Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Efficient parsing with parser combinators

Jan Kurš [a,*], Jan Vraný [b], Mohammad Ghafari [a], Mircea Lungu [c], Oscar Nierstrasz [a]

[a] Software Composition Group, University of Bern, Switzerland
[b] Software Engineering Group, Czech Technical University, Czech Republic
[c] Software Engineering and Architecture Group, University of Groningen, Netherlands

ARTICLE INFO

Article history:
Received 3 January 2017
Received in revised form 12 November 2017
Accepted 1 December 2017
Available online xxxx

Keywords:
Optimizations
Parsing expression grammars
Parser combinators

ABSTRACT

Parser combinators offer a universal and flexible approach to parsing. They follow the structure of an underlying grammar, are modular, well-structured, easy to maintain, and can recognize a large variety of languages including context-sensitive ones. However, these advantages introduce a noticeable performance overhead mainly because the same powerful parsing algorithm is used to recognize even simple languages. Time-wise, parser combinators cannot compete with parsers generated by well-performing parser generators or optimized hand-written code.

Techniques exist to achieve a linear asymptotic performance of parser combinators, yet there is a significant constant multiplier. The multiplier can be lowered to some degree, but this requires advanced meta-programming techniques, such as staging or macros, that depend heavily on the underlying language technology.

In this work we present a language-agnostic solution. We optimize the performance of parsing combinators with specializations of parsing strategies. For each combinator, we analyze the language parsed by the combinator and choose the most efficient parsing strategy. By adapting a parsing strategy for different parser combinators we achieve performance comparable to that of hand-written or optimized parsers while preserving the advantages of parsers combinators.

1. Introduction

A parser combinator is a higher-order function that takes one or more parsers as input and produces a new parser as its output. Parser combinators [53,39] represent a popular approach to parsing. They are straightforward to construct, readable, modular, well-structured and easy to maintain. Parser combinators are also highly expressive as they can parse not only context-free languages but also some context-sensitive ones (*e.g.*, layout-sensitive languages [24,2]).

Nevertheless, parser combinators at the moment are considered more a technology for prototyping than for actual deployment, since the expressive power of parser combinators comes at the price of less efficiency. A parser combinator uses the full power of a Turing-equivalent formalism to recognize even simple languages that could be recognized by finite state

* Corresponding author.
E-mail address: kurs@inf.unibe.ch (J. Kurš).
URLs: http://www.scg.unibe.ch (J. Kurš), http://www.swing.fit.cvut.cz (J. Vraný), http://www.cs.rug.nl/search (M. Lungu).

https://doi.org/10.1016/j.scico.2017.12.001
0167-6423/

machines or pushdown automata. Consequently, parser combinators cannot reach the peak performance of parser generators [37], hand-written parsers, or optimized code [6] (see section 5).

Meta-programming approaches such as macros [9] and staging [47] have been applied to Scala parser combinators [39] with significant performance improvements [6,28]. In general, these approaches remove composition overhead and intermediate representations. Other approaches attack performance problems using more efficient structures, macros *etc.* (see *Parboiled 2*,[1] *attoparsec*[2] or *FParsec*[3]).

While Scala optimizations rely on the power of the Scala compiler, and other solutions exploit knowledge about the internal implementation, our solution provides optimizations based on the domain knowledge of the parsing formalism, is language-agnostic, and does not rely on specifics of the internal implementation.

In this work we build on our idea of a *parser compiler* [34], which optimizes the parser for a language by using specialized parsing strategies for different parser combinators. A strategy is selected based on the language the given parser combinator parses. Different subsets of a language are matched to different parsing strategies. Each of these strategies fits the best for the given subset. Our approach preserves all the advantages of parser combinators and does not impose any restrictions on their expressiveness.

We choose as a case study the performance of PetitParser [46,31] – a parser combinator framework using the parsing expression grammar (PEG) formalism [16]. As a validation of the ideas presented in this work, we implement a parser combinator compiler (for short, a *parser compiler*): an ahead of time source-to-source translator. This compiler (i) analyzes parser combinators of PetitParser, (ii) chooses the most appropriate parsing strategy for each of them, and (iii) integrates these strategies into a single top-down parser, which is equivalent to the original parser. Based on our measurements covering six different grammars for PetitParser,[4] our parser compiler offers a performance improvement of a factor ranging from two to ten, depending on the grammar. Based on our Smalltalk case study, our approach is only 10% slower than a highly-optimized, hand-written parser.

To summarize, this paper makes the following contributions: i) a discussion of performance bottlenecks of parser combinators, ii) a description of optimization techniques addressing these bottlenecks, and iii) a detailed analysis of their effectiveness.

The paper is organized as follows: We explain the parsing overhead of parser combinators using a concrete example in section 2. In section 3 we introduce a parser compiler, which reduces the existing overheads of parser combinators. In section 4 we describe in detail how we identify and apply different parsing strategies. In section 5 we analyze the performance impact of different parsing strategies. In section 6 we briefly discuss related work and finally, section 7 concludes this paper.

2. Motivating example

In this section, we present, as an example, the parsing overhead of PetitParser. PetitParser [46,31] is a parser combinator framework [24] that uses packrat parsing [15], scannerless parsing [52], and parsing expression grammars (PEGs) [16].

We identify the most critical performance bottlenecks of PetitParser and explain them using an example with a grammar describing a simple programming language as shown in Listing 1 (we use a simplified version of the actual PetitParser DSL as described in detail in Table B.2).

A program conforming to this grammar consists of a non-empty sequence of classes. A class starts with `classToken`, followed by an `idToken` and `body`. The `classToken` rule is a `'class'` keyword that must be followed by a space that is not consumed. This is achieved by using the *and* predicate `&` followed by `#space`, which expects a space or a tabulator. Identifiers start with a letter followed by any number of letters or digits. The class keyword and identifiers are transformed into instances of `Token`, which maintain information about start and end positions and the string value of a token. There is a semantic action associated to a `class` rule that creates an instance of `ClassNode` filled with an identifier value and a class body.

A class body is indentation-sensitive, *i.e.*, `indent` and `dedent` determine the scope of a class (instead of commonly used brackets *e.g.*, `{` and `}`). The `indent` and `dedent` rules determine whether a line is indented, *i.e.*, on a column strictly greater than the previous line or dedented, *i.e.*, column strictly smaller than the previous line. The `indent` and `dedent` rules are represented by specialized action parsers that manipulate an indentation stack by pushing and popping the current indentation level, similarly to the scanner of Python.[5] The class body contains a sequence of classes and methods.

[1] http://www.webcitation.org/6k6195CiS.
[2] http://www.webcitation.org/6k61DC2EA.
[3] http://www.webcitation.org/6k61HcnHU.
[4] Expressions, Smalltalk, Java, Ruby, and Python.
[5] http://www.webcitation.org/6k637RJ7V.

Please cite this article in press as: J. Kurš et al., Efficient parsing with parser combinators, Sci. Comput. Program. (2017), https://doi.org/10.1016/j.scico.2017.12.001