J. Functional Programming **8** (2): 195–199, March 1998. Printed in the United Kingdom

FUNCTIONAL PEARL

Even higher-order functions for parsing
or
Why would anyone ever want to use a sixth-order function?

CHRIS OKASAKI*

School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(*e-mail:* cokasaki@cs.cmu.edu)

1 Introduction

A *higher-order* function is a function that takes another function as an argument or returns another function as a result. More specifically, a *first-order* function takes and returns base types, such as integers or lists. A kth-order function takes or returns a function of order $k - 1$. Currying often artificially inflates the order of a function, so we will ignore all inessential currying. (Whether a particular instance of currying is essential or inessential is open to debate, but we expect that our choices will be uncontroversial.) In addition, when calculating the order of a polymorphic function, we instantiate all type variables with base types. Under these assumptions, most common higher-order functions, such as *map* and *foldr*, are second-order, so beginning functional programmers often wonder: What good are functions of order three or above? We illustrate functions of up to sixth-order with examples taken from a combinator parsing library.

Combinator parsing is a classic application of functional programming, dating back to at least Burge (1975). Most combinator parsers are based on Wadler's list-of-successes technique (Wadler, 1985). Hutton (1992) popularized the idea in his excellent tutorial *Higher-Order Functions for Parsing*. In spite of the title, however, he considered only functions of up to order three.

2 Parsers as third-order functions

Using Wadler's list-of-successes technique, a parser is represented by a first-order function that takes a lazy list of tokens and returns a lazy list of partial results.

* This research was sponsored by the Advanced Research Projects Agency CSTO under the title 'The Fox Project: Advanced Languages for Systems Software', ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

https://doi.org/10.1017/S0956796898003001 Published online by Cambridge University Press

The laziness in the return value is essential for controlling backtracking. However, lazy lists are awkward in a strict language like SML (Milner *et al.*, 1990), so we will instead manage backtracking using explicit success and failure continuations. We sketch only those details that are relevant to illustrating higher-order functions.

A parser is a function that takes four arguments: a success continuation that is invoked when the parser succeeds, a failure continuation that is invoked when the parser fails, a list of tokens, and the current line number (or other error reporting information). In SML, we can write this type as

```
type 'a Parser = 'a SuccCont * FailCont * Token list * Line -> Ans
```

The type `'a` represents the result of the current parser, often a fragment of an abstract syntax tree. We leave the exact forms of `Token`, `Line`, and `Ans` unspecified. We also leave unspecified the details of managing the current line number.

A failure continuation takes the line number where the failure occurred and produces an answer.

```
type FailCont = Line -> Ans
```

A success continuation is more complicated. At the very least, it takes the result of the current parser, the list of remaining tokens, and the line number of the next token. To support full backtracking, however, it must also take the current failure continuation in case a later failure backtracks to this point.

```
type 'a SuccCont = 'a * FailCont * Token list * Line -> Ans
```

Now, failure continuations are first-order. Success continuations take failure continuations as arguments, so they are second-order. Parsers take success continuations as arguments, so they are third-order.[1]

Most primitives involving parsers either take or return parsers and are therefore at least fourth-order. However, there are two primitive parsers that are third-order. The first is the parser that always fails, the second is the parser that reads the next token.

```
(* fail : 'a Parser *)
fun fail (sc,fc,ts,n) = fc n

(* any : Token Parser *)
fun any (sc,fc,[],n) = fc n
  | any (sc,fc,t::ts,n) = sc (t,fc,ts,n)
```

3 Fourth-order functions

Most combinators that produce or manipulate parsers are fourth-order. The simplest is the parser that always succeeds, consuming no tokens. This combinator takes an

[1] Recall that, using the list-of-successes approach, parsers are first-order functions from lazy lists to lazy lists, so switching to the continuation-passing approach instantly raises the order of most parsing combinators by at least two orders.

https://doi.org/10.1017/S0956796898003001 Published online by Cambridge University Press

argument that is the value to pass to the success continuation as the result of the parse.

```
(* succeed : 'a -> 'a Parser *)
fun succeed x (sc,fc,ts,n) = sc (x,fc,ts,n)
```

This function is fourth-order because it returns a third-order parser.

In general, fourth-order combinators may both take and return third-order parsers. For example, here are combinators for sequencing and alternation.

```
(* seq : 'a Parser * 'b Parser -> ('a * 'b) Parser *)
fun seq (p,q) (sc,fc,ts,n) =
  let fun scp (x,fc,ts,n) =
        let fun scq (y,fc,ts,n) = sc ((x,y),fc,ts,n)
        in q (scq,fc,ts,n) end
  in p (scp,fc,ts,n) end

(* alt : 'a Parser * 'a Parser -> 'a Parser *)
fun alt (p,q) (sc,fc,ts,n) =
  let fun fcp np =
        let fun fcq nq = fc (max (np,nq))
        in q (sc,fcq,ts,n) end
  in p (sc,fcp,ts,n) end
```

Note that `max` is used to combine the line numbers of two failures, under the assumption that the most likely location for an error is at the end of the longest successful parse.

These functions are rather messy. However, only a handful of combinators need know the internal representation of parsers—the remaining combinators can be built from these primitive combinators. In addition, many of the primitive parsers can be simplified by assuming that the functions representing parsers and the functions representing success continuations are curried. Then, for instance, the sequencing combinator can be written

```
fun seq (p,q) sc = p (fn x => q (fn y => sc (x,y)))
```

4 Fifth-order functions

The above functions are all fourth-order. Can we go higher? Easily. Here is a fifth-order function, where the result of one parser is used to choose the next parser. For those readers familiar with monadic programming, this is just the *bind* operation from the monad of parsers (Wadler, 1992; Hutton and Meijer, 1996).

```
(* bind : 'a Parser * ('a -> 'b Parser) -> 'b Parser *)
fun bind (p,f) sc = p (fn x => f x sc)
```

The function `f` is fourth-order, so `bind` is fifth-order. (We have again written this function assuming that parsers and success continuations are curried.)

The `bind` combinator is extremely powerful and can be used to define many other useful combinators. For example, the sequencing combinator can be rewritten as

https://doi.org/10.1017/S0956796898003001 Published online by Cambridge University Press

```
fun seq (p,q) = bind (p,fn x => bind (q,fn y => succeed (x,y)))
```

This version of seq assumes nothing about the internal representation of parsers.

A more interesting example of bind uses a predicate to filter the results of a parser.

```
(* filter : 'a Parser * ('a -> bool) -> 'a Parser *)
fun filter (p,f) = bind (p,fn x => if f x then succeed x else fail)
```

Another useful fifth-order function is lookahead, which is similar to bind but restores any tokens consumed by the first parser.

```
(* lookahead : 'a Parser * ('a -> 'b Parser) -> 'b Parser *)
fun lookahead (p,f) (sc,fc,ts,n) =
  let fun scp (x,fc,_,_) = f x (sc,fc,ts,n)
  in p (scp,fc,ts,n) end
```

5 A sixth-order function

Still higher? Here is a sixth-order function – the *join* operation from the monad of parsers – where the result of the current parser is itself the next parser to use.

```
(* promote : 'a Parser Parser -> 'a Parser *)
fun promote p = bind (p,fn q => q)
```

Why is this sixth-order? The argument p is fifth-order because its success continuation is fourth-order. Its success continuation is fourth-order because it takes an ordinary (third-order) parser as the result of p.

This combinator is useful in at least two situations. The first is when one parser returns a value that chooses between several successor parsers in a bind, as in

```
bind (p,fn x => if x then p1 else p2)
```

By rewriting p to return p1 or p2 directly instead of true or false, and replacing the call to bind with

```
promote p
```

we can often avoid an extraneous branch.

The second application occurs when parsing a language in which programs include a prelude that affects how the rest of the program is to be parsed – the result of parsing the prelude is the parser to be used on the rest of the program. Fixity declarations constitute a simple example of this phenomenon, but it is easy to imagine a language that allows even more radical redefinitions of its syntax.

References

Burge, W. H. (1975) *Recursive Programming Techniques.* Addison-Wesley.

Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming* **2**(3), 323–343.

https://doi.org/10.1017/S0956796898003001 Published online by Cambridge University Press

Hutton, G. and Meijer, E. (1996) Monadic Parsing Combinators. *Technical Report NOTTCS-TR-96-4*, Department of Computer Science, University of Nottingham.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

Wadler, P. (1985) How to replace failure by a list of successes. *Conference on Functional Programming Languages and Computer Architecture*, pp. 113–128.

Wadler, P. (1992) The essence of functional programming. *ACM Symposium on Principles of Programming Languages*, pp. 1–14.

https://doi.org/10.1017/S0956796898003001 Published online by Cambridge University Press