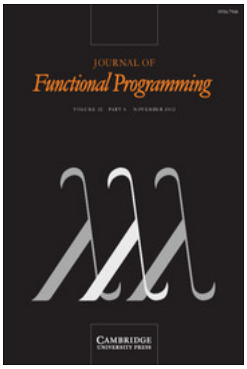

When Maybe is not good enough

MICHAEL SPIVEY

Journal of Functional Programming / Volume 22 / Issue 06 / November 2012, pp 747 - 756
DOI: 10.1017/S0956796812000329, Published online: 10 September 2012

Link to this article: http://journals.cambridge.org/abstract_S0956796812000329

How to cite this article:
MICHAEL SPIVEY (2012). When Maybe is not good enough. Journal of Functional Programming, 22, pp 747-756 doi:10.1017/S0956796812000329

JFP **22** (6): 747–756, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000329 First published online 10 September 2012

FUNCTIONAL PEARL
When Maybe is not good enough

MICHAEL SPIVEY
Department of Computer Science, University of Oxford, Oxford, UK
(*e-mail:* mike@cs.ox.ac.uk)

1 Introduction

Many variations upon the theme of parser combinators have been proposed, too many to list here, but the main idea is simple: A parser for phrases of type α is a function that takes an input string and produces results $(x, rest)$, where x is a value of type α, and *rest* is the remainder of the input after the phrase with value x has been consumed. The results are often arranged into a list, because this allows a parser to signal failure with the empty list of results, an unambiguous success with one result, or multiple possibilities with a longer 'list of successes'.

$$\textbf{type}\ \mathit{Parser}_1\ \alpha = \mathit{String} \rightarrow [(\alpha, \mathit{String})].$$

This type admits operations $\gg$ for sequencing and $\oplus$ for alternation that make it natural to translate any non-left-recursive context-free grammar directly into a parser.

Producing a list of results naturally leads to backtracking parsers that can be exponentially slow, so it is preferable where possible to substitute a different parser type,

$$\textbf{type}\ \mathit{Parser}_2\ \alpha = \mathit{String} \rightarrow \mathit{Maybe}\ (\alpha, \mathit{String}).$$

As we shall see, parsers with this type do not allow the kind of *deep backtracking* in which a parser that has produced one result can be asked to produce another one. They do, however, provide something weaker that we might call *shallow backtracking*, where a parser $xp \oplus yp$ produces a single positive result if either xp or yp would succeed on its own given the original input. Provided the grammar allows it, using this parser type reduces the amount of fruitless searching, and permits the record of choices made in recognising a phrase to be discarded as soon as one of the choices succeeds.

If parsers based on *Maybe* are preferable to those based on lists, it is natural to ask what grammars they can parse properly. With an unambiguous grammar, an input string will either fail to be in the language, or will have exactly one derivation tree. We shall say that a parser *works correctly* if it has type $\mathit{Parser}_1\ \alpha$ and in these two cases returns [ ] and [(x, “ ”)] respectively (for some value $x :: \alpha$), or if it has type $\mathit{Parser}_2\ \alpha$ and it returns *Nothing* and *Just* (x, “ ”) in the two cases.

It is not hard to show that the list-based parser works correctly for any grammar that has no left recursion. At the other extreme, grammars that are $LL(1)$ can be parsed with no backtracking at all, and for them both list-based and *Maybe*-based parsers work correctly. As we shall shortly see, it is useful to allow grammars that are ‘not quite $LL(1)$’, so that the question arises whether a *Maybe*-based parser will continue to work correctly for them. The result reported in this paper is that this question is not decidable by any general algorithm. The essence of this result has been known for a surprisingly long time, and the contribution of the present paper is to present it afresh in the context of parser combinators. Although, like all undecidability results, this result has a negative character, we shall be able to rescue positive aspects of it by animating the construction in its proof as a functional program.

Although the discussion in this paper is phrased in terms of monadic parser combinators because of their popularity, in fact we nowhere exploit the possibility that, in a compound parser $xp \ggg\!= (\lambda x \rightarrow yp)$, the *syntactic* behaviour of yp may depend on the value x returned by xp. Because of this, the results apply equally well to more restrictive frameworks such as applicative functors (McBride & Paterson, 2008).

2 Parser combinators

We shall want to experiment with both parsers that return a list of results and parsers that use *Maybe* instead. Luckily, the type class system of Haskell allows us to describe parser combinators in a way that is independent of the monad m that is used to deliver the results. As usual, we need to wrap the parser type in a **newtype** construction, so we can make it belong to various type classes.

$$\textbf{newtype}\ Parser\ m\,\alpha = Parser\ \{runParser :: String \rightarrow m\,(\alpha, String)\}.$$

The type $Parser\ m\,\alpha$ contains parsers that accept a string and deliver results of type α using the monad m. If m is indeed a monad, then so is $Parser\ m$.

```
instance Monad m ⇒ Monad (Parser m) where
  return x = Parser (λs → return (x, s))
  xp ≫= f =
    Parser (λs → runParser xp s ≫= (λ(x, s′) → runParser (f x) s′)).
```

As always with monads, we may use the notation $xp \gg yp$ as an abbreviation for $xp \ggg\!= (\lambda x \rightarrow yp)$ when the value x returned by parser xp is not used by yp.

Haskell’s type class *MonadPlus* describes monads that additionally provide a constant *mzero* and a binary operation *mplus*, which we here write as $\oplus$.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m α
  (⊕) :: m α → m α → m α.
```

Both the list type constructor [ ] and the *Maybe* constructor are declared in the standard library as instances of *MonadPlus*.

instance *MonadPlus* [ ] **where**
  $mzero = [\,]$
  $(\oplus) = (+\!\!+)$

instance *MonadPlus Maybe* **where**
  $mzero = Nothing$

  $(Just\ x) \oplus ym = Just\ x$
  $Nothing \oplus ym = ym.$

There are several equational laws that are satisfied by both of these instances of *MonadPlus*: for example, ⊕ is associative and has *mzero* as a unit element. But there is one significant law that is satisfied by the list monad and not by *Maybe*.

$$(xm \oplus ym) \gg\!= f \;=\; (xm \gg\!= f) \oplus (ym \gg\!= f). \qquad (*)$$

A simple example shows this: if we define

  $guard :: MonadPlus\ m \Rightarrow (\alpha \rightarrow Bool) \rightarrow \alpha \rightarrow m\,\alpha$
  $guard\ p\ x =$ **if** $p\ x$ **then** $return\ x$ **else** $mzero,$

and let $even :: Int \rightarrow Bool$ be the obvious predicate, then the expression

$$(return\ 1 \gg\!= guard\ even) \oplus (return\ 2 \gg\!= guard\ even)$$

evaluates to $Nothing \oplus Just\ 2 \;=\; Just\ 2$, but the expression

$$(return\ 1 \oplus return\ 2) \gg\!= guard\ even$$

evaluates to $Just\ 1 \gg\!= guard\ even \;=\; Nothing$. The alternative value 2 is discarded as soon as the expression in parentheses succeeds with the value 1; this is beneficial in terms of efficient use of time and space, but in this example it leads to failure, because the result 1 does not satisfy the subsequent test *even*.

If *m* is an instance of *MonadPlus*, then so is *Parser m*. The additional operations are obtained by lifting the operations on *m*.

instance $MonadPlus\ m \Rightarrow MonadPlus\ (Parser\ m)$ **where**
  $mzero = Parser\ (\lambda s \rightarrow mzero)$
  $xp \oplus yp = Parser\ (\lambda s \rightarrow runParser\ xp\ s \oplus runParser\ yp\ s).$

The law (∗) also fails for the parser monad *Parser Maybe* based upon *Maybe*, something that will turn out to be crucial later.

We shall use the operations $\gg\!=$ and ⊕ to build parsers that handle concatenation and alternation in context-free grammars. All that is missing now are the basic parsers that deal with individual characters. The parser *pChar c* compares the next character of the input with *c* and succeeds if they match, consuming the character *c*.

  $pChar :: MonadPlus\ m \Rightarrow Char \rightarrow Parser\ m\ (\,)$
  $pChar\ c =$
    $Parser\ (\lambda s \rightarrow$ **case** s **of** $c' : s' \mid c \equiv c' \rightarrow return\ ((\,), s');\ \_ \rightarrow mzero).$

For convenience, we also define *pString s* so that it recognises the characters in the string *s* one after another, so that *pString* "abc" is equivalent to *pChar* 'a' $\gg$ *pChar* 'b' $\gg$ *pChar* 'c'.

$$\begin{aligned} &\mathit{pString} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{String} \rightarrow \mathit{Parser}\ m\ (\,) \\ &\mathit{pString} = \mathit{foldr}\ (\gg)\ (\mathit{return}\ (\,)) \cdot \mathit{map}\ \mathit{pChar}. \end{aligned}$$

3 *LL*(1) and beyond

Broadly speaking, a grammar belongs to the class *LL*(1) if, whenever alternatives such as $A \rightarrow B \mid C$ occur, the set of tokens that can start an instance of B is disjoint from the set that can start an instance of C; this must hold whether B and C are single non-terminals or other strings that appear as the right-hand sides of productions. (The story is complicated a bit if either B or C can produce the empty string, but we can ignore that complication here.) If we build a parser for A from parsers for B and C by writing $pA = pB \oplus pC$, then we can be sure that no input string would cause both pB and pC to succeed. So if pB succeeds, it is safe to rule out a subsequent attempt to apply pC, and that is what happens in a parser based on *Maybe*.

For the *Maybe*-based parser to work, it is certainly sufficient that the grammar is *LL*(1). On the other hand, the *LL*(1) condition is not necessary in all cases, as is easily shown by the grammar $S \rightarrow$ 0 0 | 0 1, where 0 and 1 are tokens. Let us consider what happens when the parser *pString* "00" $\oplus$ *pString* "01" is applied to input "01" using the *Maybe* monad. First, the left alternative, *pString* "00" = *pChar* '0' $\gg$ *pChar* '0', is tried; the first '0' succeeds, but the second one fails, and this causes the whole alternative to fail. At this point, the alternation operator $\oplus$, seeing that its left operand has failed, is able to try its right operand, the parser *pString* "01", on the original input. This parser succeeds, and so the overall outcome is success, as it should be.

In more complicated settings, *Maybe*-based parsers often continue to work correctly even where the grammar fails the *LL*(1) condition. For example, in a programming language grammar we could write

$$\mathit{stmt} \rightarrow \mathit{variable} := \mathit{expr} \mid \mathit{expr},$$

perhaps permitting a solitary expression as a statement in order to allow for procedures called for their side effect. A *Maybe*-based parser could work correctly with this grammar, recognising a variable at the beginning of a statement, then finding that it is not followed by :=, switching to the other alternative, and parsing an expression as a statement in itself. In this case, it is not easy to make the grammar *LL*(1) without relaxing the condition that the left-hand side of an assignment should be a variable rather than a general expression.

So far we have seen that all grammars that are *LL*(1) can be parsed correctly with *Maybe*-based combinators, but so can some grammars that are not *LL*(1). The next step is to introduce an undecidable problem, which we shall use first to show that there is no algorithm that decides whether a given context-free grammar

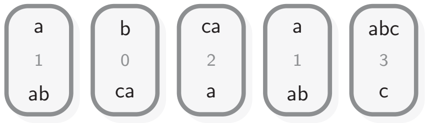

Fig. 1. A solution.

is ambiguous, then by a modification of the argument, to show that no algorithm can decide, given a context-free grammar, whether the corresponding *Maybe*-based parser works correctly.

4 Post's correspondence problem

In Post's correspondence problem, we are given an endless supply of several different kinds of tiles, each labelled with strings of letters at the top and the bottom, and we are asked whether it is possible to lay out a row of tiles in such a way that the same string is obtained by reading across the top labels or the bottom labels as in Figure 1, where both the top and the bottom rows of labels spell out "abcaaabc". Each given tile may be used once, several times or not at all in the layout.

We can represent a tile by a pair of strings, and we say that a layout is a *solution* if the upper and lower labels concatenate to give the same string:

```
type Tile = (String, String)

solution :: [Tile] → Bool
solution layout = (concat (map fst layout) ≡ concat (map snd layout)).
```

To find solutions for a given set of n tiles, we can generate non-empty layouts in increasing order of length, representing each layout by a list of indices into the list of tiles,

```
choices :: Int → [[Int]]
choices n = tail (concat (iterate step [[ ]]))
  where step css = [ c : cs | c ← [0 . . n − 1], cs ← css ].
```

For example:

```
> choices 4
[[0], [1], [2], [3], [0, 0], [0, 1], [0, 2], [0, 3], [1, 0], . . .
```

The solutions are those layouts where the upper and lower labels match,

```
solve :: [Tile] → [[Int]]
solve tiles =
  [ cs | cs ← choices (length tiles), solution (map (tiles !!) cs) ].
```

This function quickly solves the set of tiles shown in Figure 1:

```
> let sipser = [("b", "ca"), ("a", "ab"), ("ca", "a"), ("abc", "c")]
> solve sipser
[[1, 0, 2, 1, 3], [1, 0, 2, 1, 3, 1, 0, 2, 1, 3], . . .
```

We could just as well represent a solution by a list of tiles instead of a list of integer indices; but the lists of indices will play a vital role later in the story, so it is best to introduce them from the start.

If a set of tiles has a solution, then *solve* will find it eventually; but some sets have no solution, and for them *solve* does not return the empty list, but instead runs forever without producing any information. Sometimes it is obvious that there are no solutions, perhaps because there is no tile where the top and the bottom labels begin with the same character. In general, however, *it is undecidable whether a given set of tiles has a solution.* We will assume this result here without proving it; Sipser (2005) gives a proof by reduction from the halting problem for turing machines. (The tiles shown in Figure 1 are taken from an example in the same work.) Before returning to the problem of *Maybe*-based parser combinators, we will first mention a classic undecidable problem connected with parsing: the problem of determining whether a grammar is ambiguous.

5 Ambiguity

Given a set of tiles, we can construct a context-free grammar that is ambiguous exactly if the set of tiles has a solution. The layout described in Section 4 is $[1, 0, 2, 1, 3]$, leading to the labels "abcaaabc", and we will encode this as the string

"3+1+2+0+1=abcaaabc".

To the left of the equals sign appears, in reverse order, the list of tiles chosen. To the right appears the concatenated sequence of labels from the tiles, in this case the same string whether we take the upper labels or the lower ones.

Given a list of labels, either the upper ones from a list of tiles or the lower ones, we can write productions to describe the set of strings that can be assembled. We can express these productions as a function that takes the list of labels and returns a parser:

```
assembly :: MonadPlus m ⇒ [String] → Parser m ()
assembly labels = pA
  where
    pA = msum [(pString (show i) ≫ pA′ ≫ pString t | (i, t) ← zip [0 . .] labels ]
    pA′ = (pChar ‘+’ ≫ pA) ⊕ pChar ‘=’.
```

The standard function *msum* = *foldr* (⊕) *mzero* combines a list of alternatives into a single parser.

For the list *labels* = ["b", "a", "ca", "abc"], the parser *assembly labels* corresponds to the productions,

$$A \rightarrow 0\ A'\ \mathsf{b} \mid 1\ A'\ \mathsf{a} \mid 2\ A'\ \mathsf{c\ a} \mid 3\ A'\ \mathsf{a\ b\ c}$$
$$A' \rightarrow +\ A \mid =.$$

The reason for the reversed list of tile indices emerges here, because the grammar generates the list of indices and the string of labels simultaneously by growing them outwards from the middle, one in each direction. The explicit list of indices is helpful

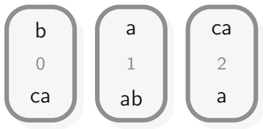

Fig. 2. Another layout.

in ensuring that the parser *assembly labels* work correctly in either monad, because the indices guide the sequence of choices without the need for deep backtracking. Each parser *pString* (*show i*) $\gg$ *pA*$'$ $\gg$ *pString t* begins by looking for the string *show i*, so that only one of them can succeed on any input string.

Now, given a tile set *tiles*, we can form the following two parsers:

$$\begin{aligned} \mathit{top} &= \mathit{assembly}\ (\mathit{map\,fst\ tiles}) \\ \mathit{bottom} &= \mathit{assembly}\ (\mathit{map\ snd\ tiles}). \end{aligned}$$

One of these parsers accepts strings that describe a layout and the labels on its top row, and the other accepts similar strings for the bottom row. The layout in Figure 2 shows that the string “2+1+0=baca” is accepted by *top* and the string “2+1+0=caaaba” is accepted by *bottom* for our usual set of tiles. The lists of indices in these two strings match, because they come from the same layout, but the labels do not match, because the layout is not a solution. If the Post correspondence problem for a set of tiles does have a solution, then there will be a string, like “3+1+2+0+1=abcaaabc” in our example, that is accepted by both *top* and *bottom*. Conversely, if any string is accepted by both, then the string contains the list of tiles in a solution, together with the labels that can be read off on both the top and the bottom rows.

It is vital to include the indices in the strings because otherwise we would be testing only whether there is a string that can form the top row of one layout and the bottom row of another one. In our example, the string “aca” can be obtained as the top row of the layout $[1, 2]$ and the bottom row of the layout $[2, 0]$, but that is not enough for a solution because a solution requires a string that is both the top and the bottom row of the *same* layout.

We can put the two parsers together with $\oplus$ to get a parser that accepts at least one string in two different ways exactly if the set of tiles has one or more solutions. For the purpose of experiment, we will use the list monad, written [ ] in the type of the parser so that it can return multiple results,

```
ambiguous :: [Tile] → Parser [ ] Int
ambiguous tiles =
  (assembly (map fst tiles) ≫ pChar '!' ≫ return 1)
     ⊕ (assembly (map snd tiles) ≫ pChar '!' ≫ return 2).
```

The character ‘!’ is used as an end-of-file marker to make sure the whole string is matched, and the return values 1 and 2 make it easier to see what is happening. Expressing the same construction as a grammar, we would have one set of productions for the top labels in the tiles, similar to those for A and A' shown earlier, a second set for the bottom labels using non-terminals B and B', and two productions $S \to A$! | B ! to join them together.

Let us try the parser on some examples.

```
> runParser (ambiguous sipser) "2+1+0=baca!"
[(1, " ")]
> runParser (ambiguous sipser) "2+1+0=caaba!"
[(2, " ")]
> runParser (ambiguous sipser) "2+1+0=babc!"
[]
> runParser (ambiguous sipser) "3+1+2+0+1=abcaaabc!"
[(1, " "), (2, " ")].
```

The top labels in Figure 2 read "baca", and the string that encodes this fact is accepted with the result 1; similarly, "caaba" appears as the bottom labels in the same layout, and a corresponding string is accepted with the result 2. On the other hand, the string "babc" does not represent either the top or the bottom labels on these tiles, so the next test string is not accepted at all. Finally, the string "abcaaabc" can be obtained from either the top or the bottom labels of the tiles 1, 0, 2, 1, 3, so the string "3+1+2+0+1=abcaaabc!" is accepted in *two* ways by the parser, showing that the grammar is ambiguous.

For any set of tiles, we can form the parser *ambiguous tiles*, and that parser will return multiple results on exactly those strings that encode a solution to the correspondence problem. So the problem of deciding whether the set of tiles has a solution is reduced to the problem of determining whether the grammar behind this parser is ambiguous, and we may deduce that the ambiguity problem is undecidable. Next, we will use a similar construction to show that it is not decidable whether a grammar is correctly parsed by a *Maybe*-based parser.

6 Shallow and deep backtracking

In place of the parser constructed by the function *ambiguous*, let us consider now another parser, also put together from two instances of *assembly*. This time, we leave the monad m unspecified:

```
backtrack :: MonadPlus m ⇒ [Tile] → Parser m Int
backtrack tiles = inner ≫= (λx → pChar '!' ≫ return x)
  where
    inner = ((assembly (map fst tiles) ≫ return 1)
      ⊕ (assembly (map snd tiles) ≫ pChar '?' ≫ return 2)).
```

Again, '!' is used as an end-of-file marker, but the grammar is carefully factored, bearing in mind that the distributive law (∗) is not satisfied by *Maybe*. Note too the presence of the character '?' in one alternative of the *inner* parser. For convenience, we give names to specialised versions of the parser:

```
backtrackL = backtrack :: [Tile] → Parser [] Int
backtrackM = backtrack :: [Tile] → Parser Maybe Int.
```

Let us examine what happens when the list-based parser reads a typical input string:

```
> runParser (backtrackL sipser) "3+1+2+0+1=abcaaabc?!"
[(2, " ")].
```

Because the input string consists of a solution followed by "?!", the parsing goes as follows:

- The parser *assembly* (*map fst tiles*) succeeds, causing *inner* to produce the result 1 and the remainder "?!".
- On this remainder, the parser *pChar* '!' fails, causing backtracking.
- Now the parser *assembly* (*map snd tiles*) succeeds, producing again the remainder "?!". After this, the parser *pChar* '?' consumes the '?', causing *inner* to produce the result 2 and the remainder "!".
- This time the parser *pChar* '!' succeeds, and the overall outcome is success.

But what happens if we use a parser based on *Maybe* instead?

```
> runParser (backtrackM sipser) "3+1+2+0+1=abcaaabc?!"
Nothing.
```

This time the story is different. The parser *assembly* (*map fst tiles*) succeeds as before without the need for backtracking, guided by the indices embedded in the input, and the parser *pChar* '!' subsequently fails. But this time there is no possibility of backtracking to try the other branch, and the whole parse fails, yielding *Nothing*.

We should also check the behaviour of the parser for strings that can be generated only from the top or the bottom of a layout. A string that can only be generated from the top labels gives no result with either parser because there is no way to consume the '?' character after matching with the top:

```
> runParser (backtrackL sipser) "2+1+0=baca?!"
[ ]
> runParser (backtrackM sipser) "2+1+0=baca?!"
Nothing.
```

In this case, the *Maybe*-based parser gives a result consistent with the list-based one. Again, if a string can only be generated from the bottom labels, it gives a positive result from both parsers:

```
> runParser (backtrackL sipser) "2+1+0=caaba?!"
[(2, " ")]
> runParser (backtrackM sipser) "2+1+0=caaba?!"
Just (2, " ").
```

So it is exactly when a string represents a solution to the correspondence problem that the list-based and *Maybe*-based parsers disagree; in that case, it is the *Maybe*-based parser that gives the wrong answer, failing to recognise a string that is generated by the underlying grammar. Whether such a string exists is, as before, undecidable, given the set of tiles, so we conclude that it is undecidable, given a grammar, whether the grammar is correctly parsed by the *Maybe*-based parser that is derived from it.

The parser *backtrack tiles* is made from parts that all work correctly with either monad, and it is only the one occurrence of $\oplus$ in the definition of *backtrack* itself where the distinction between shallow and deep backtracking matters. It is possible to design a system of parser combinators where there are two alternation operators, one providing shallow and the other deep backtracking. With such a system, the same argument can be used to show that it is undecidable whether a specific occurrence of deep alternation can safely be replaced with the shallow version.

7 Historical note

The result presented in this paper has been known for many years, and in fact for many years before parser combinators were invented. In a set of notes written for a course given in 1967, Knuth (1971, 2003) describes an abstract *parsing machine*. This machine runs programs in which the instructions either recognise and consume a token from the input or call a subroutine to recognise an instance of a non-terminal. Each instruction has two *continuations* for success and failure (though Knuth did not use the word), and part of the subroutine mechanism is that if a subroutine returns with failure, then the input pointer is reset to where it was when the subroutine was called. A subroutine that returns successfully, however, deletes the record of the old position of the input pointer. There is a natural translation of context-free grammars into programs for this machine, which behave exactly like combinator parsers based on *Maybe*. Knuth proves that the correct functioning of a program for the parsing machine is undecidable, using the same reduction presented in this paper, although the details have been changed in order to work within a fixed alphabet.

References

Knuth, D. E. (1971) Top-down syntax analysis. *Acta Inform.* **1**, 79–110. Reprinted as Chapter 14 of Knuth (2003).

Knuth, D. E. (2003) *Selected Papers on Computer Languages*. Palo Alto, CA: CSLI.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Sipser, M. F. (2005) *Introduction to the Theory of Computation*. 2nd ed. Boston, MA: Course Technology.