of (exp * exp)
                 | LET of (id * exp * exp)

    val toString : exp -> string

  end
```

A.2 Results

A.2.1 Example causing one failure

Input program

```
let
  func = fn i => fn j => i j
in
  let
    value = one
  in
    func value
  end
end
```

Output

```
FAILURE in
( func) (value)
Cannot apply
```

```
   ( ( 'A4 ) -> ( 'A3 ) ) -> ( ( 'A4 ) -> ( 'A3 ) )
to
   int
```

Explanation `func` expects a function as its parameter, but `value` is an integer.

A.2.2 Example causing multiple failures in different declarations

Input program

```
let
  func = fn i => fn j => i i
in
  let
    value = one one
  in
    func value
  end
end
```

Output

```
FAILURE in
( i) (i)
Cannot apply
  'A0
to
  'A0
FAILURE in
( one) (one)
Cannot apply
   int
to
   int
```

Explanation The first failure occurs because a function's parameter cannot be applied to itself. After finding the error in the first declaration, the program goes on to check the rest of the program (finding the second error as it does this). The assumed types for `func` and `value` ensure that no errors are announced for the final expression.

A.2.3 Example with two errors in one application

Input

```
pair (cons nil zero) (succ nil)
```

Output

```
FAILURE in
( ( cons) (nil)) (zero)
Cannot apply
  (   'A3 list list ) -> (   'A3 list list )
to
   int
FAILURE in
( succ) (nil)
Cannot apply
  (  int ) -> (  int )
to
   'A8 list
```

Explanation The list constructor, `cons`, expects an item followed by a list (this is the cause of the first error). The successor function cannot be applied to the empty list, `nil`. Even though both errors are in the same expression (not separated by declarations or abstraction), they are both detected independently.

References

[1] Dave Berry, Greg Morrisett, and Rowan Davies. `comp.lang.ml` Frequently Asked Questions and Answers.
Available from `ftp://pop.cs.cmu.edu/usr/rowan/sml-archive/faq.txt`, 1997.

[2] Alonzo Church. *The calculi of lambda conversion*. Princeton University Press, 1941.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual Symposium on Principles of Programming Languages*. Association of Computing Machinery, ACM Press, 1982.

[4] Bruce F. Duba, Robert Harper, and David MacQueen. Typing First-Class Continuations in ML. In *18th Symposium on Priniciples of Programming Languages*, pages 163–173. Association of Computing Machinery, ACM Press, 1991.

[5] Matthias Felleisen and Danial P. Friedman. $(Y\ Y)$ Works! A lecture on the *Why* of Y, 1991.

[6] Norman Ramsey. Eliminating spurious messages. Technical Report CS–97–06, Department of Computer Science, University of Virginia, 1997.

[7] Guillermo Juan Rozas. Taming the Y combinator. In *ACM Conference on Lisp and Functional Programming*. Association of Computing Machinery, 1992.