Efficient Combinator Parsers

Pieter Koopman[*] and Rinus Plasmeijer

Computer Science
Nijmegen University, The Netherlands
pieter@cs.kun.nl, rinus@cs.kun.nl

Abstract. Parser combinators enable the construction of recursive descent parsers in a very clear and simple way. Unfortunately, the resulting parsers have a polynomial complexity and are far too slow for realistic inputs. We show how the speed of these parsers can be improved by one order of magnitude using continuations. These continuations prevents the creation of intermediate data structures. Furthermore, by using an exclusive or-combinator instead of the ordinary or-combinator the complexity for deterministic parsers can be reduced from polynomial to linear. The combination of both improvements turn parser combinators from a beautiful toy to a practically applicable tool which can be used for real world applications. The improved parser combinators remain very easy to use and are still able to handle ambiguous grammars.

1 Introduction

Parser combinators [3,6,5,8] are a beautiful illustration of the use of higher order functions and currying. By using a small set of parser combinators it becomes possible to construct parsers for ambiguous grammars in a very elegant and clear way. The basis of parser combinators is the list of successes method introduced by Wadler [13]. Each parser yields a list of results: all successful parsings of the input. When the parser fails this list is empty. In this way it is very easy to handle ambiguous parsers that define multiple ways to parse a given input.

Despite the elegant formulation and the ability to handle ambiguous grammars, parser combinators are rarely used in practice. For small inputs these parsers work nice and smoothly. For realistically sized inputs the parsers consume extraordinary amounts of time and space due to their polynomial complexity.

In this paper we show that the amounts of time and memory required by the combinators parsers can drastically be reduced by improving the implementation of the parser combinators and providing a little more information about the grammar in the parser that is written. This additional information can reduce the complexity of the parser from polynomial to linear. Although the implementation of the parser combinators becomes more complex, their use in combinator parsers remains as simple as in the original setting.

This paper starts with a short review of classical parser combinators. The proposed improvements are presented hereafter. We use a running parsing example to measure the effect of the improvements.

[*] Sponsored by STW project NWI.4411

2 Conventional Parser Combinators

There are basically two approaches to construct a parser for a given grammar[1]. The first approach is based upon the construction of a finite state automaton determining the symbols that can be accepted. This approach is used in many parser generators like *yacc* [7,1,10], *Happy* [4] and *Ratatosk* [9]. In general the constructed parsers are efficient, but cumbersome to achieve and there might be a serious distance between the automaton accepting input tokens and the original grammar. If we use a generator there is a barrier between parsing and using the parsed items in the rest of the program.

The second approach to achieve a parser is to create a recursive descent parser. Such a parser follows the rules of the grammar directly. In order to ensure termination the grammars should not be left-recursive. Parser combinators are a set of higher order functions that are convenient in the construction of recursive descent parsers. The parser combinators provide primitives to recognize symbols and the sequential or alternative composition of parsers.

The advantages of parser combinators are that they are easy to use, elegant and clear. Due to the fact that the obtained parsers directly correspond to the grammar there is no separate parser generator needed. Since parser combinators are ordinary functions they are easy to understand and use. It is easy to extend the set of parser combinators with new handy combinators whenever this is desired. The full power of the functional programming language is available to construct parsers, this implies for instance that it is possible to use second order grammars. Finally, there are no problems to transfer parsed items from the parser to the manipulation functions.

Conventional parser combinators are described at many places in the literature e.g. [3,6,5,8]. Here we follow the approach outlined in [8], using the functional programming language Clean [11]. We restrict ourselves to a small, but complete set of parser combinators to illustrate our improvements.

A `Parser` is a function that takes a list of input symbols as argument and produces a `ParsResult`. A `ParsResult` is the list of successes. Each success is a tuple containing the rest of the list of input symbols and the item found. The types `Parser` and `ParsResult` are parameterized by the type of symbols to be recognized, `s`, and the type of the result, `r`.

```
:: Parser s r :== [s] -> ParsResult s r
:: ParsResult s r :== [([s],r)]
```

In the examples in this paper we will use characters, `Char`, as symbols in the lists of elements to be parsed, but in general they can be of any datatype.

2.1 Basic Parser Combinators

The basic combinator to recognize a given symbol in the input is `symbol`. This parser combinator takes the symbol to be recognized as its argument. When the first token in the input is equal to this symbol there is a single success. In all