Complexity analysis and monadic specification of memoized functional parsers.

Barbara Szydlowski

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
NOTE TO USERS

The original manuscript received by UMI contains indistinct, slanted and or light print. All efforts were made to acquire the highest quality manuscript from the author or school. Microfilmed as received.

This reproduction is the best copy available

UMI
COMPLEXITY ANALYSIS AND
MONADIC SPECIFICATION OF
MEMOIZED FUNCTIONAL
PARSERS

by

Barbara Szydlowski

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor
Windsor, Ontario, Canada 1996
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-31006-X
ABSTRACT

One approach to implementing parsers in a purely functional programming language is to model them as functions, and to define a set of higher-order combinators that allow one to build larger parsers out of smaller components. These combinators implement grammar constructions such as alternation or sequencing. They can be used to construct parsers with structures resembling the BNF notation of the grammars of the languages being processed. Such parsers are modular and easy to modify and understand. A major disadvantage of this approach is that the resulting parsers use top-down fully backtracking strategy that may lead to enormous time and space requirements.

The efficiency of parsers can be improved by adding bookkeeping features that eliminate unnecessary backtracking. In this thesis we investigate a technique called memoization. A memoized parser computes its result based on previously computed results that have been stored in a memo-table. A parser is a program that determines the syntactic structure of an input sequence of symbols in some language. It may produce some kind of abstract syntax tree as output. We consider the simplest type of parsers — language recognizers that can be thought of as programs determining only if the input sequence belongs to a given language. We show that memoized recognizers constructed for an arbitrary grammar have $O(n^3)$ time complexity where $n$ is the length of the input to be processed. The space required to store the
memo table is (at most) $O(n^3)$. In purely functional programming languages that support updateable in-place variables the space requirements could be reduced to $O(n^2)$.

Monads, which are abstract structures from Category Theory, have proven useful for addressing many computational problems in purely functional programming. The monadic approach allows one to build basic parsers and combinators out of components that represent various programming language features such as state, exceptions, or non-determinism. These features automatically become the characteristics of the resulting parser. We show how memoized recognizers could be implemented in a fully modular way using the monadic approach. We also describe how the technique could be extended to improve efficiency of more complex language processors, such as syntax-directed evaluators.
ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Richard Frost for helpful discussions, support, and his assistance in the preparation of this manuscript. In addition, special thanks to Dr. Yung Tsin and Dr. Richard Caron for their constructive comments on the final draft of this thesis.
TABLE OF CONTENTS

ABSTRACT IV

ACKNOWLEDGMENTS VI

TABLE OF CONTENTS VII

LIST OF FIGURES XIII

CHAPTER 1 1

INTRODUCTION

1 CONSTRUCTING PURELY FUNCTIONAL LANGUAGE PROCESSORS 1

2 THE TYPE OF PARSERS 2

3 BNF NOTATION AND FUNCTIONAL LANGUAGE PROCESSORS 3

3.1 Basic parsers 4

3.2 Combinators 7

4 NON-DETERMINISTIC LANGUAGE PROCESSORS 8

5 Memoization 9

6 MONADS 11

7 MONADS AND FUNCTIONAL PROGRAMMING 12

8 MONADIC CONSTRUCTION OF LANGUAGE PROCESSORS 13
CHAPTER 2

INTRODUCTION TO CATEGORY THEORY AND MONADS

1 INTRODUCTION

2 CATEGORIES

3 FUNCTORS

4 NATURAL TRANSFORMATIONS

5 FUNCTOR CATEGORIES

6 MONADS

7 KLEISLI TRIPLES

8 KLEISLI CATEGORIES

CHAPTER 3

COMPUTATIONS AND MONADS

1 INTRODUCTION

2 CATEGORICAL SEMANTICS OF COMPUTATIONS

3 CATEGORY THEORY MONADS IN FUNCTIONAL PROGRAMMING

3.1 Monads in functional programs
3.2 Kleisli triples in functional programs

CHAPTER 4

MONADIC CONSTRUCTION OF PURELY FUNCTIONAL RECOGNIZERS

1 Introduction

2 The type of the monadic recognizers

3 Basic monads
   3.1 The identity monad
   3.2 Exceptions
   3.3 Non-determinism

4 Combining monads
   4.1 Construction of a combined monad
   4.2 The parametrized state reader monad
   4.3 Deterministic recognizers
   4.4 Non-deterministic recognizers

5 Concluding remarks

CHAPTER 5

MONADIC CONSTRUCTION OF MEMOIZED LANGUAGE PROCESSORS USING TYPE CONSTRUCTOR CLASSES IN GOFER
1 The system of type constructor classes in Gofer 48
2 Monads and type constructor classes 48
3 The class Recognizer 50
4 Memoized recognizers 52
   4.1 The state monad 52
   4.2 The parametrized list monad 53
   4.3 The memo-table 55
5 Parsers 57
   5.1 The type of the monadic parsers 57
   5.2 The class Parser 58
   5.3 The parametrized input monad 59
   5.4 Memoized parsers 62
6 Complexity of memoized language processors 63

CHAPTER 6

CONCLUSION 66

BIBLIOGRAPHY 68

APPENDIX A

MEMOIZING PURELY-FUNCTIONAL TOP-DOWN BACKTRACKING LANGUAGE PROCESSORS 71
1 CONSTRUCTING MODULAR NON-DETERMINISTIC LANGUAGE PROCESSORS IN FUNCTIONAL PROGRAMMING LANGUAGES 72

2 MEMOIZING LANGUAGE PROCESSORS 78

3 MEMOIZATION IN PURELY-FUNCTIONAL LANGUAGES 80

4 MEMOIZING PURELY-FUNCTIONAL RECOGNIZERS 84

4.1 The memo-table 85
4.2 The memoized recognizers 87
4.3 The Algorithm 89

5 COMPLEXITY ANALYSIS 93

5.1 Elementary operations 94
5.2 The size of the memo-table 94
5.3 Memo-table lookup and update 95
5.4 Basic recognizers 95
5.5 Alternation 96
5.6 Sequencing 96
5.7 Merging results returned when a recognizer is applied to a list of start positions 96
5.8 The execution tree 97
5.9 Proof of $O(n^3)$ time complexity 98

6 A MONADIC APPROACH TO INCORPORATE MEMOIZATION 101

6.1 Monads 102
6.2 Non-memoized monadic recognizers 104
6.3 Memoized monadic recognizers 106

7 Memoizing parsers and syntax-directed evaluators 109

8 Concluding comments 111

9 Acknowledgments 112

10 References 112

APPENDIX B

IMPLEMENTATION OF MONADIC LANGUAGE PROCESSORS USING TYPE CONSTRUCTOR CLASSES IN GOFER 115

VITA AUCTORIS 125
LIST OF FIGURES

Figure 1  Basic parsers  4
Figure 2  Basic recognizers  5
Figure 3  Example definitions of orelse and then  7
Figure 4  Example diagram  16
Figure 5  Functor F mapping category C into D  19
Figure 6  Natural transformation mapping functor F into functor G  20
Figure 7  Monad laws  30
Figure 8  List monad  31
Figure 9  Kleisli triple laws  32
Figure 10  Kleisli triple for lists  33
Figure 11  The type of monadic recognizers  37
Figure 12  The identity monad  38
Figure 13  The exception monad with zero and plus  40
Figure 14  The list monad with zero and plus  42
Figure 15  The parametrized state reader monad  44
Figure 16  Classes: Monad and MonadPlusZero  49
Figure 17  The class Recognizer  51
Figure 18  The state monad  53
Figure 19  The parametrized list monad  54
Figure 20  The class Parser  58
Figure 21  The parametrized input monad  59
Figure 22  A functional program containing a definition of the recognizer

Figure 23  The relationship between the grammar and the program implementing the recognizer

Figure 24  Definition of the Fibonacci function

Figure 25  A memoized version of the Fibonacci function

Figure 26  The relationship between a recognizer and its memoized version
Chapter 1

INTRODUCTION

1 Constructing purely functional language processors

One approach to implementing language processors in a purely functional programming language is to model them as functions, and to define a set of higher-order combinators that allow one to build larger parsers out of smaller components. This approach dates back to Burge’s book on recursive programming techniques [BR75], and it has been popularized in functional programming by Wadler [WA85], Frost [FT92], Hutton [HT92], and others. According to the approach, a parser is a program that takes a string of tokens as input and yields some kind of an abstract tree, that describes the grammatical structure of the string, as result.

Owing to the fact that a parser might not consume all of the input string, it is convenient to represent parsers as functions that when applied to the input string of tokens return a pair: value (an abstract tree) and the unconsumed part of the input. Furthermore, a parser might fail on its input. One way of distinguishing between success and failure is to have parsers return a list of pairs rather than a single pair, with the convention that a singleton list denotes success and an empty list denotes failure [WA85].
We start by defining the type for parsers and recognizers. Next, we define basic parsers and combinators. The implementation language for this thesis is Gofer [JN94], a purely functional programming language that supports many useful features, such as lambda expressions and type constructor classes [JNA93]. These features will be discussed later in this thesis.

2 The type of parsers

Using the conventions described in the previous section we define our parsers to be of the type:

type Parser a = String -> [(a, String)].

That is, a parser of the type Parser a is a function from the input string of characters to the list of pairs (value_of_type_a, rest_of_the_string).

A recognizer is a language processor that simply determines whether or not the input string of characters belongs to a defined language. Using the same principles as for parsers, we define our recognizers to be functions from the input string of characters to the list containing an unconsumed part of the input. A singleton list of results denotes success; an empty list of results denotes failure.

type Recognizer = String -> [String]

Throughout the remainder of this thesis we will also introduce the definitions of parsers and recognizers that are of slightly different types than
those given above. For example, instead of applying language processors to
the string of tokens yet to be processed (with the assumption that the first
token to be processed is the first token of the input string), we can
apply each of them to the whole input string and a single start position (the
position specifies the first token to be processed). One advantage of this
representation is that the input string remains unchanged through the whole
process of parsing or recognition, and there is no need for each processor to
return an unconsumed part of it. Instead, each parser or recognizer returns a
start position for the next processor. The output of processors is more
compact and therefore more suitable for storing (for example in a memo-
table). The corresponding modified types of parsers and recognizers are
given below.

\[
\text{type ParserInt a = String} \to \text{Int} \to [(a, \text{Int})]
\]
\[
\text{type RecognizerInt = String} \to \text{Int} \to [\text{Int}]
\]

3 BNF notation and functional language processors

In BNF notation grammars are constructed by defining a set of
terminals and a set of productions. The symbol `\epsilon` denotes an empty
production, the symbol `|` denotes alternation, and juxtaposition denotes
sequencing. More complex productions can be built from simpler ones by
combining them using alternation or sequencing.

Corresponding language processors can be constructed by defining the
functions `empty` and `term` that correspond to an empty production and a
terminal, and the higher-order functions `orElse` and `then` that correspond to
alternation and sequencing in BNF respectively. Similarly as in BNF, larger parsers can be built from smaller components by combining them using the alternation or sequencing (higher-order) operators. The structure of the resulting parsers closely resembles the structure of the underlying grammars.

3.1 Basic parsers

In this section we define three basic parsers and recognizers that can be used as building blocks for more complex language processors. The parser `emptyP` always succeeds without consuming any of the input string, and it returns a value `v`. The parser `failP` always fails, regardless of its input. The parser `termP c` processes a single character at the beginning of the input string. It fails if the first character to be processed is not `c`, or if the input is an empty string. Example definitions are given below.

```haskell

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser a = String -&gt; [(a, String)]</td>
<td></td>
</tr>
<tr>
<td>emptyP :: a -&gt; Parser a</td>
<td></td>
</tr>
<tr>
<td>emptyP v inp = [(v, inp)]</td>
<td></td>
</tr>
<tr>
<td>failP :: Parser a</td>
<td></td>
</tr>
<tr>
<td>failP inp = []</td>
<td></td>
</tr>
<tr>
<td>termP :: Char -&gt; Parser Char</td>
<td></td>
</tr>
<tr>
<td>termP c inp = case inp of</td>
<td></td>
</tr>
<tr>
<td>[] -&gt; failP inp</td>
<td></td>
</tr>
<tr>
<td>(x:xs) -&gt; if x == c then [(c, xs)] else failP inp</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1 Basic parsers

4
Similarly, one can define the corresponding recognizers. The recognizer emptyR always succeeds returning its input unchanged. The recognizer failR always fails. The recognizer termR c consumes a single character c at the beginning of the input string or fails if the first character is not c, or if the input is an empty string.

```
type Recognizer = String -> [String]

emptyR :: Recognizer
   emptyR inp = [inp]

failR :: Recognizer
   failR inp = []

termR :: Char -> Recognizer
termR c inp = case inp of
    []   -> failR inp
    (x:xs) -> if x == c then [xs]
                      else failR inp
```

Figure 2  Basic recognizers

The following illustrates the use of the above definitions.

```
? termR 'a' "aaa"
[('a','aa')]  
? emptyR 'a' "aa"
[('a','aa')]  

? termR 'a' "aaa"
["aa"]   
? emptyR "aa"
["aa"]
```

It is not difficult to modify the above definitions so that the language processors built with them accept as parameters a single start position and
the whole input string. However, rather than rewriting all the definitions, let us consider the relationship between the types of the two corresponding processors. A parser that is applied to an unconsumed part of the input string is of the type:

\[
\text{type } \text{Parser } a = \text{String } \rightarrow \{(a, \text{String})\}.
\]

The corresponding parser that is applied to a string and a single start position is of the type:

\[
\text{type } \text{ParserInt } a = \text{String } \rightarrow \text{Int } \rightarrow \{(a, \text{Int})\}.
\]

We can generalize the type of the first parser by abstracting over the type of the strings (considering it as an additional parameter). The type of the first parser could be written as:

\[
\text{type } \text{Parser } a b = b \rightarrow \{(a, b)\}.
\]

In a similar way we can parametrize the type of the second parser by abstracting over the actual representation of a "position"

\[
\text{type } \text{ParserInt } a b = \text{String } \rightarrow b \rightarrow \{(a, b)\}.
\]

We can now write the definition of the type ParserInt in terms of the type Parser

\[
\text{type } \text{ParserInt } a b = \text{String } \rightarrow \text{Parser } a b.
\]

Later in this thesis we shall see the advantages of this approach. The basic parsers and parser combinators of the type ParserInt no longer need to
be defined explicitly (except for the definition of `term` which depends on the actual representation of the input). They arise as special instances of lifting operations of the type `Parser` to the operations of the type `ParserInt`. The same applies to the definitions of corresponding recognizers.

3.2 Combinators

Basic parsers can be combined using the operators `orelse` and `then` to form more complex parsers. The operator `then` corresponds to sequencing in BNF. It applies the second parser to the result returned by the first one. The operator `orelse` corresponds to alternation in BNF. It applies two parsers to the same input and concatenates their results. Example definitions of these operators for the recognizers are given below. The notation `p 'orelse' q` is equivalent to `orelse p q`; single quotes are used in Gofer to denote an infix operator.

```haskell
thenR :: Recognizer -> Recognizer -> Recognizer
(p `thenR` q) inp |
| rp /= [] = q (rp!!0)
| otherwise = []
where rp = p inp

orelseR :: Recognizer -> Recognizer -> Recognizer
(p `orelseR` q) inp = p inp ++ q inp
```

Figure 3  Example definitions of `orelse` and `then`

The recognizer `(p `thenR` q)` fails if the recognizer `p` fails. Otherwise, the recognizer `q` is applied to the first element of the list returned by `p`. The
expression (rp!!0) denotes the first element of the list rp (the element at index 0); the symbol \'/=' is used in Gofer to represent the "not equal" operator. Using the definitions above we can construct recognizers with structures closely resembling the structures of the underlying grammars.

\[
\begin{align*}
\text{a\_then\_b} & \,:= \text{Recognizer} \\
\text{a\_then\_b} &= \text{termR 'a' \text{\_thenR} termR 'b'} \\
\text{a\_or\_b} & \,:= \text{Recognizer} \\
\text{a\_or\_b} &= \text{termR 'a' \text{\_orR} termR 'b'}
\end{align*}
\]

? \text{a\_then\_b} "abc"  \\
["c"]  \\
? \text{a\_or\_b} "abc"  \\
["bc"]

4 Non-deterministic language processors

Representing language processors as functions that return a list of results has one advantage: it is relatively easy to modify the processors so that they can return more than one result. One approach to modifying the definitions of the basic processors and combinators so that they can be used to build non-deterministic parsers is to first modify their types, so that they accept a list of inputs as parameter and return the list of outputs as their results. For example the type of the recognizers could be defined as

\[
\text{type RecognizerAmb} = \text{[String] -> [String].}
\]

Our first implementation of memoized recognizers in the purely functional language Miranda\(^1\) [TU90] is based on this approach [FS96] (a

\(^1\)Miranda is a trademark of Research Software Limited
copy is given in Appendix A). This approach has certain disadvantages: 1) most of the definitions of basic parsers and combinators must be modified, 2) the new definitions are often more complex and difficult to understand than the definitions of corresponding non-deterministic processors.

A better approach is to start with recognizers that are of the type \texttt{String $\rightarrow$ String} (they either succeed or return an internal error) and think about ambiguous recognizers as functions that involve an additional effect of non-determinism. The non-deterministic recognizers are of the type \texttt{String $\rightarrow$ [String]} where the resulting list can have any number of elements (an empty list denotes failure). The definitions of the initial recognizers are almost identical as those presented earlier in this chapter; the definitions of the corresponding non-deterministic recognizers arise automatically as special instances of lifting operations that return a single result into operations that return a list of results.

5 Memoization

Non-deterministic language processors can use backtracking and return multiple results. This ability, however, comes at a price. If an underlying grammar is ambiguous, the corresponding functional processor may have exponential space and time complexity. The main reason of such complexity is the repetition of the same computations during backtracking.

Memoization is a dynamic programming method which allows one to avoid performing the same computation more than once. A memoized
functional language processor is a function that takes an additional parameter – a memo-table containing all previously computed results. If the input has been processed before, the processor simply returns the corresponding result from the memo-table. If the input has not been processed yet, the new result is calculated and then the memo-table is updated.

One approach to implementing memoized recognizers is to slightly modify the definitions of basic recognizers and combinators so that the processors built with them accept a memo-table as part of their input and return a memo-table as part of their output. Next, the higher order function memoize is applied to each recognizer to store its result in the memo-table. This approach is described in detail in Appendix A.

Section 6 of Appendix A presents a slightly different approach to implementing memoization. If we consider our initial recognizers to be functions that return a list of values, then memoized recognizers can be represented as function that, when applied to a memo-table, return a list of values paired with the modified memo-table. One advantage of this approach is that there is no need to define memoized versions of basic recognizers and combinators. They arise automatically as special instances of lifting computations of the type \([a]\) into computations of the type \(\text{State} \to ([a], \text{State})\).

We have already suggested earlier that exploring the relationship between the types of two programs may help us to avoid unnecessary rewriting of one program into the other. Modifying a program by rewriting
its components is time consuming. Furthermore, the correctness of the initial program does not guarantee the correctness of its modified version. The monadic approach allows one to transform one program into the other in such a way that certain properties of the initial program are preserved. The basic ideas of this approach come from Category Theory and were introduced to Computing Science by Eugenio Moggi [MO89, MO90].

6 Monads

The notion of monads comes from Category Theory [MA71, BA90, PI91]. Informally speaking, a monad over a category is an abstract structure that allows one to reason about the objects of the category in terms of “how these objects interrelate” [MA71]. In Computing Science a category of interest is a category C of types and programs, and the relationship in mind, is a function on types in C [MO89, MO90].

In order to define this function, one should distinguish the type of values a program produces from the type of the program itself. For example, a program that is “effect free” does nothing but return a value of some type a. Therefore, the type of such a program is always identical to the type of the explicitly returned value. Consider a non-deterministic (or ambiguous) computation. This computation returns a set of possible results. Such a set could be represented, for example, as a list (that is the result could be of the type [a]). Using the same principles, a computation that handles exceptions could be of (an algebraic) type Raise String | Return a. In each of these
cases the type of the computation can be defined as a function of the type of
the value the computation produces.

A monad in a category of types and programs is a triple: a function on
types T (described above) that defines the type of programs, together with
two operations (that can be interpreted as composition and identity) that
allow the combination of such programs. The type constructor T abstracts
over the "effect" the program incorporates. The type of the two operations is
defined in terms of the type constructor T. Having defined a monad, one can
write a program as a set of components of the abstract type T a and use the
two operations to combine them. In other words, the structure of the
resulting program does not depend on the "effect" the program incorporates.

7 Monads and functional programming

Monads abstract over the kind of an "effect" that is added to a
program. This idea inspired Wadler [WA90] to introduce monads as a tool
for structuring purely functional programs. Programs written in purely
functional programming languages such as Haskell [PT96], Gofer [JN94], or
Miranda [TU90] are sometimes very difficult to modify if one wants to add
the "effects" such as state, interactive I/O, or just to print some error
messages. Wadler noticed that monads can be used to easily incorporate such
effects.

One advantage of the "monadic" approach is that the "effects" are not
"visible" in most of the function definitions of the program. The kind of the
effects the program includes can be determined by examining the definition of the monad. In order to add a new “effect” one simply has to change the definition of the monad and make some additional, usually trivial, local changes.

8 Monadic construction of language processors

It has been noted by Wadler [WA90, WA92] that the monadic approach allows one to build language processors that are more modular and easier to modify than traditional ones. A parser can be thought of as a program that deals with interactive input. When applied to the input string it returns a pair: value and the unconsumed part of the input. A non-deterministic parser can be thought of as a program that incorporates two “effects”: interactive input and non-determinism. Similarly, a deterministic parser is a program that combines interactive input and the ability to fail (that is, either it returns a value \( v \), or it fails producing no value – this can be captured by the type constructor: \( \text{Ok} \, v \mid \text{Fail} \)).

Defining a monad that represents a single “effect” is not difficult. It is also usually possible (using an ad-hoc approach) to define a “combined” monad that represents a composition of features. However, “how to combine arbitrary monads?” was a long-standing question and a topic of research in the area of so-called “monadic functional programming”. Attempts at finding a general technique for composing two arbitrary monads, were made by King and Wadler [KN92], Cenciarelli and Moggi [CE93], Jones and Duponcheel [JNB93], Steele [ST94], and Espinosa [ES95]
and ended with partial successes yielding techniques that were not general. More recently, Liang, Hudak, and Jones have proposed a new method, based on the theory of monad transformers [LI95], that allows one to compose monads in a fully modular way. One application of the technique was the construction of modular programming language interpreters.

Using the above technique we have implemented different types of language processors. Our processors are fully modular; they are built up from components that represent various programming language features, such as non-determinism, exceptions (used to represent determinism), interactive input (parsers), and state (memoization). This approach allowed us to easily extend the same technique we have initially used to memoize functional recognizers, to improve complexity of other language processors (such as, for example, syntax-directed evaluators). The details of this implementation can be found in Chapter 5.

9 Organization of this thesis

The remainder of this thesis is organized as follows.

Chapter 2 gives a brief introduction to Category Theory. We introduce basic definitions and explain category-theoretic notions of monads and Kleisli triples.

Chapter 3 describes how Category Theory monads and Kleisli triples are represented in purely functional programming. We start this presentation
with a short description of the category-theoretic semantics of computations proposed by Eugenio Moggi.

Chapter 4 presents how purely functional recognizers can be constructed using the monadic approach. We begin by discussing monadic recognizers that incorporate a single “effect”. Next, we describe how monads can be combined to yield recognizers that involve a combination of different features.

Chapter 5 describes the details of implementation of memoized language processors using type constructor classes in Gofer. We start by discussing the implementation of memoized recognizers. Next, we show how easily the technique for memoizing functional recognizers can be extended to more complex language processors.

Chapter 6 concludes summarizing the main advantages of using the monadic approach to construct purely functional language processors.

Appendix A contains a copy of the paper summarizing our early efforts to implement memoization in the purely functional programming Miranda. This paper includes a detailed description of the memoization algorithm together with its formal complexity analysis.

Appendix B contains the Gofer source code that implements memoized monadic language processors using type constructor classes.
INTRODUCTION TO CATEGORY THEORY AND MONADS

1 Introduction

In Category Theory mathematical concepts are represented using abstract diagrams. Such diagrams consist of vertices representing objects in a category, and directed edges (arrows) representing the mappings between these objects. A diagram is called commutative if for each pair of vertices $X$ and $Y$, any two paths formed from directed edges leading from $X$ to $Y$ yield, by composition of the corresponding mappings, equal mappings from $X$ to $Y$. A diagram

![Diagram](image)

Figure 4  Example diagram

can be used to represent the category of sets and functions. If $f$, $g$, and $h$ are functions such that $f : X \rightarrow Y$, $g : X \rightarrow Z$, $h : Z \rightarrow Y$, then the above diagram is commutative if $f = h \circ g$, where $\circ$ denotes usual composition of
functions. The same diagram may be used in many other contexts where $X$, $Y$, and $Z$ represent objects, $f$, $g$, and $h$ represent mappings between them, and the operation $\circ$ defines how two mappings can be composed.

Category Theory offers an abstract view of mathematical concepts. The concepts are abstracted from the context in which they were made precise, and therefore they can be instantiated into other contexts that were not considered before. This section gives a brief overview of basic concepts in Category Theory and presents a category-theoretic introduction to monads. The presentation here is based on the basic textbooks on Category Theory [MA71, BA90, PI91].

2 Categories

A category is a collection of objects, a collection of arrows (also called morphisms), together with two operations:

- **identity**, that assigns to each object $A$ an arrow $\text{Id}_A$ (the arrow pointing from the object $A$ to itself),

- **composition**, that assigns to each pair of arrows $f : A \to B$, $g : B \to C$ an arrow $g \circ f : A \to C$ called their composite.

The composition of morphisms must obey the associative law, that is for any morphisms

$$f : A \to B, \ g : B \to C, \ h : C \to D,$$

the condition
h \circ (g \circ f) = (h \circ g) \circ f \text{ must be satisfied.}

Another requirement of a category is that the \text{ld} function is an identity for the composition. That is, for any morphism \( f : A \rightarrow B \), the following condition must be satisfied

\[ f \circ \text{ld}_A = f = \text{ld}_B \circ f. \]

The categories of interest from the functional programming point of view are those where objects are types and morphisms are programs. Identity is an identity program. Composition is the way of combining two programs.

3 Functors

A \textit{functor} is a morphism of categories. Given two categories \( C \) and \( D \), a functor \( F : C \rightarrow D \) is a pair of functions:

- the object function that maps each object \( A \) of the category \( C \) into the corresponding object \( F(A) \) of the category \( D \),

- the arrow function that assigns to each arrow \( f \) in \( C \) the corresponding arrow \( F(f) \) in \( D \).

Each functor is required to preserve the identity and the structure of the composition of morphisms, that is for any two morphisms \( f \) and \( g \) and the identity morphism \( \text{ld}_A \) in \( C \), the following conditions must be satisfied:

\[ F(\text{ld}_A) = \text{ld}_{F(A)}, \]

\[ F(f \circ g) = F(f) \circ F(g). \]
The graphical representation of a functor is given on Figure 5.

![Diagram](image)

In category $C$:
- $A$ points to $B$ with $g$.
- $B$ points to $C$ with $f \circ g$.

In category $D$:
- $F(A)$ points to $F(B)$ with $F(g)$.
- $F(B)$ points to $F(C)$ with $F(f \circ g)$.

Figure 5  Functor $F$ mapping category $C$ into $D$

An endofunctor is a functor from a category to itself. It maps objects of a category $C$, and mappings between them, into corresponding objects and mappings in the same category. In a category of types and programs an endofunctor is a pair of functions: a function on types, and a function on programs. A functional programming example of an endofunctor is the pair [ES95]:

- a type constructor on lists that maps a type $a$ into the corresponding type $[a]$
type List a = [a],

- a standard library function map that can be thought of as a mapping of a
  program from a to b into a program from [a] to [b]

\[
\text{map} \quad : \quad (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \\
\text{map } f \; [] = [1] \\
\text{map } f (x:xs) = f \; x : \text{map } f \; xs.
\]

4 Natural transformations

Given two functors \( F, G : \mathcal{C} \rightarrow \mathcal{D} \) that are the mappings between the
same categories, a natural transformation \( \gamma \) from \( F \) to \( G \) is a mapping that
assigns to each object \( c \) of \( \mathcal{C} \) an arrow \( \gamma_c : F \; c \rightarrow G \; c \). In pictorial
representation the natural transformation \( \gamma \) can be thought of as a way of
"sliding" the diagram defining the functor \( F \) onto the diagram that defines
the functor \( G \), such that all parallelograms (like those shown on Figure 6) are
commutative.

![Diagrams of functors and natural transformations]

Figure 6  Natural transformation mapping functor \( F \)
into functor \( G \)

20
Natural transformations are families of arrows. If \( F \) and \( G \) are two functors in a category \( C \) of types and programs then given a type \( a \) (an object in \( C \)) we can think of a natural transformation as of a polymorphic function of the type \( F \ a \rightarrow G \ a \) [ES95]. It is not difficult to find examples of natural transformations in purely functional programming languages. For instance a polymorphic function list that takes as argument an element of any type \( a \) and returns a singleton list of the same type is a natural transformation.

\[
\begin{align*}
  \text{type } & F \ a = a \\
  \text{type } & G \ a = [a] \\
  \text{list } & :: F \ a \rightarrow G \ a \\
  \text{list } & x = [x]
\end{align*}
\]

5 Functor categories

Natural transformations can be composed. The composition of two natural transformations is a natural transformation. It is also associative and for each functor \( F \) there exists an identity natural transformation \( 1_F : F \rightarrow F \) (a mapping of a functor into itself). Therefore, given the categories \( C \) and \( D \), we can formally construct a functor category \( \mathcal{J} \) that has functors \( F : C \rightarrow D \) as its objects and natural transformations between such functors as its morphisms (see [MA71], [BA90], or [PI91] for details of this construction).

Owing to the fact that it is very convenient to abstract over the objects of a category and to reason about the category only in terms of functors and natural transformations, functor categories are extensively used in Category Theory. An example of a functor category is a monad – a functor category with one object. Monads are described in the next section.
6 Monads

In Category Theory, a monad over a category $C$ is a triple $(T, \eta, \mu)$, where $T : C \to C$ is an endofunctor (a functor mapping to and from the same category), and $\eta$ and $\mu$ are two natural transformations defined as follows:

$$\eta : \text{Id} \to T,$$

$$\mu : T \circ T \to T.$$

For the triple $(T, \eta, \mu)$ to be a monad, the three laws called the associative monad law, and the left and right identity laws must hold:

$$\mu \circ (T \circ \mu) \equiv \mu \circ (\mu \circ T) - \text{associative law}$$

$$\mu \circ (\eta \circ T) \equiv \text{Id}_C \equiv \mu \circ (T \circ \eta) - \text{left and right unit law}.$$

These laws (if satisfied) guarantee that the triple forms a functor category over the category $C$.

7 Kleisli triples

Kleisli triples are alternative descriptions of monads and there is a one-to-one correspondence between the two (see [MA71] for the proof). A Kleisli triple over a category $C$ is a triple $(T, \eta, \_\_)$, where

$$T : \text{Obj}(C) \to \text{Obj}(C) \quad (T \text{ is a function on objects, not a functor}),$$

$$\eta_A : A \to TA \text{ for } A \in \text{Obj}(C),$$
\[ f^* : TA \to TB \text{ for } f : A \to TB, \]

and the following conditions hold:

\[ \eta_A \circ f = \text{Id}_{TA} \circ f \] (right unit),

\[ f \circ \eta_A = f \] (left unit),

\[ g \circ (f \circ h) = (g \circ f) \circ h \] (associativity).

Given a monad \((T, \eta, \mu)\) one can construct the corresponding Kleisli triple \((T, \eta, \_\_\_\_)\) by restricting the endofunctor \(T\) to objects. Conversely, given a Kleisli triple the corresponding monad can be constructed by extending the function \(T\) to an endofunctor.

Monads are more widely used in Category Theory than Kleisli triples. They have the advantage of being defined only in terms of functors and natural transformations, which makes them more suitable for abstract manipulation. Kleisli triples, according to Moggi [MO89], are easier to justify from a computational perspective.

8 Kleisli Categories

Given a Kleisli triple \((T, \eta, \_\_\_\_)\) over a category \(C\) the corresponding Kleisli category \(C_T\) can be defined as follows:

- the objects of \(C_T\) are the same as those of \(C\),

- if \(f : A \to B\) is a morphism in \(C\) then \(f^* : A \to TB\) is the corresponding morphism in \(C_T\),

23
• the composition of two morphisms \( f : A \to T B \) and \( g : B \to T C \) in \( C_T \)
is defined as \( g \circ f \),

• this composition must be associative with \( \eta \) as its left and right unit.

If an underlying category \( C \) is a category of types and programs, then
given a Kleisli triple \((T, \eta, \_\_\_)\) we can formally construct a Kleisli category
\( C_T \) of types and programs over the category \( C \). The objects in \( C_T \) are types
(as in \( C \)); the morphisms in \( C_T \) are programs from the type \( A \) to the type \( T B \),
where the endofunctor \( T \) is a function on types in \( C \). The expression \( g \circ f \)
represents the composition of the two programs: \( f \) and \( g \). Moggi's categorical
semantics of computations, which is discussed in the next chapter, is based
on this idea.
1 Introduction

Kleisli categories were originally proposed by Eugenio Moggi as a convenient framework for structuring the semantics of programming languages [MO89, MO90]. The principle underlying Moggi’s work on monads was the distinction between simple data-valued functions and functions that perform computations. A data valued function is one that simply returns its value (and does nothing else). By contrast, a function that performs a computation can encompass ideas such as exceptions, state, or non-determinism, and as a consequence, it can implicitly produce more results than the result explicitly returned.

Wadler [WA90] noticed that Moggi’s ideas of using monads to structure the semantics of computations fitted well into the purely functional programming environment and proposed monads as a technique for structuring functional programs. He showed that monads can be used to express “imperative features” like updateable state, exceptions, non-determinism, or I/O in pure functional languages, while retaining the strong reasoning principles valid for these languages.
This chapter presents a brief overview of Moggi’s categorical semantics of computations based on monads. It also describes how category-theoretic monads are represented in purely functional programming languages. The presentation here owes much to the papers of Moggi [MO89, MO90], Wadler [WA90, WA92], and Hill and Clarke [HI94].

2 Categorical semantics of computations

The basic idea behind Moggi’s categorical semantics of computations is that, in order to interpret a programming language in a category $C$, one has to distinguish the object $A$ of values (of the type $A$) from the object $TA$ of computations (computations that produce a value of the type $A$). If $T$ is an unary operation on objects in $C$ that maps objects of values into corresponding objects of computations, then a program from $A$ to $B$ can be identified with a morphism from $A$ (the set of values of the type $A$) to $TB$ (the set of computations that produce a value of the type $B$) in $C$. In other words, a program is a function from values to computations.

In category-theoretic terms $T$ is an object mapping part of an endofunctor in $C$; in the context of functional programming, $T$ is a type constructor (a function on types). Moggi calls an operator $T$ “a notion of computation”, since it abstracts away from the actual type of values computations may produce. Examples of notions of computations that are of particular interest to functional programming are as follows:
• computations with side effects that denote a mapping from a state to a pair: value and the modified state

  type T a = State -> (a, State),

• non-deterministic computations that denote the set of all possible values

  type T a = [a],

• computations with exceptions that denote either a value or an exception

  data T a = Raise String | Return a,

• interactive input that denotes a function from the input string of tokens to a pair: the first token and the rest of the input

  type T a = [a] -> (a, [a]),

• interactive output that denotes a pair: a value and a function that maps a string (the output of the rest of the program) into a string (the output of the whole program)

  type T a = (a, [a] -> [a]).

Rather than focusing on a specific notion of computation, Moggi proposed Kleisli triples for modeling the notions of computations and Kleisli categories for modeling categories of programs. The components of the Kleisli triple \((T, \eta, \_\_\_\_)\) can be interpreted as follows. The endofunctor \(T\) is a function on types that maps the type of values into the type of corresponding computations. The natural transformation \(\eta\) applied to a
value returns a computation producing this value. The expression \( g \circ f \) where \( f : A \rightarrow TB \) and \( g : B \rightarrow TC \) has the following meaning: first apply \( f \) to some value of the type \( A \) to produce a computation of the type \( TB \), then evaluate this computation to obtain a value of the type \( B \), finally apply \( g \) to this value and return a computation of the type \( TC \) as a result. This expression corresponds to sequencing of two computations. The expression \( \eta a \) may be interpreted as a "pure" (i.e., effect-free) computation that does nothing but delivers a value. The composition \((g \circ f) a\) represents a computation that includes all of the "effects" of \( f \) followed by applying \( g \) to the value computed by \( f \).

In a similar vein one can interpret monads. Given a Kleisli triple \((T, \eta, \_\_\_\_\_)\) over a category \( C \) the corresponding monad is \((T, \eta, \mu)\), where \( T \) is a functor (that is \( T \) is a pair of mappings: an object mapping and a morphism mapping). The natural transformation \( \eta \) has the same meaning as for Kleisli triples; the natural transformation \( \mu \) (which is of the type \( T (T \ a) \rightarrow T \ a \)) can be thought of as a way of "flattening" a computation of computations into a single computation.

By using monads, Moggi defined the semantics of computations with "effects" which is independent of the kind of the effect these computations incorporate. Each effect is simply an instance of the same "notion of computation". Note that also "no effect" (or a pure computation) is such an instance. Based on the categorical semantics of computations Moggi built a system called computational \( \lambda \)-calculus, that can be used for proving
equivalence of programs. The analysis of the system is beyond the scope of this thesis. The detailed description of computational $\lambda$-calculus can be found in Moggi's papers [MO89, MO90].

3 Category Theory monads in functional programming

In functional programming, monads are usually presented as a kind of an abstract data type. The type definition includes the definition of the monad itself and definitions of primitive operations related to the particular effect the monad represents (see [WA92] and many other papers). For example if the effect in mind is state, the primitive operations may include: new (that creates a new structure representing the state), lookup (that searches the state), and update (that updates it). These operations have clearly nothing in common with Category Theory and they are in many cases application dependent. The functional definitions of the triples, however, closely resembles their corresponding category-theoretic definitions [HI94]. This section explores the relationship between the two.

3.1 Monads in functional programs

Given a Category Theory monad $(T, \eta, \mu)$, the functional programming monad is represented by a quadruple $(M, \text{map}, \text{unit}, \text{join})$ [WA90], where:

- $M$ is a type constructor, for example

  ```
  type M a = [a],
  ```

- $\text{map}$ is a higher order function (analogous to standard $\text{map}$ on lists):
map :: (a -> b) -> M a -> M b,

- unit represents the natural transformation \( \eta \)

  unit :: a -> M a,

- join represents the natural transformation \( \mu \)

  join :: M (M a) -> M a.

If \( a \) is a type of values then \( M a \) represents the type of programs that return values of the type \( a \). A pair \( M \) and \( \text{map} \) models a functor: the type constructor \( M \) is a mapping on objects (types), the higher-order function \( \text{map} \) is a mapping on arrows (programs). The definitions of the functions \( \text{unit} \) and \( \text{join} \) depend on the effect the monad represents. The function \( \text{unit} \) converts values to corresponding computations. The function \( \text{join} \) "flattens" a computation of computations into a single computation. A good example of \( \text{join} \) is the standard library function \( \text{concat} \) that "flattens" a list of lists into a single lists (see example below). The quadruple must satisfy the laws equivalent to the monad laws given in the previous chapter.

<table>
<thead>
<tr>
<th>join ( \circ ) unit = id = join ( \circ ) map unit</th>
<th>(left and right unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>join ( \circ ) map join = join ( \circ ) join</td>
<td>(associativity)</td>
</tr>
</tbody>
</table>

Figure 7 Monad laws
Lists represent non-deterministic computations (computations that return a set of possible results). The monad for lists [WA92] is given below. The type constructor \( M \) defines the type of computations. The function unit creates a one element list. The function join takes a list of lists and concatenates all sublists into a single list. The function map is a standard map on lists.

\[
\begin{align*}
type M \, a &= [a] \\
unit &:: a \rightarrow M \, a \\
unit &= \lambda x \rightarrow [x] \\
join &:: M \,(M \, a) \rightarrow M \, a \\
join &= \text{concat} \\
map &:: (a \rightarrow b) \rightarrow M \, a \rightarrow M \, b \\
map \, f &= \lambda x \rightarrow \text{case } x \text{ of} \\
&\hspace{1em} [] \rightarrow [] \\
&\hspace{1em} (x:x:\text{xs}) \rightarrow f \, x : \text{map } f \, \text{xs}
\end{align*}
\]

Figure 8  List monad

An expression of the form \( \lambda x \rightarrow e \) is called a lambda-expression, and denotes a function that takes an argument \( x \) and returns the value of the expression \( e \). Therefore, the function unit could equally well be defined as:

\[\text{unit } x = [x].\]

The definition given in Figure 8, however, is more expressive and corresponds more closely to the type of unit.
3.2 Kleisli triples in functional programs

In Wadler's more recent papers, the use of monads bears closer resemblance to Kleisli triples. Given a Category Theory Kleisli triple \((T, \eta, \_\_\_\)\), the corresponding "functional programming triple" is represented by a triple \((M, \text{unit}, \text{bind})\), where:

- \(M\) and \(\text{unit}\) have the same meaning as for monads from the previous section,
- \(\text{bind}\) is a polymorphic function such that the expression \((f \text{ `bind` } g)\) corresponds to \((g \circ \circ f)\).

The type of the function \(\text{bind}\) is:

\[
\text{bind} :: M a \to (a \to M b) \to M b;
\]

the meaning of the expression \((f \text{ `bind` } g)\) is: first apply the function \(f\) to produce a computation of the type \(M a\), evaluate this computation, apply the function \(g\) to the result, and return a computation of the type \(M b\). The function \(\text{bind}\) is simply a composition in a Kleisli category of computations.

The triple must obey the three laws given below, which are equivalent to the laws given in the previous chapter.

<table>
<thead>
<tr>
<th>law</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{unit} a \text{ <code>bind</code> } _b \rightarrow = n [a / b])</td>
<td>left unit</td>
</tr>
<tr>
<td>(m \text{ <code>bind</code> } _b \rightarrow \text{unit } b = m)</td>
<td>right unit</td>
</tr>
<tr>
<td>(m \text{ <code>bind</code> } (\lambda a \rightarrow n \text{ <code>bind</code> } _b \rightarrow m) = (m \text{ <code>bind</code> } _a \rightarrow n) \text{ <code>bind</code> } _b \rightarrow m)</td>
<td>associativity</td>
</tr>
</tbody>
</table>

where \(n [a / b]\) denotes \(n\) with a substituted for \(b\)

Figure 9 Kleisli triple laws
The Kleisli triple for lists, equivalent to the monad defined earlier, is given on Figure 8 [WA92].

\[
\begin{align*}
type \ M \ a &= [a] \\
unit &: a \rightarrow M a \\
unit &= \lambda x \rightarrow [x] \\
bind &: M a \rightarrow (a \rightarrow M b) \rightarrow M b \\
x \ "bind" \ y &= \text{case } x \text{ of} \\
&\quad [l] \rightarrow [l] \\
&\quad (a:x) \rightarrow (y a) ++ (x \ "bind" \ y)
\end{align*}
\]

Figure 10  Kleisli triple for lists

The functionality of the function bind in the example above is straightforward: if the first computation returns a list of possible results then the second computation must be applied to each element of this list; the results of each application should be combined, so that the final result is a single list. The operator "++" in the example above denotes list concatenation. This operator could be replaced with any other associative operator that combines two lists (e.g., merge). The functions map and join from the previous paragraph can be defined in terms of unit and bind as follows [WA92]:

\[
\begin{align*}
\text{map } f \ x &= x \ "bind" \ \lambda a \rightarrow \\
&\quad \text{unit } (f \ a) \\
\text{join } x &= x \ "bind" \ \lambda a \rightarrow \\
&\quad a
\end{align*}
\]
Moggi [MO89] proposed Kleisli triples as a representation of computations with "effects". His claim was that Kleisli triples were more convenient for expressing computations than monads. The same seems to be true in functional programming. The functionality of the `bind` operator as a composition of two programs is intuitive and easy to understand. By contrast, the functions `map` and `join` are rather difficult to justify from computational point of view. In the remainder of this thesis we will use the formulation of monads as Kleisli triples.
Chapter 4

MONADIC CONSTRUCTION OF PURELY FUNCTIONAL RECOGNIZERS

1 Introduction

Monads are a powerful tool in functional programming. If a program is written using a monad to pass around a variable (like the state or exception) then it is easy to change what is passed around simply by changing the monad. Only the parts of the program that deal directly with the quantity concerned need to be altered, parts which merely pass it on will stay the same.

This chapter describes how monads can be used to construct functional recognizers. We start by defining the type of the monadic recognizers. Next we give the definitions of basic monads and discuss how these monads can be used to build processors that incorporate a single "effect". The remainder of this chapter addresses the problem of combining monads. Different "effects" can be combined by using parametrized monads [LI95] to yield the recognizers that involve a combination of different features.

2 The type of the monadic recognizers

In the introduction of this thesis we have defined the recognizers as functions that applied to some input consume as much of it as possible and
return the unconsumed part of the input for further processing. If the input is represented as a string of tokens to be processed, the type of the recognizers can be written as

\[ \text{type Recognizer} = \text{String} ightarrow \text{String}. \]

Suppose that we want to have recognizers that return exactly one result or fail otherwise. This can be achieved by modifying the type above as follows

\[ \text{type RecognizerEx} = \text{String} ightarrow \text{Ex String} \]

where the type constructor Ex is defined as

\[ \text{data Ex a} = \text{Ok a} \mid \text{Fail}. \]

(That is the result of the form Ok v represents the successful recognition of the input with the single value \( v \) returned; Fail represents failure.)

Similarly, the recognizers that return a list of possible results (where an empty list denotes failure) can be represented using the type constructor

\[ \text{type RecognizerList} = \text{String} ightarrow [\text{String}]. \]

The monadic approach allows one to abstract over the “effect” the recognizers incorporate. Suppose that \( M \) is a “monadic” type constructor that represents some feature. By defining the type of the processors in terms of the type constructor \( M \), we can make the type of the recognizers independent of the effect they incorporate. Owing to the fact that the
representation of the input strings of tokens is application dependent, we can make it into a parameter to the type Recognizer.

\[
\text{type Recognizer } a = a \rightarrow M a
\]

Figure 11  The type of monadic recognizers

The type definition on Figure 11 should be interpreted as follows: a recognizer is a function that applied to a value of some type \( a \) returns (instead of returning a result of the type \( a \)) a computation that produces a result of the type \( a \). Such a computation may encompass ideas such as state, exceptions, or non-determinism. Later in this chapter we will discuss the advantages of defining the type of recognizers this way.

3 Basic monads

3.1 The identity monad

The identity monad represents computations as the values they deliver (i.e. "effect-free" computations). It is the starting point to which other capabilities can be added. The monad is represented by a triple: the type constructor \( M \) that can be thought of as an identity function on types, the function \( \text{unit} \) (which is an identity function), and the function \( \text{bind} \) that applies the function \( k \) to the value produced by the computation \( x \).
Using the operators unit and bind of the identity monad we can define the “identity” (“effect-free”) recognizers – the recognizers that either succeed returning a single result or, if something goes wrong, they simply produce an internal error. The “monadic” definitions of the recognizer empty and of the sequencing operator then are given below.

```
type Id a = a

unit :: a -> Id a
unit = \a -> a

bind :: Id a -> (a -> Id b) -> Id b
x `bind` k = k x
```

![Figure 12 The identity monad](image)

```
type Recognizer a = a -> Id a

emptyR :: Recognizer a
emptyR = unit

thenR :: Recognizer a -> Recognizer a -> Recognizer a
(p `then` q) inp = p inp `bind` \x1 ->
q x1 `bind` \x2 ->
unit x2
```

The definition of the recognizer (p `then` q) can be interpreted as follows. First the recognizer p is applied to the initial input and the value returned by p is bound to the variable x1. Next the recognizer q is applied to the input x1 and its value is bound to the variable x2. Finally the value x2 is converted into the corresponding computation and this computation is returned as a result.
The identity monad does not provide us with the notion of failure. This notion is required to give meaningful definitions of the recognizers fail and term. In addition, in order to define the operator `orElse` we need to specify the notion of "choice". The monad defined in the next subsection provides us with both of these notions.

3.2 Exceptions

Exceptions in purely functional programming languages were studied by Spivey who, independently of Moggi, noticed that monads are a useful tool for representing exceptions in functional programs [SP90]. We can think of a value of the type `Ex a` (defined on Figure 13) as a computation that either succeeds with a single value of the type `a`, or it fails producing no value. Therefore exceptions correspond to deterministic choice.

In addition to operators `unit` and `bind`, it makes sense to define two additional operations on the values of the type `Ex a` [LI95]. The operation `plus` defines a composition (in the sense of generalized addition) of two programs of the type `Ex a`. `plus` for exceptions can be interpreted as a (deterministic) choice operator that returns the first computation if it succeeds, and the second otherwise. The operation `zero` is an identity for `plus` and it represents a computation that always fails. The operations `plus` and `zero` are not part of the exception monad. They simply provide a different way of combining programs of the type `Ex a`. 
We shall see later in this chapter that the exception monad is not the only monad for which it makes sense to define zero and plus. Another example is a monad for lists where plus could be defined as concatenation (or merge) of two lists with an identity (zero) an empty list. The operators bind (with identity unit) and plus (with identity zero) provide the means to structure programs in modular way.

```haskell
data Ex a = Ok a | Fail

unit :: a -> Ex a
unit = \a -> Ok a

bind :: Ex a -> (a -> Ex b) -> Ex b
Fail `bind` k = Fail
(Return a) `bind` k = k a

zero :: Ex a
zero = Fail

plus :: Ex a -> Ex a -> Ex a
Fail `plus` x = x
x `plus` _ = x
```

Figure 13  The exception monad with zero and plus

We can now define the monadic recognizers fail and `term` as well as the operator `|>`. The definitions of `empty` and `then` given in the previous subsection do not have to be modified. We simply modify the type of the recognizers and replace `unit` and `bind` of the identity monad with the corresponding operators of the exception monad.

type Recognizer a = a -> Ex a
orelseR :: Recognizer a -> Recognizer a -> Recognizer a
(p `orelseR` q) inp = p inp `plus` q inp

failR :: Recognizer a
failR = zero

termR :: Char -> Recognizer String
termR c inp = case inp of
  [] -> fail
  (x:xs) -> if x == c
            then unit xs
            else fail

Using the above definitions we can build (deterministic) recognizers,
for example:

s :: Recognizer String
s = (termR 'a' `thenR` s `thenR` s) `orelseR` emptyR

?s "aaaaa"
Ok []

Although the underlying grammar is ambiguous, the recognizer s defined
above returns only one result. The empty string returned as the result means
that the whole input string has been successfully recognized as an s.

3.3 Non-determinism

We will use the list monad to transform our deterministic recognizers
into the recognizers that return a set of possible results. The definition of the
monad for lists given below is almost identical to that given in Figure 10.
The only difference is that the operator ++ (list concatenation) has been
replaced with the operator merge_res which merges two lists sorted in
ascending order with duplicates removed. The same operator merge_res is also a suitable plus for the monad.

```haskell
type List a = [a]

unit :: a -> [a]
unit = \x -> [x]

bind :: [a] -> (a -> [b]) -> [b]
x `bind` y = case x of
  [] -> []
  (a:x) -> (y a) `merge_res` (x `bind` y)

zero :: [a]
zero = []

plus :: [a] -> [a] -> [a]
plus = merge_res

merge_res :: [a] -> [a] -> [a]
merge_res x [] = x
merge_res [] y = y
merge_res (x:xs) (y:ys)
  | x < y    = x:merge_res xs (y:ys)
  | y < x    = y:merge_res (x:xs) ys
  | otherwise = x:merge_res xs ys
```

Figure 14  The list monad with zero and plus

The deterministic recognizers can now be transformed into non-deterministic ones by modifying their type

type Recognizer a = a -> [a],

and by replacing unit, bind, zero, and plus of the exception monad with the corresponding operators of the list monad. If these changes are made the recognizer s from the previous subsection behaves as follows.
4 Combining monads

4.1 Construction of a combined monad

So far in this chapter we have discussed recognizers that take as an argument a string of characters to be processed. Suppose that we decide to modify them so that they accept two parameters: the whole input string and a single position that specifies which character should be processed first. As we have shown in chapter 1, the type of such processors can be expressed in terms of the type Recognizer as follows

\[
\text{type RecognizerNew a = String} \to \text{Recognizer a.}
\]

The additional effect which recognizers of the type RecognizerNew incorporate can be represented using the state-reader monad \([\text{WA90, WA92}].\) Owing to the fact that we would like to add this effect on top of the effects that are already in place (exceptions or lists), we will use the parametrized state reader monad \([\text{LI95}]\) which is shown on Figure 15.

Assume that we already have a monad \((M, \text{unit}, \text{bind})\). A monad parametrized over \(M\) can be constructed by defining a new bind operation, in terms of the old bind, and by defining a function \(\text{lift}\),

\[
\text{lift} :: M a \to MN\text{ew} a
\]

that lifts operations of the type \(M a\) in into operations of the new type \(MN\text{ew} a\). If the underlying monad has zero and plus then the
corresponding operators zero and plus for the new monad can be defined in terms of the old ones. The main advantage of this technique is that the definition of the operators unit, bind, zero, and plus of a combined monad are independent of the choice of the base monad.

4.2 The parametrized state reader monad

The state reader monad abstracts over computations that read from the state but never update it. Owing to the fact that the “post-state” is always assumed to be identical to the “pre-state”, there is no need to return it. The definition of the parametrized state reader monad is given on Figure 15.

```
type Strm m s a = s -> m a

liftStrm :: m a -> Strm m s a
liftStrm x = \t -> x

unitStrm :: a -> Strm m s a
unitStrm x = liftStrm (unit x)

bindStrm :: Strm m s a -> (a -> Strm m s b) -> Strm m s b
(bindStrm k) t = a t `bind` \va ->
                  k va t

zeroStrm :: Strm m s a
zeroStrm = \s -> zero

plusStrm :: Strm m s a -> Strm m s a -> Strm m s a
(plusStrm x y) s = x s `plus` y s
```

Figure 15 The parametrized state reader monad

44
The type constructor StRM defines the type of computations that are applied to some state (of the type s) and return a computation of the type m a. The operations unit and bind are defined in terms of the unit and bind of the underlying monad. Although the state reader monad does not have its own plus and zero, these operations can be defined in terms of the corresponding operations of the base monad.

We can now define the type of our recognizers in terms of the type of the parametrized state reader monad.

\[
\text{type Recognizer } a = a \rightarrow \text{StRM } m \text{ String } a
\]

That is our new recognizers are functions that applied to a value of the type a (current position) and a state (an input string) return a computation of the type m a. The definitions of fail, empty, then, and orelse do not have to be modified (except for the fact that the operators unit, bind, zero, and plus are now of the different monad). The only function that must be modified is the recognizer term.

\[
\text{termR :: Char } \rightarrow \text{ Int } \rightarrow \text{ StRM } m \text{ String } \text{ Int}
\]

\[
\text{termInt } c \times s \\
| (x<1) | | (x>\text{length } s) | | (s!!(x-1) /= c) = \text{ zero} \\
| \text{otherwise} = \text{ unit } (x+1)
\]

4.3 Deterministic recognizers

If the base monad of the parametrized state reader monad is the monad for exceptions, the type of the recognizers is

\[
\text{type Recognizer } a = a \rightarrow \text{StRM } \text{Ex String } a.
\]
In other words, our new recognizers when applied to a single start position and an input string of characters either return a value of the form Ok v (that represents a single start position for the next processor), or they return the value Fail (that represents failure). One advantage of defining the new recognizers this way is that the previously given definitions of fail, then, orelse, and empty do not have to be modified. We simply replace the operators unit, bind, plus, and zero with the corresponding operators of the parametrized state reader monad.

The recognizer $s$ defined earlier, which is now of the type

\[ \text{Int} \rightarrow \text{StRM Ex String Int} \]

behaves as follows.

```plaintext
?s 1 "aaaaa"
Ok 6
```

The result Ok 6 returned by the recognizer means that the whole input string has been successfully recognized as an $s$ (the position 6 of the input string is the end of the string).

### 4.4 Non-deterministic recognizers

If the base monad of the parametrized state reader monad is the monad for lists, the type of the recognizers is

```plaintext
type Recognizer a = a \rightarrow \text{StRM \{\} String a}.
```
The recognizer $s$ when applied to a start position and the whole input string returns now a set of possible start positions for the next processor.

$s \oplus ^{1} \ "aaaaa"$

$[1, 2, 3, 4, 5, 6]$

5 Concluding remarks

The technique described in this chapter allows one to construct programs out of components that represent various programming language features. The technique can be implemented in any purely functional programming language. However, in a programming language that does not support overloaded operators: 1) different names must be used for each operator unit, bind, plus, and zero, 2) for each combination of features the corresponding operators must be defined explicitly. It is up to a programmer to determine which definitions of the operators to use in a given context.

The system of constructor classes in Gofer [JNA93] allows one to define classes of types with overloaded operators. The main advantage of using constructor classes to implement "combined" monads is that there is no need to use different names for the operators such as bind and unit if they are used in different contexts (that is, if they are parts of definitions of different monads). The type checker automatically determines which definition of bind or unit to use. The construction of language processors using constructor classes is described in the next chapter.
Chapter 5

MONADIC CONSTRUCTION OF MEMOIZED LANGUAGE PROCESSORS USING TYPE CONSTRUCTOR CLASSES IN GOFER

1 The system of type constructor classes in Gofer

The system of type constructor classes in Gofer allows one to define classes of types with overloaded operators [JNA93]. Overloading enables the definition and use of functions in which the meaning of a function symbol may depend on the types of its arguments.

Classes can be related in a class hierarchy. For example one class may be defined as a "subclass" of another (one of its "superclasses"), or it may be composed of other classes. For each class a set of suitable operations (methods) can be defined. A subclass inherits all of the methods of its superclasses.

2 Monads and type constructor classes

Each monad is a triple (M, unit, bind) where the types of the two operators unit and bind are defined in terms of the type constructor M. These types always have the same structure (no matter what feature the monad represents). Therefore we can define a monad as a class parametrized over the type constructor M, with two methods: unit and bind.
As we have shown in the previous chapter for some monads it makes sense to define two additional operators: `plus` and `zero`. The structure of the type of these two operators is "effect" independent. We can define a class `MonadPlusZero` as a subclass of the class `Monad` that has two additional methods: `plus` and `zero`.

```haskell
class Monad m where
  unit :: a -> m a
  bind :: m a -> (a -> m b) -> m b

class Monad m => MonadPlusZero m where
  plus :: m a -> m a -> m a
  zero :: m a
```

Figure 16  Classes: Monad and MonadPlusZero

There are three parts in any class declaration. In the example above the first line (called the header) of the declaration introduces the name `Monad` for the class and indicates that the class has a single parameter, represented by the type variable `m`. The second part (the signature part) is a list of function (method) declarations. For each instance of the class `Monad` we can define two methods: `unit` and `bind`. The third part (not present in the example above) may contain default definitions of the methods. For example Figure 17 shows the class `Recognizer` with default definitions of the functions `emptyR`, `thenR`, `orElseR`, and `failR`.

All the basic monads defined in the previous chapter can be now represented as instances of the class `Monad` and `MonadPlusZero` (if applicable). Instances of a type class in Gofer are defined using declarations
similar to those used to define the corresponding type class. For example the
following declarations specify that Monad Ex and MonadPlusZero Ex are
instances of classes Monad m and MonadPlusZero m respectively

instance Monad Ex where
  unit = unitEx
  bind = bindEx

instance MonadPlusZero Ex where
  plus = plusEx
  zero = zeroEx

(where operations with the suffix "Ex" are those defined previously for the
exception monad).

Similarly, parametrized monads can be represented as instances that
inherit (denoted using the symbol "=>") the monad operations from the base
monad m.

instance Monad m => Monad (StRM m s) where
  unit = unitStRM
  bind = bindStRM

instance MonadPlusZero m
    => MonadPlusZero (StRM m s) where
  zero = zeroStRM
  plus = plusStRM

3 The class Recognizer

We have shown in the previous chapter that most of the basic
recognizers and combinators can be defined in terms of the operators unit,
bind, plus, and zero. Therefore it is convenient to define the class of
recognizers as a subclass of the class MonadPlusZero.
class MonadPlusZero m => Recognizer m a where
  termR :: Char -> a -> m a
  emptyR :: a -> m a
  orelseR :: (a -> m a) -> (a -> m a) -> (a -> m a)
  thenR :: (a -> m a) -> (a -> m a) -> (a -> m a)
  failR :: m a

  emptyR = unit
  (p `thenR` q) inp = p inp `bind` \x1 ->
    q x1 `bind` \x2 ->
    unit x2
  (p `orelseR` q) inp = (p inp) `plus` (q inp)
  failR = zero

Figure 17 The class Recognizer

One advantage of defining the recognizers this way is that once the underlying monad m is defined, the corresponding operators empty, then, orelse, and fail are defined automatically. The definition of term depends on the representation of the input (which is either a string, or a pair: an integer number representing a start position and a string). Therefore we can define two instances of the class Recognizer (where termInt and termChar are the two definitions of term that were given in the previous chapter).

instance MonadPlusZero (StRM m String)
  => Recognizer (StRM m String) Int where
  termR = termInt

instance MonadPlusZero m => Recognizer m String where
  termR = termChar

We can now define the recognizer s.

s = (termR 'a' `thenR` (s `thenR` s))
  `orelseR` emptyR
By simply changing the type definition of \( S \) we can change its behavior. For example if the recognizer \( S \) is of the type

\[
S : : \text{Recognizer } (\text{StrM } [ ] \text{ String}) \text{ Int} \\
\Rightarrow \text{Int} \rightarrow \text{StrM } [ ] \text{ String Int}
\]

then it behaves as follows

\[
S \ 1 \ "aaaaa" \\
[1, 2, 3, 4, 5, 6].
\]

If we change its type to

\[
S : : \text{Recognizer } (\text{StrM Ex String}) \text{ Int} \\
\Rightarrow \text{Int} \rightarrow \text{StrM Ex String Int}
\]

then the same recognizer applied to the same input returns value

\[
\text{ok 6}.
\]

More examples can be found in Appendix B.

4 Memoized recognizers

Memoization involves interaction with state. Stateful computations can be represented in purely functional programs by using the state monad [WA90, WA92]. We start by presenting the definition of this monad.

4.1 The state monad

The monad defined below can be used for adding state operations to a purely functional program. The type of programs that interact with the state is defined as the type of a function that takes as its parameter an initial state
and returns its value paired with the final state. The function \texttt{unit} takes a value and a state and returns the same value paired with the initial state. In other words, the function \texttt{unit} a is "an identity" state transformer. The function \texttt{bind} combines two "stateful" computations. First, the computation \texttt{x} is evaluated in the initial state \texttt{t}, next, the function \texttt{k} is applied to the value returned by \texttt{x} and to the new state \texttt{tx}.

```
type St s a = s -> (a, s)

unitSt :: a -> St s a
unitSt a = \t -> (a, t)

bindSt :: St s a -> (a -> St s b) -> St s b
     (a `bindSt` k) t = k va ta
     where
         (va, ta) = a t

instance Monad (St s) where
    unit = unitSt
    bind = bindSt
```

Figure 18  The state monad

4.2 The parametrized list monad

Owing to the fact that the purpose of memoization is to improve the efficiency of processors that return a set of possible results, the memoized recognizers involve a combination of two features: state and non-determinism. In order to combine these features we introduce the parametrized monad for lists.
type ListM m a = m [a]

unitLsM :: Monad m => a -> ListM m a
unitLsM x = liftLsM (unit x)

bindLsM :: Monad m => ListM m a -> (a -> ListM m b) -> ListM m b
x `bindLsM` k = x `bind` \x1 ->
  foldr plusLsM zeroLsM (map k x1)

liftLsM :: Monad m => m a -> ListM m a
liftLsM x = x `bind` \x1 ->
  unit [x1]

instance Monad m => Monad (ListM m) where
  unit = unitLsM
  bind = bindLsM

instance Monad m => MonadPlusZero (ListM m) where
  zero = zeroLsM
  plus = plusLsM

zeroLsM :: Monad m => ListM m a
zeroLsM = unit []

plusLsM :: Monad m => ListM m a -> ListM m a -> ListM m a
(x `plusLsM` y) = x `bind` \x1 ->
  y `bind` \x2 ->
  unit (x1 `merge_res` x2)

Figure 19  The parametrized list monad

If the base monad for the parametrized list monad is the monad for state then the combination of the two monads yields the computations that are of the type

ListM (St s) a = s -> ([a], s).
In other words each computation is applied to an initial state and returns a list of values paired with the modified state. (Note that combining the same monads in the reverse order yields computations of different type).

Our memoization algorithm applies to the recognizers that take as parameters a single position and the whole input string, therefore on top of the two features: state and non-determinism, they involve one more "effect" that can be represented using the parametrized state reader monad. We can define the type of memoized recognizers in terms of the types of the three monads as follows

\[
\text{type Recognizer a} = \text{Int} \to \text{StRM (ListM (St s)) String Int.}
\]

4.3 The memo-table

The introduction of the state monad does not immediately improve the performance of memoized recognizers. Operations that access the state are required to store and retrieve the previously computed results. We have decided to represent the state as a list of pairs. The first component of each pair is an integer number that acts as an index to the memo-table, the second component is a list of pairs (recognizer_name, recognizer_value). Owing to the fact that different processors may return values of different types, it is convenient to parametrize the type of the memo-table over the type of values it stores.

\[
\text{type State v} = \{(\text{Int}, \{(\text{String}, v)\})\}
\]
The purpose of the function `lookupSt` is to return the value that corresponds to a start position and a recognizer name (given as parameters). The function `updateSt` given a start position, recognizer name, and the value returned by the recognizer, updates the corresponding entry of the memo-table. The function `newSt` creates a new memo-table. The definitions of these functions can be found in Appendix B.

The top level function `memoize` is applied to each recognizer to store its result in the memo table. The function is defined in terms of the `unit` and `bind` operators of the state monad.

```haskell
memoizeRec ::
Recognizer (StRM (ListM (St (State [Int])))) String Int
    => String
  -> (Int -> StRM (ListM (St (State [Int])))) String Int
  -> (Int -> StRM (ListM (St (State [Int])))) String Int

memoizeRec name f i s
    = lookupSt i name `bindSt` \x1 \x2 ->
        if x1 /= []
            then unitSt (x1!!0)
            else f i s `bindSt` \x2 ->
                updateSt i name x2 `bindSt` () ->
                unitSt x2
```

The definition of the function `memoize` corresponds closely to the memoization algorithm. In order to memoize the recognizer `f` that is applied to a start position `i` and the whole input string of tokens `s` we first search the memo-table for the result that corresponds to the recognizer name and the start position `i`. If the result was computed before this result is returned. Otherwise, the new result is computed and the memo-table is updated.
The memoized recognizer \( ms \) defined as

\[
ms :: \text{Recognizer} \\
(\text{StRM} \ (\text{ListM} \ (\text{St} \ (\text{State} \ [\text{Int}]))) \ \text{String}) \ \text{Int} \\
\rightarrow \text{Int} -> \text{StRM} \ (\text{ListM} \ (\text{St} \ (\text{State} \ [\text{Int}]))) \ \text{String} \ \text{Int}
\]

\[
ms = \text{memoizeR} \ "ms" \ ((\text{termR} \ 'a' \ \text{thenR} \ (ms \ \text{thenR} \ ms)) \\
\text{"orelseR" emptyR})
\]

when applied to a string "aaaaa" at position 1 returns the following result.

\[
(1, 2, 3, 4, 5, 6], [(1, [("ms", [1, 2, 3, 4, 5, 6])]), \\
(2, [("ms", [2, 3, 4, 5, 6])]), \\
(3, [("ms", [3, 4, 5, 6])]), \\
(4, [("ms", [4, 5, 6])]), \\
(5, [("ms", [5, 6])]), \\
(6, [("ms", [6])])]
\]

5 Parsers

5.1 The type of the monadic parsers

The monadic approach allows one to easily extend the techniques presented in the previous sections to more complex language processors. Recall once again the type of parsers:

\[
\text{type} \ \text{Parser} \ a = s -> (a, s).
\]

The type of parsers corresponds directly to the type of the state monad where the state is a sequence of tokens to be processed. Such a monad is sometimes called the input monad \([\text{WA90, WA92}]\). Similarly to the type of monadic recognizers we can express the type of parsers in terms of the "monadic" type constructor \( m \).
type Parser a = m a

5.2 The class Parser

Basic parsers and combinators can be defined in terms of the operators unit, bind, plus, and zero. We can define the class of parsers as a subclass of the class MonadPlusZero.

```
class MonadPlusZero m => Parser m a where
    term :: Char -> m a
    empty :: a -> m a
    orelse :: m a -> m a -> m a
    fail :: m a

    empty = unit
    orelse = plus
    fail = zero
```

Figure 20  The class Parser

(Note that we have not defined the operator then for parsers. The reason for that is that the monadic operator bind which integrates the sequencing of functions with the processing of their values is much more convenient to use.)

Having defined the class of parsers we can build various parsers in a modular way by combining appropriate monads. In addition to the monads defined so far, we will use the parametrized input monad which is described in the next subsection.)
5.3 The parametrized input monad

The input monad is identical to the monad for state where the state is a sequence of tokens. The type of programs is the type of functions from a string (the input to the program) to a computation that involves a pair: value and a string (the input to the rest of the program).

```haskell
type InpM m s a = s -> m (a, s)

unitInpM :: Monad m => a -> InpM m s a
unitInpM a = liftInpM (unit a)

bindInpM :: Monad m
    => InpM m s a -> (a -> InpM m s b) -> InpM m s b
    (a `bindInpM` k) inp = a inp `bind` \(va, outa) ->
    k va outa

liftInpM :: Monad m => m a -> InpM m s a
liftInpM x inp = x `bind` \x1 ->
    unit (x1, inp)

zeroInpM :: MonadPlusZero m => InpM m s a
zeroInpM = \inp -> zero

plusInpM :: MonadPlusZero m
    => InpM m s a -> InpM m s a -> InpM m s a
    (x `plusInpM` y) = \inp -> x inp `plus` y inp

instance Monad m => Monad (InpM m s) where
    unit = unitInpM
    bind = bindInpM

instance MonadPlusZero m
    => MonadPlusZero (InpM m s) where
    zero = zeroInpM
    plus = plusInpM
```

Figure 21 The parametrized input monad
If the underlying monad of the input monad is the monad for lists, the

\[ \text{InpM \ [] \ s \ a = s \rightarrow [(a, s)], \} \]

which is exactly the same as the type of non-deterministic parsers defined
earlier. If the base monad is the exception monad, the resulting parsers are
deterministic, they either return a value of the form \text{Ok} (a, s), or they
return the value \text{Fail} that denotes failure.

To obtain parsers that take as parameters a single position and the
whole input string of tokens we simply apply the parametrized state reader
monad on top of the parametrized input monad. The type of the resulting
processors is

\[ \text{StRM (InpM m Int) String a = String \rightarrow Int \rightarrow m (a, Int).} \]

That is each parser when applied to an the input string of tokens and a single
position returns a computation that involves a pair: the value of the parser
and the start position for the next parser. The corresponding parser \text{term} can
be defined as follows,

\[
\text{termEInt :: MonadPlusZero m} \\
\quad \text{=> Char \rightarrow StRM (InpM m Int) String Int} \\
\text{termEInt c s x} \\
\quad | (x<1) | | (x>length s) | | (s!!(x-1) /= c) = zero \\
\quad | otherwise = unit (eval c, x+1)
\]
where the function \texttt{eval} is application dependent, and returns the value that corresponds to a character given as its parameter. The corresponding instance of the class \texttt{Parser} is defined below.

\begin{verbatim}
instance MonadPlusZero (StrRM (InpM m Int) String) => Parser (StrRM (InpM m Int) String) Int where
term = termEInt
\end{verbatim}

New parsers or evaluators can now be constructed by combining simpler processors using the operators bind and otherwise. For example an evaluator with the structure that corresponds to the recognizer \texttt{s} defined earlier can be constructed as follows.

\begin{verbatim}
e = (term 'a' `bind` \x1 ->
    e `bind` \x2 ->
    e `bind` \x3 ->
    unit (x1 + x2 + x3))
  `otherwise`
  empty 0
\end{verbatim}

Suppose that the function \texttt{eval} is defined as \texttt{eval 'a' = 1}. If the type of the evaluator \texttt{e} is defined as

\begin{verbatim}
e :: Parser (StrRM (InpM [] Int) String) Int
  => StrRM (InpM [] Int) String Int
\end{verbatim}

then the evaluator applied to the string “aaaaa” at position 1 returns a set of values.

\begin{verbatim}
[(0,1), (1,2), (2,3), (3,4), (4,5), (5,6)]
\end{verbatim}

If the type of the same evaluator is defined as
e :: Parser (StrM (InpM Ex Int) String) Int
    => StrM (InpM Ex Int) String Int

then the same evaluator returns a single value.

Ok (5,6)

5.4 Memoized language processors

The above processors can be memoized by replacing the monad for exception or non-determinism with the combined monad for list and state (exactly the same combination as was previously introduced to memoize recognizers). The values stored in the memo-table are now lists of pairs of integer numbers. The first component of each pair is the value returned by the evaluator, the second component is the start position for the next processor. The type of the resulting evaluators is

(StrM (InpM (ListM (St (State [(Int,Int)]))) Int) String) Int.

Although the above type definition may look a little complex, this is in fact the only addition that is required to obtain memoized processors. A memoized evaluator me can now be defined as

me = memoize "me" ((term 'a' `bind` \x1 ->
    me `bind` \x2 ->
    me `bind` \x3 ->
    unit (x1 + x2 + x3))
    `orelse``
    empty 0)

When applied to the string "aaaaaa" at position 1 it returns a pair: set of values and a memo-table.
6 Complexity of memoized language processors

A detailed complexity analysis of memoized functional recognizers can be found in Appendix A. The analysis there holds for monadic recognizers as described in this chapter. Each memoized recognizer is built up out of three monads: the list monad, the state monad, and the state reader monad. The use of the list monad guarantees that each processor is applied to all elements of the input list, and the corresponding results are combined using the function merge_res (that merges two lists sorted in ascending order, removing duplicates). The same function merge_res is used to combine results of alternate processors (orelse). The use of the state monad to implement memoization guarantees that the memo-table is passed to each recognizer as a parameter and the modified memo-table is returned as its result. Finally, the state reader monad enables accessing current position of the input string.

The memo-table is structured as a list of \((n + 1)\) pairs where the first element of each pair is an integer number \(i\) representing the start position; the second element is a list of results corresponding to each application of a recognizer at the position \(i\). Each result is a list of at most \((n + 1)\) integer numbers. The number of recognizers is independent on the size of the input,
therefore the size of the memo-table is $O(n^2)$. Owing to the fact that purely functional programming languages do not (in general) allow variables to be destructively updated, each update of the memo-table results in the creation of the new modified memo-table. The number of possible updates is linear in the size of the input, therefore the space required by the algorithm is $O(n^3)$.

This performance can be improved in a purely functional language that supports updatable objects. In his first paper about monads [WA90] Wadler noticed that it is possible, within the monadic framework, to add updatable in-place arrays to purely functional programming languages without compromising strong reasoning principles valid for these languages. He proposed to implement such arrays as an abstract data type with a set of well-defined operations. The encapsulation of an array guarantees that the programmer cannot duplicate it. Combined with the use of monadic sequencing, it guarantees single threading of the array through the program. Based on the Wadler's work on monads, Launchbury and Peyton Jones presented a way of securely encapsulating stateful computations that manipulate multiple mutable objects [LA93, LA94, LA95]. Updatable variables are currently implemented in the Glasgow Haskell Compiler.

Time and space complexity of more complex processors such as parsers or syntax-directed evaluators is application dependent. If the number of different results to be returned is exponential in the length of input, the corresponding language processor will have exponential complexity. One advantage of the monadic approach to building memoized language
processors is that the programmer can easily switch on and off the features the processors incorporate (by simply redefining the type of the processor). In other words, depending on the application a memoized or non-memoized version of the processors can be used.
CONCLUSION

In this thesis we have described how memoization can be implemented to improve efficiency of purely functional language processors. An important contribution of this thesis is the monadic specification of such processors. Monads provide a general technique of adding various features to purely functional programs. We have shown how memoization can be treated as one such feature.

The monadic approach allows one to abstract over features a program incorporates. A monadic program is built from components of an abstract type m a. To add a new "effect" to the program one simply changes the meaning of "m" and then adds or modifies only those components that deal directly with the "effect" being added. Assuming the correctness of the initial program, only parts that have been added or modified must be tested.

The program given in Appendix B illustrates how expressive, modular, and easy to modify applications one can develop using the system of constructor classes and monads. The program consists of a library of monads and a library of simple monadic language processors and combinators. New language processors can be built from simpler components by combining
them using higher-order combinators. The behavior of such processors can be changed by simply modifying their types.

We have used the technique proposed by Liang, Hudak, and Jones [LI95] to construct memoized language processors, in a fully modular way, from components that represent various "effects". We have been pleasantly surprised by the flexibility of this method, in particular, by the ease with which it was possible to extend the technique for memoizing purely functional recognizers to more complex language processors. The extended technique can be used to memoize parsers as well as programs that are constructed as executable attribute grammars [FR92, FR95].
NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.


Appendix A

MEMOIZING PURELY-FUNCTIONAL TOP-DOWN BACKTRACKING LANGUAGE PROCESSORS

Richard A. Frost and Barbara Szydlowski School of Computer Science, University of Windsor, Windsor, Ontario, Canada N9B 3P4
November 1996

ABSTRACT

Language processors may be implemented directly as functions. In a programming language that supports higher-order functions, large processors can be built by combining smaller components using higher-order functions corresponding to alternation and sequencing in the BNF notation of the grammar of the language to be processed. If the higher-order functions are defined to implement a top-down backtracking parsing strategy, the processors are modular and, owing to the fact that they resemble BNF notation, are easy to understand and modify. A major disadvantage of this approach is that the resulting processors have exponential time and space complexity in the worst case owing to the reapplication of processors during backtracking. This paper shows how a technique called memoization can be used to improve efficiency of such processors whilst preserving their modularity. We show that memoized functional recognizers constructed for arbitrary non-left-recursive grammars have $O(n^3)$ complexity where $n$ is the length of the input to be processed. The paper also shows how the initial processors could have been memoized using a monadic approach, and discusses the advantages of reengineering the definitions in this way.

---

$^2$ This paper appeared in the Journal “Science of Computer Programming” (FS96).
1 Constructing modular non-deterministic language processors in functional programming languages

One approach to implementing language processors in a modern functional programming language is to define a number of higher-order functions which when used as infix operators (denoted in this paper by the prefix $\$\$) enable processors to be built with structures that have a direct correspondence to the grammars defining the languages to be processed. For example, the function \texttt{s}, defined in the functional program in Fig. 22, is a recognizer for the language defined by the grammar \texttt{s} ::= 'a' \texttt{s} | empty if the functions term, orelse, then, and empty are defined as shown in the next few pages of this paper.

\begin{verbatim}
  s  = (a $then s $then s) orelse empty
  a  = term 'a'
\end{verbatim}

Figure 22 A functional program containing a definition of the recognizer \texttt{s}

This approach, which is described in detail in Hutton [10], was originally proposed by Burge [2] and further developed by Wadler [18] and Fairburn [4]. It is now frequently used by the functional-programming community for language prototyping and natural-language processing. In the following, we describe the approach with respect to language recognizers although the technique can be readily extended to parsers, syntax-directed evaluators and executable specifications of attribute grammars [1, 6, 7, 12].
According to the approach, recognizers are functions mapping lists of inputs to lists of outputs. Each entry in the input list is a sequence of tokens to be analyzed. Each entry in the output list is a sequence of tokens yet to be processed. Using the notion of "failure as a list of successes" [18] an empty output list signifies that a recognizer has failed to recognize the input. Multiple entries in the output occur when the input is ambiguous. In the examples in this paper it is assumed that all tokens are single characters. The notation of the programming language Miranda\(^3\) [17] is used throughout, rather than a functional pseudo-code, in order that readers can experiment with the definitions directly.

The types token and recognizer may be defined as follows where = = means "is a synonym for", x -> y denotes the type of functions from objects of type x to objects of type y, and square brackets denote a list.

\[
\text{token} == \text{char} \\
\text{recognizer} == [[\text{token}]] \rightarrow [[\text{token}]]
\]

That is, a recognizer takes a list of lists of tokens as input and returns a list of lists of tokens as result. Note that this differs from the type found in many other papers on functional recognizers. The reason for this difference is that it simplifies the memoization process as will be explained later.

The simplest type of recognizer is one that recognizes a single token at the beginning of a sequence of tokens. Such recognizers may be constructed using the higher-order function term defined below. The notation x :: y

\(^3\) Miranda is a trademark of Research Software Ltd.
Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors

decares \( x \) to be of type \( y \). The function \texttt{concat} takes a list of lists as input and concatenates the sublists to form a single list. \texttt{map} is a higher-order function which takes a function and a list as input and returns a list that is obtained by applying the function to each element in the input list. Function application is denoted by juxtaposition, i.e. \( f \ x \) means \( f \) applied to \( x \). Function application has higher precedence than any operator, and round brackets are used for grouping. The empty list is denoted by \( \[] \) and the notation \( x : y \) denotes the list obtained by adding the element \( x \) to the front of the list \( y \). The applicable equation is chosen through pattern matching on the left-hand side in order from top to bottom, together with the use of guards following the keyword \texttt{if}.

\[
\text{term} :: \text{token} \to \text{recognizer}
\text{term} \ c \ \text{inputs} = (\text{concat} \ . \ \text{map} \ \text{test\_for\_c}) \ \text{inputs}
\text{where}
\text{test\_for\_c} \ [\] = [\]
\text{test\_for\_c} \ (t:ts) = [ts], \text{if} \ t = c
\text{test\_for\_c} \ (t:ts) = [\] , \text{if} \ t \neq c
\]

The following illustrates use of \texttt{term} in the construction of two recognizers \( c \) and \( d \), and the subsequent application of these recognizers to three inputs. The notation \( x \Rightarrow y \) is to be read as "\( y \) is the result of evaluating the expression \( x \)". The empty list in the second example signifies that \( c \) failed to recognize a token 'c' at the beginning of the input "xyz". The notation "\( x_1 \ldots x_n \)" is shorthand for \( [x_1, \ldots, x_n] \).

\[
c = \text{term 'c'}
\]

74
Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors

d = term 'd'
c ["cxyz"] => ["xyz"]
c ["xyz"] => []
d ["dabc", "dxyz"] => ["abc", "xyz"]

Alternate recognizers may be built using the higher-order function `orelse` defined below. The operator `++` appends two lists.

`orelse :: recognizer -> recognizer -> recognizer`
(p $orelse q) inputs = p inputs ++ q inputs

According to this definition, when a recognizer p $orelse q is applied to a list of inputs inputs, the value returned is computed by appending the results returned by the separate application of p to inputs and q to inputs. The following illustrates use of `orelse` in the construction of a recognizer `c_or_d` and the subsequent application of this recognizer to three inputs.

c_or_d = c $orelse d
c_or_d ["abc"] = []
c_or_d ["cxyz"] => ["xyz"]
c_or_d ["dxyz"] => ["xyz"]

Sequencing of recognizers is obtained through use of the higher-order function then defined as follows:

`then :: recognizer -> recognizer -> recognizer`
(p $then q) inputs = [] , if r = []
= q r , otherwise
where
  r = p inputs

According to this definition, when a recognizer p $then q is applied to a list of inputs inputs, the result returned is an empty list if p fails when applied to
inputs, otherwise the result is obtained by applying q to the result returned by p. (Note that in general, then does not have the same effect as reverse composition. In particular, replacing p $\text{then } q$ by $q \cdot p$ will result in non-terminating computations for certain kinds of recursively-defined recognizers.) The following illustrates use of then in the construction of a recognizer c_then_d, and the subsequent application of c_then_d to two inputs:

\[
\begin{align*}
\text{c\_then\_d} & = \text{c } $\text{then } d \\
\text{c\_then\_d} ["\text{cdxy}"] & \Rightarrow ["\text{xy}"] \\
\text{c\_then\_d} ["\text{cxyz}"] & \Rightarrow []
\end{align*}
\]

The “empty” recognizer, which always succeeds and which returns the complete list of inputs as output to be processed, is implemented as the identity function:

\[
\text{empty inputs } = \text{ inputs}
\]

The functions term, orelse, then, and empty as defined above, may be used to construct recognizers whose definitions have a direct structural relationship with the context-free grammars of the languages to be recognized. Fig. 23 illustrates this relationship.
Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors

<table>
<thead>
<tr>
<th>BNF grammar of the language</th>
<th>The program</th>
</tr>
</thead>
<tbody>
<tr>
<td>s ::= 'a' s s</td>
<td>empty</td>
</tr>
<tr>
<td>terminals = {'a'}</td>
<td>a = term 'a'</td>
</tr>
</tbody>
</table>

Figure 23  The relationship between the grammar and the program implementing the recognizer

The example application given below illustrates use of the recognizer $s$ and shows that the prefixes of the input “aaa” can be successfully recognized in nine different ways. Empty strings in the output, denoted by “”, correspond to cases where the whole input “aaa” has been recognized as an $s$. The output shows that there are five ways in which this can happen. The two strings in the output consisting of “a” correspond to cases where the prefix “aa” has been recognized leaving “a” for subsequent processing. The output shows that there are two ways in which this can happen. The string in the output consisting of two letters “aa” corresponds to the case where the prefix “a” has been recognized leaving “aa” for subsequent processing. This can only happen in one way when “a” is recognized as an $s$.

s ["aaa"] => ["", "", ", "a", ", ", "a", ", "aa", ", "aaa"]

The major advantage of this approach is that the processors created are modular executable specifications of the languages to be processed. Components can be defined, compiled and executed directly. For example, (a $then s $then s) is a recognizer that may be executed directly as for example:

[a $then s $then s] ["bcd"] => []
[a $then s $then s] ["aab"] => ["b", "b", "ab"]

The advantages of building language processors using this technique come at a price. The processors employ a naive top-down fully-backtracking search strategy and consequently exhibit exponential-time and space behavior in the worst case. In the following, we show how this problem can be overcome through a process of memoization. We begin by discussing techniques that have been proposed by other researchers concerning the use of memoization with top-down backtracking language processors. We then describe how memoization can be achieved at the source-code level in purely-functional programming languages and show how the technique can be adapted for use to improve the efficiency of top-down backtracking recognizers. We provide a formal description of the algorithm and a proof of the complexity result. In addition, we show how the same result can be obtained in a more structured way by use of a monad. We conclude with a discussion of how the approach can be used with parsers and executable attribute grammars.

2 Memoizing language processors

Memoization [9, 14] involves a process by which functions are made to automatically recall previously-computed results. Conventional implementations involve maintenance of memo-tables which store previously-computed results. Application of a memoized function begins by reference to its memo-table. If the input has been processed before, the previously-computed result is returned. If the input has not been processed
before, the result is computed using the original definition of the function, with the exception that all recursive calls use the memoized version, the memo-table is then updated and the result returned.

Many of the efficient algorithms for recognition and parsing make use of some kind of table to store well-formed substrings of the input and employ a form of memoization. Earley's algorithm [3] is an example. In most of these algorithms, the parsing and table update and lookup are intertwined. This results in relatively complex processors that are not modular. Norvig [16] has shown how memoization can be used to obtain a modular processor, with properties similar to Earley's algorithm, by memoizing a simple modular top-down backtracking parser generator. Norvig's memoized parser generator cannot accommodate left-recursive productions but would appear to be as efficient and general as Earley's algorithm in all other respects. According to Norvig, the memoized recognizers have cubic complexity compared to exponential behavior of the original unmemoized versions.

In Norvig's technique, memoization is implemented at the source-code level in Common Lisp through definition and use of a function called memoize. When memoize is applied to a function f, it modifies the global definition of f such that the new definition refers to and possibly updates a memo-table. A major advantage of Norvig's approach is that programs may, in some cases, be made more efficient with no change to the source-code definition. In Norvig's approach, both the process of memoizing a function,
and the process of updating the memo-table, make use of Common Lisp's updateable function-name space. This precludes direct use of Norvig's approach when language processors are to be constructed in a purely-functional programming language where updateable objects are not permitted.

Leermakers [12] and Augusteijn [1] have also described how memoization can be used to improve the complexity of functional top-down backtracking language processors but have not indicated how the memoization process itself would be achieved. In particular, they have not addressed the question of how memoization would be achieved in a purely-functional implementation of the language processors.

3 Memoization in purely-functional languages

A functional programming language is one in which functions are first-class objects and may, for example, be put in lists, passed to other functions as arguments, and returned by functions as results. A purely-functional language, such as Miranda [17], LML [5], and Haskell [8], is one in which functions provide the only control structure and side-effects, such as assignment, are not allowed. This restriction is a necessary condition for referential transparency, a property of programs that simplifies reasoning about them and which is one of the major advantages of the purely-functional programming style [19].
Owing to the fact that side-effects are forbidden, purely-functional languages do not accommodate any form of updateable object. Consequently, Norvig's technique for improving the efficiency of top-down backtracking language processors cannot be implemented directly in any purely-functional language. However, we can adapt Norvig's approach if we use a variation of memoization that has been described by Field and Harrison [5] and investigated in detail by Khoshnevisan [11]. This memoization technique differs from conventional approaches in that memo-tables are associated with the inputs to and outputs from functions, rather than with the functions themselves. A function may be memoized by modifying its definition to accept a table as part of its input, to refer to this table before computing a result, and to update the table before returning it as part of the output. The memo-table is passed as an input to the top-level call of recursively defined functions and is threaded through all recursive calls. To illustrate this technique, we show how the Fibonacci function can be memoized. We begin with a textbook definition given in Fig. 24.

\[
\begin{align*}
\text{fib} 0 &= 1 \\
\text{fib} 1 &= 1 \\
\text{fib} n &= \text{fib} (n - 1) + \text{fib} (n - 2), \text{ if } n \geq 2
\end{align*}
\]

Figure 24 Definition of the Fibonacci function

Defined in this way, evaluation of the Fibonacci function has exponential complexity. The cause of the exponential behavior is the
replication of computation in the two recursive calls. This replication can be avoided by memoization. We begin by modifying the definition of \texttt{fib} so that it accepts a table as part of its input and returns a table as part of its result. In the modified definition, round brackets and commas are used to denote tuples. The table \texttt{t1}, which is output from the first recursive call of \texttt{tfib}, is passed as input to the second recursive call of \texttt{tfib}. The table \texttt{t2} which is output from the second recursive call is returned as result from the top-level call of \texttt{tfib}:

\begin{align*}
\texttt{tfib} (0, t) &= (1, t) \\
\texttt{tfib} (1, t) &= (1, t) \\
\texttt{tfib} (n, t) &= (r1 + r2, t2) \\
& \quad \text{where} \\
& \quad (r1, t1) = \texttt{tfib} (n - 1, t) \\
& \quad (r2, t2) = \texttt{tfib} (n - 2, t1)
\end{align*}

Note that \texttt{tfib} still has exponential behavior. When applied to an input, it returns the table unchanged. Rather than modifying the definition of \texttt{tfib} directly to make use of the memo-table, as is done in Field and Harrison and in Khoshnevisan, we choose to abstract the table-lookup and update process into a general-purpose higher-order function \texttt{memo} which we can apply to \texttt{tfib} to obtain a memoized version. This variation is comparable to Norvig's technique. When \texttt{memo} is applied to a function \texttt{f} it returns a new function \texttt{newf} whose behavior is exactly the same as \texttt{f} except that it refers to, and possibly updates, the memo-table given in the input.

In the definition of the function \texttt{memo} below, the expression \texttt{mr $pos 1} denotes the first element of the list of memorized results \texttt{mr}. The
Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors

definition of lookup makes use of a list comprehension, \([r \mid (y, r) \leftarrow t; y = i]\), which is to be read as “the list of all \(r\) such that the pair \((y, r)\) is a member of the table \(t\) and \(y\) is equal to the index \(i\).”

\[
\begin{align*}
\text{memo } f &= \text{newf} \\
&\quad \text{where} \\
&\quad \text{newf } (i, t) = (r1, t1) \\
&\quad \quad \text{where} \\
&\quad \quad (r1, t1) = (mr \& \text{pos } 1, t) \quad \text{if } mr \neq [] \\
&\quad \quad \quad = (r2, \text{update } i \ r2 \ t2) \quad \text{if } mr = [] \\
&\quad \quad (r2, t2) = f (i, t) \\
&\quad \quad mr = \text{lookup } i \ t
\end{align*}
\]

\[
\begin{align*}
\text{update } i \ r \ t &= (i, r) : t \\
\text{lookup } i \ t &= [r \mid (y, r) \leftarrow t; y = i]
\end{align*}
\]

We can now complete the process of memoizing the Fibonacci function by applying \text{memo} to the two recursive calls in the definition of \text{fFib} as shown in Fig. 25. The result is a function called \text{mFib} which has linear complexity.

\[
\begin{align*}
\text{mFib } (0, t) &= (1, t) \\
\text{mFib } (1, t) &= (1, t) \\
\text{mFib } (n, t) &= (r1 + r2, t2) \\
&\quad \text{where} \\
&\quad \quad (r1, t1) = \text{memo mFib } (n - 1, t) \\
&\quad \quad (r2, t2) = \text{memo mFib } (n - 2, t1)
\end{align*}
\]

\[
\text{mFib } (4, []) => [5, [(3, 3), (2, 2), (0, 1), (1, 1)]]
\]

Figure 25  A memoized version of the Fibonacci function

Some readers may realize that it is only necessary to store the two most-recently computed values of the Fibonacci sequence in the memo-table.
Modifying the function `update` accordingly would decrease the space requirements of `mfib` but would improve neither time nor space complexity. It should also be noted that there are many other ways to improve the complexity of the Fibonacci function. We do not claim that the use of memoization is the most appropriate technique in this application. We have chosen to use the Fibonacci function as an example so that our technique can be easily compared with that described by Norvig who also used the Fibonacci example for expository purposes.

The technique described above is not as elegant as Norvig’s in the sense that the process of memoization has resulted in changes to the definition of the Fibonacci function at the source-code level. Later we show how to reduce the number of changes required for memoization and limit them to local changes only.

4 Memoizing purely-functional recognizers

A memoized functional recognizer is a function that takes, as an extra parameter, a memo-table containing all previously computed results. One approach to memoization is to modify the definitions of the functions `term`, `sorelse`, and `then`, so that the recognizers built with them accept a memo-table as part of their input and return a memo-table as part of their output. Next, a higher-order function `memoize` is applied to each recognizer to create a memoized version of it.
4.1 The memo-table

In order to improve efficiency we have chosen to store the input sequence of tokens in the memo-table and to represent the points at which a recognizer is to begin processing by a list of numbers which are used as indexes into that sequence.

The memo-table is structured as a list of triples of length \( n + 1 \), where \( n \) is the length of the input sequence:

```python
memo_table == [(num, token, [(rec_name, [num])])]
rec_name == [char]
```

The last element of the memo-table is a special token \# representing the end of the input. The first component of the ith triple is an integer \( i \). This number acts as an index into memo-table entries. The second component is the ith token in the input sequence. The third component is a list of pairs representing all successful recognitions of the input sequence starting at position \( i \). The first component of each pair is a recognizer name, the second component is a list of integer numbers. The presence of a number \( j \), where \( i \leq j \leq n + 1 \) in this list indicates that the recognizer succeeded when applied to the input sequence beginning at position \( i \) and finishing at position \( j - 1 \).

Initially, the third component of each triple in the memo-table is an empty list. The following example shows the initial table corresponding to the input “aaa”.

```python
[(1, 'a', []), (2, 'a', []), (3, 'a', []), (4, '#', [])]
```
Two operations are required for table lookup and update. The operation lookup applied to an index i, a recognizer name name and a memo-table t returns a list of previously computed end positions where the recognizer name succeeded in processing the input beginning at position i. The operation update applied to an index i, a result res and a memo-table t, returns a new memo-table with the ith entry updated. A result is a pair consisting of a recognizer name and a list of successful end-positions. Update adds the result res to the list of successful recognitions corresponding to the ith token.

```plaintext
lookup i name t
  = [], if i > #t
  = [bs | (x, bs) <- third (t $pos i); x = name], otherwise
    where
      third (x, y, z) = z

update i res t
  = map (add_res i res) t
    where
      add_res i res (x, term, res_list)
        = (x, term, res:res_list), x = i
        = (x, term, res_list), otherwise
```

The function memoize takes as input a recognizer name n, a recognizer f, a list of positions where the recognizer should begin processing the input, and a memo-table. For each start position in the list, the function memoize first calls the function lookup to determine if this application of the recognizer has been computed previously. If lookup returns an empty list, the recognizer is applied, a new result is calculated and the function update is used to add the result to the memo-table. Otherwise the
previously computed result is returned. Results returned for each of the start
positions are merged with the removal of duplicates.

\[
\text{memoize } n \ f \ (\ [], \ t) \ = \ (\ [], \ t) \\
\text{memoize } n \ f \ (b:bs, \ t) \\
\quad = \ (\text{merge\_res } rl \ rs, \ trs) \\
\quad \quad \text{where} \\
\quad \quad (r1, \ t1) = (mr \ \$\text{pos} \ 1, \ t), \ \text{if } mr = \ [] \\
\quad \quad \quad = (r2, \ \text{update } b \ (n, \ r2) \ \ t2), \ \text{otherwise} \\
\quad \quad (r2, \ t2) = f \ ([b], \ t) \\
\quad (rs, \ trs) = \text{memoize } n \ f \ (bs, \ t1) \\
\quad mr = \text{lookup } b \ n \ t
\]

\[
\text{merge\_res } x \ [\ ] = x \\
\text{merge\_res } [\ ] \ y = y \\
\text{merge\_res } (x:xs) \ (y:ys) = x:\text{merge\_res } xs \ (y:ys), \ \text{if } x < y \\
\quad = y:\text{merge\_res } (x:xs) \ ys, \ \text{if } y < x \\
\quad = x:\text{merge\_res } xs \ ys, \ \text{if } x = y
\]

4.2 The memoized recognizers

The definitions of term, \$\text{then}, and \$\text{orElse} given in section 1 are
modified to take as input a list of positions where the recognizer should
begin processing the input, and a memo-table. Owing to the fact that the
entire input sequence of tokens is represented in the memo-table, there is no
need for the recognizers to explicitly return unprocessed segments of the
input. Instead they return a number as index into the input sequence.

The next modification to the definitions of \$\text{orElse} and \$\text{then} is to
allow threading of the memo-table through recursive calls. The function
term is modified owing to the fact that the input sequence is now stored in
the memo-table. The function \text{merge} is used to combine and remove
duplicates that arise if the same segment of the input can be recognized in
more than one way by a recognizer. For recognition purposes such duplicates can be considered equal.

\[
\text{mterm } c \ (bs, t) \\
\quad = \ ((\text{concat} \ . \ (\text{map} \ \text{test}_\text{for}_c)) \ bs, t) \\
\quad \text{where} \\
\quad \text{test}_\text{for}_c \ b = [] \quad , \text{if } b > \#t \\
\quad \text{test}_\text{for}_c \ b = [] \quad , \text{if second} \ (t \ $\text{pos} \ b) = c \\
\quad \text{test}_\text{for}_c \ b = [b + 1] \quad , \text{if second} \ (t \ $\text{pos} \ b) = c \\
\quad \text{second} \ (x, \ y, \ z) = y \\
\quad (p \ $\text{m\_orelse} \ q) \ (bs, t) = (\text{merge\_res} \ rp \ rq, \ tq) \\
\quad \text{where} \\
\quad \ (rp, \ tp) = p \ (bs, \ t) \\
\quad \ (rq, \ tq) = q \ (bs, \ tp) \\
\quad (p \ $\text{m\_then} \ q) \ (bs, t) = q \ (rp, \ tp) \quad , \text{if } rp = [] \\
\quad \quad = ([], \ tp) \quad , \text{otherwise} \\
\quad \text{where} \\
\quad \ (rp, \ tp) = p \ (bs, \ t)
\]

These functions can now be used to improve the complexity of functional recognizers whilst preserving their structural simplicity and modularity. As example, Figure 26 shows the relationship between the original recognizer for the grammar \( S ::= 'a' \ S | \text{empty} \) and the memoized version. Note that it is not necessary to change the definition of empty nor is it necessary to memoize the recognizers constructed with \text{mterm}.

<table>
<thead>
<tr>
<th>The original recognizer</th>
<th>The memoized version</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S = (a \ \text{then} \ s \ \text{then} \ s) )</td>
<td>( \text{ms} = \text{memoize} \ &quot;ms&quot; \ ((\text{ms} \ \text{m_then} \ \text{ms} \ \text{m_then} \ \text{ms}) )</td>
</tr>
<tr>
<td>( \text{m_orelse} \ \text{empty} )</td>
<td>( \text{ms} \ \text{m_orelse} \ \text{empty} )</td>
</tr>
<tr>
<td>( a = \text{term} \ 'a' )</td>
<td>( \text{ma} = \text{mterm} \ 'a' )</td>
</tr>
</tbody>
</table>

Figure 26  The relationship between a recognizer and its memoized version
4.3 The Algorithm

We begin our description of the algorithm by presenting an example. Suppose that the string "aa" is to be processed using the memoized recognizer \( ms \) defined in Figure 26. The initial input is as follows, where the second component of the tuple is the initial memo-table:

\[
([1], [(1, 'a', []), (2, 'a', []), (3, '#', [])])
\]

Owing to the fact that no results have been computed yet and that \( ms \) is an alternation (\$m\_orelse) of two recognizers, the first alternative of \( ms \) is applied to the initial input. This recognizer is itself a sequence of the recognizers \( ma \) and \( ms \) \$then \( ms \), therefore the first of this sequence, i.e. \( ma \), is applied to the initial input. The recognizer \( ma \) succeeds in recognizing an 'a' and returns a result consisting of a pair with first element [2], indicating that the first element of the sequence of tokens has been consumed, and the memo-table unchanged (because basic recognizers do not update the memo-table). The evaluation tree at this point is as follows, where '?' indicates values yet to be computed. Sequencing is denoted by continuous lines and alternation by broken lines.

\[
\begin{align*}
ms & \rightarrow [?] \\
ma & \rightarrow [2] \\
ms & \rightarrow [?] \\
ms & \rightarrow [?] \\
mempty & \rightarrow [?] \\
\end{align*}
\]
Next, ms $\text{then}$ ms is applied to this result. The application of the first ms in this sequence results in a similar computation to the initial application except that the starting position is [2]. The same holds when ms is applied to position [3]. The third element of the input memo-table corresponds to the end-of-input. The recognizer ma applied at position [3] fails, returning an empty list, and thus ma ms ms fails. The recognizer mempty applied at the same position returns as result a tuple whose first element is the list [3]. Now the results of both alternatives of the recognizer ms have been determined and the value of ms applied at position [3] is computed. The following shows the evaluation tree when all values up to ms [2] have been computed:
The memo-table is now:

\[(1, 'a', []), (2, 'a', ["ms", [2, 3]]), (3, '#', ["ms", [3]])\]

Note that when the recognizer \texttt{ms} is applied to the position [3] for the second time, the corresponding result is simply copied from the memo-table.

When a recognizer is applied to a list that contains more than one element, the result is obtained by applying the recognizer to each element in the list and merging the results. This is illustrated below:

\[
\text{ms} \ [2, 3] \Rightarrow [2, 3]
\]

\[
\text{ms} \ [2] \Rightarrow [2, 3] \ (\text{memo})
\]

\[
\text{ms} \ [3] \Rightarrow [3] \ (\text{memo})
\]

The final result is:

\[
([1, 2, 3], [(1, 'a', ["ms", [1, 2, 3]]), (2, 'a', ["ms", [2, 3]]), (3, '#', ["ms", [3]])])
\]

The following is a more formal description of the algorithm:

1. Input:
   a. A context-free, non-left-recursive, grammar with productions and terminals represented using functions mterm, $m\_then$, $m\_orelse$, 

91
and mempty. The start symbol for the grammar is the name of the first recognizer to be applied.

b. A pair whose first component is the list [1], and whose second component is a memo-table corresponding to the input sequence of tokens.

2. Output:

a. A pair whose first component is a list of positions where the recognition process of the input sequence of tokens (starting from the first token) was successfully completed. The second component is the final state of the memo-table.

3. Method:

a. At each step we apply a recognizer to a list of start positions and a memo-table:

- If the list is empty, the result is an empty list and the unchanged memo-table.
- Otherwise, we first apply the recognizer to the first element of the list and the memo-table. The result is a list r1 and a possibly modified memo-table t1. Then we apply the recognizer to the rest of the list and the memo-table t1. The result of this application is a list r2 and a memo-table t2. The final result is a pair: a list obtained by merging r1 and r2, and the table t2.
Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors

b. Application of a recognizer $m$ at a position $j$ begins by reference to the current memo-table:

- If the $j$th row of the memo-table contains a result corresponding to $m$, this result is returned.
- Otherwise a new result is computed, the memo-table is updated and the result returned.

c. Each recognizer can be either the basic recognizer $\text{mempty}$, a basic recognizer constructed using $\text{rterm}$, or it can be a combination constructed from two or more components using $\text{m_then}$ or $\text{m_orelse}$.

- Results for basic recognizers are obtained immediately by applying the corresponding function.
- For sequences or alternations, the results of the components are computed first and then combined to obtain the final result.

5 Complexity analysis

We now show that memoized recognizers have worst-case time complexity of $O(n^3)$ compared to exponential behavior of the unmemoized form. The analysis is concerned only with the variation of time with the length of the input list of tokens. Although a grammar could be very complex, its size will always be independent of the length of the input.
5.1 Elementary operations

We assume that the following operations require a constant amount of time:

1. Testing if two values are equal, less than, etc.

2. Extracting the value of a component of a tuple.

3. Adding an element to the front of a list.

4. Obtaining the value of the ith element of a list whose length depends upon the size of the grammar but not on the size of the input list.

5.2 The size of the memo-table

The memo-table is structured as a list of \((n + 1)\) tuples, where \(n\) is the length of the input sequence of tokens. The first component of each tuple is an integer ranging from 1 to \(n + 1\). The second component of a tuple whose first component is \(i\), is the \(i\)th token in the input. The third component is a list of pairs \((\text{recognizer\_name}, \text{result})\). Owing to the fact that the grammar is fixed, the number of recognizers, denoted by \(r\), is constant. Therefore, for each tuple in the memo-table, the length of the list of pairs is \(\leq r\).

The second component of each pair is a list of positions represented by integers where the corresponding recognizer succeeded in completing the recognition of a segment of the input. The length of the lists that correspond to the \(i\)th tuple is at most \((n - i + 2)\) owing to the fact that a recognizer applied to input at position \(i\) may succeed at any position \(j\), \(i \leq j \leq n + 1\).
5.3 Memo-table lookup and update

The function `lookup` applied to an index i, a recognizer name, and a memo-table, first searches the memo-table to access the ith element, then it searches the list of results in the ith tuple to access the element that corresponds to the given recognizer name. The function `lookup` requires $O(n)$ time.

The function `update` applied to an index i, a result res, and a memo-table, returns a new memo-table with the ith tuple updated. The result res is added in front of the list of successful recognitions corresponding to the ith token. The function `update` requires $O(n)$ time.

5.4 Basic recognizers

Application of the recognizer `mempty` simply creates a pointer to the input. This takes constant time. Application of a recognizer `mterm a` to a single start position i, requires the ith entry in the memo-table to be examined to see if the ith token is equal to a. If there is a match then the result $i + 1$ is added to the list of results returned by `mterm`. Otherwise the recognizer fails. This operation is $O(n)$.

Note that we are only considering, here and in the next two subsections, the time required to apply a recognizer to a single position in the input list. We consider application of a recognizer to a more-than-one-element list later.
5.5 Alternation

Assuming that the results \( p \) \( i \) and \( q \) \( i \) have been computed, application of a memoized recognizer \( (p \; \text{m}_\text{orelse} \; q) \) to a single start position \([i]\), involves the following steps:

- one memo-table lookup \(-\); \( \mathcal{O}(n) \)
- and, if the recognizer has not been applied before:
  - merging of two result lists, each of which is in the worst case of length \( n + 1 \), \( - \mathcal{O}(n) \),
  - one memo table update \(- \mathcal{O}(n) \).

5.6 Sequencing

Assume that \( p \; [i] \) has already been calculated. In the worst case the result is the list \([i, \; i + 1, \ldots, \; n + 1]\). Assume also that \( q \; [i, \; i + 1, \ldots, \; n + 1] \) has already been calculated. Now, application of a memoized recognizer \( p \; \text{m}_\text{then} \; q \) to a single start position \( i \) involves the following steps:

- one memo-table lookup \((\mathcal{O}(n))\) and if the lookup fails,
  - computation of the result plus
  - one memo-table update \((\mathcal{O}(n))\).

5.7 Merging results returned when a recognizer is applied to a list of start positions

The function \( \text{merge} \) is also used to combine the results returned by a single memoized recognizer when applied to a list of start positions with
more than one entry. (See the definition of memoize). Suppose a recognizer \( f \) is applied to a \( k \)-element list of start positions \([1, \ldots, k]\). The corresponding evaluation tree is as follows:

\[
\begin{array}{c}
\text{\( f [1, 2, \ldots, k] \)} \\
\downarrow \\
\text{\( f [1] \)} & \rightarrow & \text{\( f [2, 3, \ldots, k] \)} \\
\downarrow & & \downarrow \\
\text{\( f [2] \)} & & \text{\( f [3, 4, \ldots, k] \)} \\
\downarrow & & \downarrow \\
\text{\( f [k] \)} & & \text{\( f [\cdot] \)} \\
\end{array}
\]

Assuming that the results of \( f [i] \) and \( f [i + 1, \ldots, k] \) have already been computed, computation of \( f [i, i + 1, \ldots, k] \) requires one memo-table lookup (\( O(n) \)) and one merge, of two lists which are in the worst case of length \( n + 1 \). The total time is \( O(n) \). Note also that application of a recognizer \( f \) to a \( k \)-element list of start positions, results in an execution tree with \( 2^k + 1 \) nodes representing applications of the recognizer \( f \).

5.8 The execution tree

The analysis so far can be summarized in terms of execution trees (such as those shown earlier). Each non-leaf node of an execution tree corresponds to an application of a recognizer to a list of start positions, or to an application of \( \text{m_orelse} \) or \( \text{m_then} \). Leaf-nodes correspond either to an
application of \texttt{mempty}, or \texttt{mterm a} for some \texttt{a}, or to a computation that has been performed before and stored in the memo-table:

Lemma 1: We have shown that the result corresponding to \texttt{mempty}, \texttt{mterm a} for some \texttt{a}, and \texttt{lookup} can be computed in \texttt{O(n)} time.

Lemma 2: We have also shown that results corresponding to non-leaf nodes can be computed in \texttt{O(n)} time provided that the values of their children are available.

5.9 Proof of \texttt{O(n^3)} time complexity

Theorem

Given an arbitrary context-free non-left-recursive grammar \texttt{G}, the corresponding memoized functional recognizer requires \texttt{O(n^3)} time to process an input sequence of length \texttt{n}. If the grammar is not ambiguous, the time complexity is \texttt{O(n^2)}.

Proof

Let \texttt{f_1, f_2, \ldots, f_l} be a set of recognizers corresponding to the grammar \texttt{G}, and let \texttt{f_1} correspond to the start symbol in the grammar. We begin by applying the recognizer \texttt{f_1} to the list \texttt{[1]}. This application yields an execution tree similar to the ones shown earlier. We will show that for an arbitrary grammar the number of nodes in such a tree is \texttt{O(n^2)}, and if the grammar is not ambiguous this number reduces to \texttt{O(n)}. Owing to the fact that the time required to perform computations at each node is linear in the length of the
input sequence (Lemma 1 and Lemma 2), this concludes the proof of the theorem.

For simplicity assume that each recognizer is either mempty, mterm a for some a, or is of the form \((p \ $m\_orelse \ q)\), or \((p \ $m\_then \ q)\) for some \(p\) and \(q\). In practice recognizers can be a combination \((\$m\_orelse\ and/or\ \$m\_then)\) of more than two recognizers, but the number will always be bounded by the size of the grammar and will be independent of the length of the input sequence of tokens.

Suppose that the recognizer \(f_i\) is of the form \((f_j \ $m\_orelse \ f_j)\) for some \(2 \leq i, j \leq r\). Suppose also that the recognizer \(f_i\) is of the form \((f_k \ $m\_then \ f_p)\) for some \(2 \leq k, p \leq r\) and \(k \neq i, p \neq i\). The corresponding tree in the worst case is as follows:

```
        f_i[1]
       /   \
  f_i[1]   f_j[1]
 /        \
f_k[1]    f_p[1, 2, \ldots, n + 1]
```

Consider the expansion of those subtrees that correspond to an application of a recognizer to the one-element list of start positions \([1]\). Owing to the fact that the grammar is non-left-recursive and that it consists of \(r\) recognizers, after a maximum number of steps in each path, which depends only on the size of the grammar, there must be an application of a recognizer that consumes some input. It follows that the total number of
applications of a recognizer to a one-element list is independent of the length of the input. For the same reason, the total number of applications of a recognizer to a more than one-element list (in the worst case an \((n + 1)\) element list) is independent of the length of the input.

When the first step is completed, there will be only \(O(r)\) subtrees to be further expanded. This is because the result corresponding to a pair (recognizer, start position) is calculated only once. If the same recognizer is applied to the same start position again, the corresponding result is simply copied from the memo-table. At the next stage, the same procedure is repeated for each recognizer that is applied to the list \([2]\). The only difference is that now \(O(r)\) sub-trees must be expanded not just one. Only \(O(r)\) nodes will be generated for each recognizer applied to the list \([2]\), and \(O(r)\) nodes for each application of a recognizer to a more-than-one-element list.

At the \(i\)th step, there will be \(O(r)\) nodes corresponding to an application of a recognizer to the list \([i]\), and \(O(r)\) nodes corresponding to an application of a recognizer to a more-than-one-element list (in the worst case, an \((n - i + 2)\) element list). The total number of steps is \(n + 1\). Owing to the fact that an application of a recognizer to a \(k\)-element list yields a tree that contains \(2 \times k + 1\) nodes, as discussed in sub-section 5.7, the total number of nodes is given by the following, where \(c\) is proportional to the number of recognizers \(r\).
\[ N = \sum_{k=1}^{n+1} (c + c \cdot (2 \cdot k + 1)) = 2 \cdot c \cdot (n + 1) + 2 \cdot c \cdot (n + 1) \cdot (n + 2) / 2 = O(n^2) \]

If the grammar is not ambiguous then each input sequence of tokens can be recognized in just one way. Therefore, each recognizer applied at some position \( i \) will return at most a one-element-list as result. The corresponding formula for unambiguous grammars given below concludes the proof.

\[ N = \sum_{k=1}^{n+1} (c + c \cdot 1) = 2 \cdot c \cdot (n + 1) = O(n) \]

6 A monadic approach to incorporate memoization

So far, we have used an ad hoc method to redefine the recognizer functions in order to incorporate memoization. This method is susceptible to error. In fact, an earlier version of the paper contained an insidious error: the function \( \text{m\_then} \) was defined as follows:

\[
(p \ \text{m\_then} \ q) \ (bs, t) = q \ (rp, tp), \text{ if } rp = []
\]
\[
\text{where } (rp, tp) = p \ (bs, t)
\]

According to this definition, if the recognizer \( p \) fails, then the memo table is returned unchanged in the result \( ([], t) \). This error would result in exponential complexity for certain grammars when applied to certain inputs which fail to be recognized. In this section, we show how the recognizer definitions can be modified in a structured way which reduces the possibility
of such errors. The method treats memoization as a specific instance of the more general notion of adding features to purely functional programs.

6.1 Monads

Monads were introduced to computing science by Moggi [15] who noticed that reasoning about programs that involve handling of the state, exceptions, I/O, or non-determinism can be simplified, if these features are expressed using monads. Inspired by Moggi’s ideas, Wadler [21] proposed monads as a way of structuring functional programs. The main idea behind monads is to distinguish between the type of values and the type of computations that deliver these values. A monad is a triple \((M, \text{unit}, \text{bind})\) where \(M\) is a type constructor, and \(\text{unit}\) and \(\text{bind}\) are two polymorphic functions. \(M\) can be thought of as a function on types, that maps the type of values into the type of computations producing these values. \(\text{unit}\) is a function that takes a value and returns a corresponding computation; the type of \(\text{unit}\) is \(a \rightarrow M\ a\). The function \(\text{bind}\) represents sequencing of two computations where the value returned by the first computation is made available to the second [and possibly subsequent] computation. The type of \(\text{bind}\) is

\[
M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b.
\]

The identity monad [21] below represents computations as the values they deliver.
id * = id *
unit1 :: * -> id *
unit1 x = x
bind1 :: id * -> (* -> id **) -> id **
(p $bind1 k) = k p

The state monad (also defined in [21]) is an abstraction over computations that deal with the state. The definition is given below.

stm * == state -> (*, state)
state == [([num, [[char], [num]]])
unit2 :: * -> stm *
unit2 a = f
    where f t = (a, t)
bind2 :: stm * -> (* -> stm **) -> stm **
(m $bind2 k) = f
    where
    f x = (b, z)
        where
        (b, z) = k a y
            where
            (a, y) = m x

We will use the identity and the state monad to construct non-memoized and memoized monadic recognizers respectively. In the description below, we refer to a third monad which we use as an analogy in the construction of our monadic recognizers. This is the monad for lists [21]. Owing to the fact that our recognizers can be applied to a list of inputs, it is necessary to have a well structured way of doing that.

list * == [*]
unit :: * -> list *
unit a = [a]
bind :: list * -> (* -> list **) -> list **
[1] $bind y = []
(a:x) $bind y = (y a) ++ (x $bind y)

6.2 Non-memoized monadic recognizers

In order to use monads to provide a structured method for adding new effects to a functional program, we begin by identifying all functions that will be involved in those effects. We then replace those functions, which can be of any type a -> b, by functions of type a -> M b. In effect, we change the program so that selected function applications return a computation on a value rather than the value itself. This computation may be used to add features such as state to the program. In order to effect this change, we use the function unit to convert values into computations that return the value but do not contribute to the new effects, and the function bind is used to apply a function of type a -> M b to a computation of type M a. Having made these changes, the original program can be obtained by using the identity monad id_M, as shown below. In order to add new effects such as state, or exceptions, we simply change the monad and make minimal local changes as required to the rest of the program. In the following sub-section, we show how to add the new effect of memoization by replacing the identity monad with the state monad st_M, and making some local changes.

The non-memoized recognizers introduced earlier in this paper were functions taking a list of input sequences of tokens and returning a similar list of sequences yet to be processed. The definition of the non-memoized monadic recognizers differs slightly in that the list of inputs is represented by
a pair: a list of start positions and the whole input sequence of tokens. Owing to the fact that the input sequence remains unchanged during the execution of the program, there is no need for any recognizer to return it.

In order to construct non-memoized monadic recognizers, we start by defining the type of non-memoized recognizers. We define it using the type constructor idm of the identity monad.

\[
\text{rec } * = [\text{char}] \rightarrow [\text{num}] \rightarrow \text{idm } *
\]

That is, a recognizer of type \( \mathbb{A} \) is a function that applied to an input string and a list of start positions returns an "identity" computation of type \( \mathbb{A} \). We can now define the function \( \text{term}_1 \), which when applied to a character, returns a function that is always applied to a one-element list of positions.

\[
\text{term}_1 :: \text{char} \rightarrow \text{rec } [\text{num}]
\]
\[
\text{term}_1\ c\ s\ [x] = \text{unitl } [], \text{ if } [x > \#s] \lor (s!(x-1) == c)
\]
\[
= \text{unitl } [x+1], \text{ otherwise}
\]

In analogy with the bind operator for the list monad, the function \( \text{term} \), which when applied to a character returns a recognizer that can be applied to more than one-element list of start positions, is defined as follows.

\[
\text{term} :: \text{char} \rightarrow \text{rec } [\text{num}]
\]
\[
\text{term}\ c\ s\ [] = \text{unitl } []
\]
\[
\text{term}\ c\ s\ [x:xs] = \text{term}_1\ c\ s\ [x] \$\text{bindl } f
\]
\[
\text{where } f\ a = \text{term}\ c\ s\ xs \$\text{bindl } g
\]
\[
\text{where } g\ b = \text{unitl } (\text{merge\_res } a \ b)
\]

105
The definitions of orelse, then, and empty are given below. Note that we have replaced the append operator ++ with the function merge_res that combines two lists removing duplicates.

orelse :: rec [num] -> rec [num] -> rec [num]
(p $orelse q) s input = p s input $bind1 f
  where
  f a = q s input $bind1 g
  where
  g b = unit1 (merge_res a b)

then :: rec [num] -> rec [num] -> rec [num]
(p $then q) s input = p s input $bind1 f
  where f a = unit1 [], if a = []
            = q s a , if a ~= []

empty :: rec [num]
empty s x = unit1 x

Notice that we have not rewritten the application of merge_res using bind1. The reason for this is that we know that in this application, merge_res will not be involved in memoization and therefore the result of its application can be viewed as a value rather than a computation.

6.3 Memoized monadic recognizers

We now consider the state monad as given earlier and define the two operations on the state: lookup and update. The type of the state is [num, [(char), [num]]].

lookup :: num -> [char] -> stm [[num]]
lookup ind name st
  = ([] , st) , if ind > #st
  = ([bs | (x,bs) <-(snd (st!(ind-1)));x=name], st) , otherwise
update :: num -> [char] -> [num] -> stm [ ]
update ind name val st
    = (undef, map (update_mt_entry ind name val) st)

update_mt_entry ind name val (x: list)
= (x, (name, val) : list), if x = ind
= (x, list), otherwise

We define the type of the memoized recognizers in terms of the type constructor of the state monad stm:

rec * == [char] -> [num] -> stm *

and define the function memoize (there is an analogy to the definition of term and bind for the list monad here).

memoize :: [char] -> mrec [num] -> mrec [num]
memoize name f s [] = unit2 []
memoize name f s (x: xs) = memoize name f s [x] $bind2 g
    where
        g a = memoize name f s xs $bind2 h
        where
            h b = unit2 (merge_res a b)

memoize1 :: [char] -> mrec [num] -> mrec [num]
memoize1 name f s [i] = lookup i name $bind2 g
    where
        g a = unit2 (a!0), if a == []
        = f s [i] $bind2 h, otherwise
        where
            h b = update i name b $bind2 r
            where
                r any = unit2 b

The definitions of term, orelse, and then remain unchanged except that unit1 and bind1 are replaced by unit2 and bind2 respectively. The memo-table is completely hidden in the definition of term, orelse, and then. One of the advantages is that having identified all recognizer functions as
being involved in the memoization effect, the monadic form of `orElse` is straightforward and thereby this approach reduces the chance of making the kind of error referred to at the beginning of this section. The definition of monadic memoized recognizers is exactly the same as with the original memoized recognizers, and the complexity analysis presented earlier holds also for memoized monadic recognizers.

The following table shows the results obtained when an unmemoized monadic recognizer for the grammar `s ::= a s s | empty` and a memoized monadic version were applied to inputs of various length. The results suggests that the recognizers also have $O(n^3)$ space complexity as well as $O(n^3)$ time complexity.

<table>
<thead>
<tr>
<th># of 'a's</th>
<th>Number of reductions</th>
<th>Space in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unmemoized</td>
<td>memoized</td>
</tr>
<tr>
<td>3</td>
<td>2,132</td>
<td>4,490</td>
</tr>
<tr>
<td>6</td>
<td>24,914</td>
<td>11,867</td>
</tr>
<tr>
<td>9</td>
<td>222,792</td>
<td>24,396</td>
</tr>
<tr>
<td>12</td>
<td>1,830,567</td>
<td>43,638</td>
</tr>
<tr>
<td>15</td>
<td>14,726,637</td>
<td>70,935</td>
</tr>
<tr>
<td>18</td>
<td>117,938,202</td>
<td>107,745</td>
</tr>
<tr>
<td>21</td>
<td>out of space</td>
<td>155,526</td>
</tr>
</tbody>
</table>
More information on the use of monads to structure functional language processors can be found in Wadler [20, 21, 22].

7 Memoizing parsers and syntax-directed evaluators

The memoization technique presented in this paper could be readily extended so that the memo-tables contain parse tables similar to those created by Earley’s algorithm [3], or the more compact representation of factored syntax trees suggested by Leiss [13]. However, to do so would not be in keeping with an approach that is commonly used by the purely-functional programming community in building language processors. That approach is to avoid the explicit construction of syntax trees unless the trees are specifically required to be displayed as part of the output. Instead of constructing labeled syntax trees which are subsequently evaluated, an alternative approach is used: semantic actions are closely associated with the executable grammar productions so that semantic attributes are computed directly without the need for the explicit representation of syntax trees. User-defined types can be introduced to accommodate different types of attributes as has been done in the W/AGE attribute grammar programming language [7]. This approach is viable owing to the lazy-evaluation strategy employed by most purely-functional languages. The memoization technique described above can be used to improve the efficiency of such syntax-directed evaluators with two minor modifications:

1. The definition of m_orelse is changed so that the function merge is replaced with a function that removes results that are regarded as
duplicates under application-dependent criteria which may be less inclusive than the criterion used for recognizers. Results that are returned by a recognizer are regarded as duplicates if they have the same end points. For recognition purposes the end points are all that is required to be maintained in the memo-table. With syntax-directed evaluators, the end points may be augmented with semantic values. A single end point pair may have more than one value associated with it. In some cases syntactic ambiguity may result in semantic ambiguity. Results returned by a language processor would only be regarded as being duplicates if they have the same end points and have equivalent semantic attributes. The function merge would be replaced by an application-dependent function that identifies and removes such duplication. In this approach, if syntax trees are required as part of the output, they are simply treated as another attribute. In such cases syntactic ambiguity is isomorphic with semantic ambiguity and the function merge would be replaced by concatenation in the definition of m_orelse.

2. The memo-tables and the update and lookup functions are modified according to the attributes that are required in the application.

One advantage that derives from this approach is that all unnecessary computation is avoided. Memoization prevents language processors from
reprocessing segments of the input already visited and the use of merge, or an application-dependent version of it, removes duplication in subcomponents of the result as soon as it is possible to detect it. It should be noted that the complexity of language processors constructed in this way is application dependent. If syntax trees are required to be represented in full, the language processor may have exponential complexity in the worst case owing to the fact that the number of syntax trees can be exponential in the length of the input for highly ambiguous grammars. A compact representation of the trees could be produced in polynomial time and the trees could then be passed on to an evaluator. However, this would detract from the modularity of the language processor and would provide no benefit if the trees were to be subsequently displayed or otherwise processed separately as this could be an exponential process.

8 Concluding comments

This paper was inspired by Norvig's demonstration that memoization can be implemented at the source-code level in languages such as Common Lisp to improve efficiency of simple language processors without compromising their simplicity. We have shown that Norvig's technique can be adapted for use in purely-functional programming languages that do not admit any form of updateable object. The technique described in this paper can be thought of as complementing that of Norvig's in that it enables memoization to be used to improve the efficiency of highly-modular language processors constructed in purely-functional languages.
This application has also illustrated how monads can be used to structure functional programs in order to avoid errors when modifications such as the addition of state are made. We are now exploring the use of monads in the memoization of programs that are constructed as executable attribute grammars.

9 Acknowledgments

We would like to thank Young Gil Park, Dimitris Phoukas, and Yung H. Tsin of the University of Windsor for helpful discussions, and to the anonymous referees for careful reading of the paper, for many useful suggestions, and for identifying the error in the code that was discussed in section 6. Richard Frost also acknowledges the support received from the Canadian Natural Sciences and Engineering Research Council.

10 References


Appendix A: Memoizing Purely-Functional Top-Down Backtracking Language Processors


Appendix B

IMPLEMENTATION OF MONADIC LANGUAGE PROCESSORS USING TYPE CONSTRUCTOR CLASSES IN GOFER

---

-- CLASSES
--

class Monad m where
    unit :: a -> m a
    bind :: m a -> (a -> m b) -> m b

class Monad m => MonadPlusZero m where
    plus :: m a -> m a -> m a
    zero :: m a

class MonadPlusZero m => Recognizer m a where
    termR :: Char -> a -> m a
    emptyR :: a -> m a
    orelseR :: (a -> m a) -> (a -> m a) -> (a -> m a)
    thenR :: (a -> m a) -> (a -> m a) -> (a -> m a)
    failR :: m a

    emptyR = unit
    (p `thenR` q) inp = p inp `bind` \x1 ->
        (q x1 `bind` \x2 ->
            unit x2
        )
    (p `orelseR` q) inp = (p inp) `plus` (q inp)
    failR = zero

class MonadPlusZero m => Parser m a where
    term :: Char -> m a
    empty :: a -> m a
    orelse :: m a -> m a -> m a
    fail :: m a

    empty = unit
    orelse = plus
    fail = zero

115
class Recognizer m a => MRecognizer m a where
  memoizeR :: String -> (a -> m a) -> a -> m a

class Parser m a => MParser m a where
  memoize :: String -> m a -> m a

-- BASIC MONADS

-- LIST MONAD

type List a = [a]
  in unitLs, bindLs, termEInt

unitLs :: a -> [a]
unitLs a = [a]

bindLs :: [a] -> (a -> [b]) -> [b]
x "bindLs" k = case x of
  [] -> []
  (a:x) -> (k a) "merge_res" (x "bindLs" k)

instance Monad [] where
  unit = unitLs
  bind = bindLs

instance MonadPlusZero [] where
  zero = []
  plus = merge_res

merge_res :: [a] -> [a] -> [a]
merge_res x [] = x
merge_res [] y = y
merge_res (x:xs) (y:ys)
  | x <= y = x:merge_res xs (y:ys)
  | y <= x = y:merge_res (x:xs) ys
  | otherwise = x:merge_res xs ys

-- To avoid problems with the class Ord defined in the
-- standard prelude

primitive (<<) "primGenericLt" :: a -> a -> Bool

-- STATE MONAD

type St s a = s -> (a, s)
in unitSt, bindSt, lookupSt, updateSt,
  newSt, parseSt, memoizeEv, parseStEv
unitSt :: a -> St s a
unitSt a = \t -> (a, t)

bindSt :: St s a -> (a -> St s b) -> St s b
(a `bindSt` k) t = k va ta
    where
        (va, ta) = a t

instance Monad (St s) where
    unit = unitSt
    bind = bindSt

type State v = [(Int, [(String, v)])]
in unitSt, bindSt, lookupSt, updateSt,
    newSt, parseSt, memoizeEv, parseStEv

lookupSt :: Int -> String -> St (State v) [v]
lookupSt ind name t |
    | ind > length t = ([], t)
    | otherwise = ([bs| (x,bs)<- (snd(t!!(ind-1)))],x=name),t)

updateSt :: Int -> String -> v -> St (State v) ()
updateSt ind name val st
    = (([], map (update_mt_entry ind name val) st)
    where
        update_mt_entry ind name val (x, list)
            | x == ind = (x, (name, val) : list)
            | otherwise = (x, list)

ewSt :: Int -> State v
newSt n = (zip [1..(n + 1)] (repeat []))

-- EXCEPTION MONAD

data Ex a = Fail | Ok a

unitEx :: a -> Ex a
unitEx a = Ok a

bindEx :: Ex a -> (a -> Ex b) -> Ex b
(Ok a) `bindEx` k = k a
Fail `bindEx` k = Fail

zeroEx :: Ex a
zeroEx = Fail
Appendix B: Implementation of Monadic Language Processors Using Type Constructor Classes

plusEx :: Ex a -> Ex a -> Ex a
Fail `plusEx` x = x
x `plusEx` _ = x

instance Monad Ex where
  unit = unitEx
  bind = bindEx

instance MonadPlusZero Ex where
  plus = plusEx
  zero = zeroEx

-- PARAMETRIZED MONADS
-- PARAMETRIZED STATE READER MONAD
type Strm m s a = s -> m a
  in unitStrm, liftStrm, bindStrm, zeroStrm, plusStrm,
  termInt, parseInt, termEIInt, parseEIInt,
  memoizeRec, parseSt, memoizeEv, parseStEv

liftStrm :: Monad m => m a -> Strm m s a
liftStrm x t = x

unitStrm :: Monad m => a -> Strm m s a
unitStrm x = liftStrm (unit x)

bindStrm :: Monad m =>
  Strm m s a -> (a -> Strm m s b) -> Strm m s b
(a `bindStrm` k) t = a t `bind` \va ->
  k va t

zeroStrm :: MonadPlusZero m => Strm m s a
zeroStrm = \s -> zero

plusStrm :: MonadPlusZero m
  => Strm m s a -> Strm m s a -> Strm m s a
(x `plusStrm` y) s = x s `plus` y s

instance Monad m => Monad (Strm m s) where
  unit = unitStrm
  bind = bindStrm

instance MonadPlusZero m
  => MonadPlusZero (Strm m s) where
  zero = zeroStrm
  plus = plusStrm
-- PARAMETERIZED LIST MONAD

\[
\begin{align*}
type\ ListM\ m a & = m [a] \\
& \quad \text{in unitLsM, bindLsM, plusLsM, zeroLsM, liftLsM,}
& \quad \text{memoizeRec, parseSt, memoizeEv, parseStEv}
\end{align*}
\]

unitLsM :: Monad m => a -> ListM m a
unitLsM x = liftLsM (unit x)

bindLsM :: Monad m
\quad => ListM m a -> (a -> ListM m b) -> ListM m b
\quad x `bindLsM` k = x `bind` \(\lambda x1 \rightarrow\)
\quad \quad foldr plusLsM zeroLsM (map k x1)

liftLsM :: Monad m => m a -> ListM m a
liftLsM x = x `bind` \(\lambda x1 \rightarrow\)
\quad unit [x1]

instance Monad m => Monad (ListM m) where
\quad unit = unitLsM
\quad bind = bindLsM

instance Monad m => MonadPlusZero (ListM m) where
\quad zero = zeroLsM
\quad plus = plusLsM

zeroLsM :: Monad m => ListM m a
zeroLsM = unit []

plusLsM :: Monad m =>
\quad ListM m a -> ListM m a -> ListM m a
\quad (x `plusLsM` y) = x `bind` \(\lambda x1 \rightarrow\)
\quad y `bind` \(\lambda x2 \rightarrow\)
\quad unit (x1 `merge_res` x2)

-- PARAMETERIZED INPUT MONAD

type InpM m s a = s -> m (a, s)
in unitInpM, bindInpM, liftInpM, zeroInpM, plusInpM,
termInInt, parseInInt, memoizeEv, parseStEv

unitInpM :: Monad m => a -> InpM m s a
unitInpM a = liftInpM (unit a)

bindInpM :: Monad m
\quad => InpM m s a -> (a -> InpM m s b) -> InpM m s b
\quad (a `bindInpM` k) inp = a inp `bind` \(\lambda (va, outa) \rightarrow\)
\quad \quad k va outa
Appendix B: Implementation of Monadic Language Processors Using Type Constructor Classes

\[
\text{liftInpM :: Monad } m \Rightarrow m a \rightarrow \text{InpM } m s a
\]
\[
\text{liftInpM } x \text{ inp } = x \text{ bind } \text{xln } \rightarrow
\]
\[
\text{unit (xln, inp)}
\]

\[
\text{zeroInpM :: MonadPlusZero } m \Rightarrow \text{InpM } m s a
\]
\[
\text{zeroInpM } = \text{\inp } \rightarrow \text{zero}
\]

\[
\text{plusInpM :: MonadPlusZero } m
\]
\[
\Rightarrow \text{InpM } m s a \rightarrow \text{InpM } m s a \rightarrow \text{InpM } m s a
\]
\[
(x \text{ plusInpM } y) = \text{\inp } \rightarrow x \text{ inp } \text{plus } y \text{ inp}
\]

\[
\text{instance Monad } m \Rightarrow \text{Monad (InpM } m s) \text{ where}
\]
\[
\text{unit } = \text{unitInpM}
\]
\[
\text{bind } = \text{bindInpM}
\]

\[
\text{instance MonadPlusZero } m
\]
\[
\Rightarrow \text{MonadPlusZero (InpM } m s) \text{ where}
\]
\[
\text{zero } = \text{zeroInpM}
\]
\[
\text{plus } = \text{plusInpM}
\]

---

--- RECOGNIZERS

---

-- Recognizers that are applied to a position and the whole list of tokens

\[
\text{termInt :: MonadPlusZero } m
\]
\[
\Rightarrow \text{Char } \rightarrow \text{Int } \rightarrow \text{StRM } m \text{ String } \text{Int}
\]
\[
\text{termInt } c \text{ x s}
\]
\[
\mid (x<1) \mid | (x>\text{length s}) \mid | (s!!(x-1) /= c) = \text{zero}
\]
\[
\mid \text{otherwise} = \text{unit (x+1)}
\]

\[
\text{parseInt :: (Int } \rightarrow \text{StRM } m \text{ String } \text{Int}) \rightarrow \text{Int } \rightarrow \text{String}
\]
\[
\rightarrow m \text{ Int}
\]
\[
\text{parseInt } x \text{ inp } s = x \text{ inp } s
\]

\[
\text{instance MonadPlusZero (StRM } m \text{ String)}
\]
\[
\Rightarrow \text{Recognizer (StRM } m \text{ String) Int where}
\]
\[
\text{termR } = \text{termInt}
\]

\[
\text{memoizeRec ::}
\]
\[
\text{Recognizer (StRM (ListM (St (State [Int])))) String) Int}
\]
\[
\Rightarrow \text{String}
\]
\[
\rightarrow (\text{Int } \rightarrow \text{StRM (ListM (St (State [Int])))} \text{ String Int})
\]
\[
\rightarrow (\text{Int } \rightarrow \text{StRM (ListM (St (State [Int])))} \text{ String Int})
\]

120
memoizeRec name f i s = lookupSt i name `bindSt` \x1 ->
    if x1 /= []
    then unitSt (x1!!0)
    else f i s `bindSt` \x2 ->
        updateSt i name x2 `bindSt` () ->
        unitSt x2

parseSt ::
    (Int -> StRM (ListM (St (State [Int]))) String Int)
    -> Int -> String -> ([Int], State [Int])
parseSt x inp s = x inp s (newSt (length s))

instance Recognizer
    (StRM (ListM (St (State [Int]))) String Int)
    => MRecognizer
    (StRM (ListM (St (State [Int]))) String Int)
    where
        memoizeR = memoizeRec

-- EXAMPLES

-- non-deterministic recognizer
r1 :: Recognizer (StRM [] String Int)
    -> Int -> StRM [] String Int
r1 = (termR 'a' `thenR` (r1 `thenR` r1))
    `orelseR` emptyR

-- deterministic recognizer
r2 :: Recognizer (StRM Ex String Int)
    -> Int -> StRM Ex String Int
r2 = (termR 'a' `thenR` (r2 `thenR` r2))
    `orelseR` emptyR

-- memoized recognizer
r3 :: Recognizer
    (StRM (ListM (St (State [Int]))) String Int)
    -> Int -> StRM (ListM (St (State [Int]))) String Int
r3 = memoizeR "r3" ((termR 'a' `thenR` (r3 `thenR` r3))
    `orelseR` emptyR)

-- ? parseInt r1 1 "aaaaa"
-- [1, 2, 3, 4, 5, 6]
-- (10448 reductions, 16026 cells)
-- ? parseInt r2 1 "aaaaa"
-- Ok 6
-- (569 reductions, 813 cells)
Appendix B: Implementation of Monadic Language Processors Using Type Constructor Classes

```
-- ? parseSt r1 l "aaaaaa"
-- ([1, 2, 3, 4, 5, 6],
--   (("r3",[1, 2, 3, 4, 5, 6])),
--   (("r3",[2, 3, 4, 5, 6])),
--   (("r3",[3, 4, 5, 6])),
--   (("r3",[4, 5, 6])),
--   (("r3",[5, 6])),
--   ([4, (["r3",[6]]))
-- (4069 reductions, 7867 cells)

-- Recognizers that are applied to a list of tokens and
-- return a list of tokens yet to be recognized

termChar :: MonadPlusZero m
  => Char -> String -> m String

termChar c inp = case inp of
  []       -> zero
  (x:xs)   -> if x == c
            then unit xs
            else zero

parseChar :: (String -> m String) -> String -> m String
parseChar x s = x s

instance MonadPlusZero m => Recognizer m String where
  termR = termChar

-- EXAMPLES
-- non-deterministic recognizer
r4 :: Recognizer [] String -> String -> [String]
r4 = (termR 'a' `thenR` (r4 `thenR` r4))
    `orelseR` emptyR

-- deterministic recognizer
r5 :: Recognizer Ex String -> String -> Ex String
r5 = (termR 'a' `thenR` (r5 `thenR` r5))
    `orelseR` emptyR

-- ? parseChar r4 "aaaaaa"
-- [] ["a", "aa", "aaa", "aaaa", "aaaaa"]
-- (? parseChar r5 "aaaaa"
-- Ok []
-- (143 reductions, 186 cells)
```
Appendix B: Implementation of Monadic Language Processors Using Type Constructor Classes

-- EVALUATORS

-- "term" for an (Int) evaluator

```
termEInt :: MonadPlusZero m
    => Char -> StRM (InpM m Int) String Int

    termEInt c s x
        | (x<1) || (x>length s) = zero
        | otherwise = unit (eval c, x+1)

    eval 'a' = 1
```

```
parseEInt :: StRM (InpM m Int) String Int -> String
    -> Int -> m (Int, Int)
parseEInt x s inp = x s inp
```

```
instance MonadPlusZero (StRM (InpM m Int) String)
    => Parser (StRM (InpM m Int) String) Int where
    term = termEInt
```

```
memoizeEv ::
Parser
    (StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String) Int
    => String
    -> StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String Int
    -> StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String Int
memoizeEv name f s i
    = lookupSt i name "bindSt" \x1 ->
        if x1 /= []
            then unitSt (x1!!0)
            else f s i "bindSt" \x2 ->
                updateSt i name x2 "bindSt" \() ->
                unitSt x2
```

```
parseStEv ::
StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String Int
    -> String -> Int -> ([(Int,Int)], State [(Int,Int)])
parseStEv x s inp = x s inp (newSt (length s))
```

```
instance Parser
    (StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String) Int
    => MP parser
    (StRM (InpM (ListM (St (State [(Int,Int)]))) Int) String) Int
where
    memoize = memoizeEv
```
-- EXAMPLES
-- non-deterministic evaluator
e1 :: Parser (StrM (InpM [] Int) String) Int
  => StrM (InpM [] Int) String Int
  e1 = (term 'a' ~bind~ \x1 ->
    e1 ~bind~ \x2 ->
    e1 ~bind~ \x3 ->
    unit (x1 + x2 + x3))
    `orelse`
    empty 0

-- deterministic evaluator
e2 :: Parser (StrM (InpM Ex Int) String) Int
  => StrM (InpM Ex Int) String Int
  e2 = (term 'a' ~bind~ \x1 ->
    e2 ~bind~ \x2 ->
    e2 ~bind~ \x3 ->
    unit (x1 + x2 + x3))
    `orelse`
    empty 0

-- memoized evaluator
e3 :: MParse
  (StrM (InpM (ListM (St (State [[(Int,Int)]]) Int) String) Int
  =>
  StrM (InpM (ListM (St (State [[(Int,Int)]]) Int) String Int
  e3 = memoize "e3" ((term 'a' ~bind~ \x1 ->
    e3 ~bind~ \x2 ->
    e3 ~bind~ \x3 ->
    unit (x1 + x2 + x3))
    `orelse`
    empty 0)

-- ? parseEInt e1 "aaaaa" 1
-- ([0,1], (1,2), (2,3), (3,4), (4,5), (5,6])
-- (13414 reductions, 22161 cells)
-- ? parseEInt e2 "aaaaa" 1
-- Ok (5,6)
-- (651 reductions, 1006 cells)
-- ? parseStEv e3 "aaaaa" 1
-- ([[(0,1), (1,2), (2,3), (3,4), (4,5), (5,6)]],
-- [[[1,[["e3"*,[[0,1], (1,2), (2,3), (3,4), (4,5), (5,6)]]]]],
-- [[2,[["e3"*,[[0,2], (1,3), (2,4), (3,5), (4,6)]]]]],
-- [[3,[["e3"*,[[0,3], (1,4), (2,5), (3,6)]]]]],
-- [[4,[["e3"*,[[0,4], (1,5), (2,6)]]]]],
-- [[5,[["e3"*,[[0,5], (1,6)]]]]],
-- [[6,[["e3"*,[[0,6]]]]]]
-- (5381 reductions, 10856 cells)
VITA AUCTORIS

Barbara Szydlowski was born on January 2, 1964, in Lublin, Poland. After receiving her high school diploma from the Hetman Jan Zamoyski Gymnasium in 1982, she began her study at the Marie Curie-Sklodowska University in Lublin. She graduated with Master's Degree in Mathematics in 1987. Currently she is a candidate for Master's Degree in Science at the University of Windsor.