Computer Languages, Systems & Structures 44 (2015) 114–140

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Bounded seas

Jan Kurš*, Mircea Lungu, Rathesan Iyadurai, Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland[1]

ARTICLE INFO

Article history:
Received 1 June 2015
Accepted 7 August 2015
Available online 24 August 2015

Keywords:
Semi-parsing
Island parsing
Parsing expression grammars

ABSTRACT

Imprecise manipulation of source code (semi-parsing) is useful for tasks such as robust parsing, error recovery, lexical analysis, and rapid development of parsers for data extraction. An island grammar precisely defines only a subset of a language syntax (islands), while the rest of the syntax (water) is defined imprecisely.

Usually water is defined as the negation of islands. Albeit simple, such a definition of water is naïve and impedes composition of islands. When developing an island grammar, sooner or later a language engineer has to create water tailored to each individual island. Such an approach is fragile, because water can change with any change of a grammar. It is time-consuming, because water is defined manually by an engineer and not automatically. Finally, an island surrounded by water cannot be reused because water has to be defined for every grammar individually.

In this paper we propose a new technique of island parsing — bounded seas. Bounded seas are composable, robust, reusable and easy to use because island-specific water is created automatically. Our work focuses on applications of island parsing to data extraction from source code. We have integrated bounded seas into a parser combinator framework as a demonstration of their composability and reusability.

1. Introduction

Island grammars [1] offer a way to parse input without complete knowledge of the target grammar. They are especially useful for extracting selected information from source files, reverse engineering and similar applications. The approach assumes that only a subset of the language syntax is known or of interest (the islands), while the rest of the syntax is undefined (the water). During parsing, any unrecognized input (water) is skipped until an island is found.

A common misconception is that water should consume everything until some island is detected. Rules for such water are easy to define, but they cause composability problems. Consider a parser where local variables are defined as islands within a method body. Now suppose a method declaring no local variables is followed by one that does. In this case the water might consume the end of the first method as well as the start of the second method until a variable declaration is found. The method variables from the second method will then be improperly assigned to the first one.

In practice, language engineers define many small islands to guide the parsing process. However it is difficult to define such islands in a robust way so that they function correctly in multiple contexts. As a consequence they are neither reusable nor composable.

* Corresponding author.
E-mail address: kurs@iam.unibe.ch (J. Kurš).
[1] http://scg.unibe.ch

http://dx.doi.org/10.1016/j.cl.2015.08.002
1477-8424/

To prevent our variable declaring island from skipping to another method, we have to make its water stop at most at the end of a method. In general, we have to analyze and update each particular island's water, depending on its context. Yet island-specific water is fragile, hard to define and it is not reusable. It is fragile, because it requires re-evaluation by a language engineer after any change in a grammar. It is hard to define, because it requires the engineer's time for detailed analysis of a grammar. It is not reusable, because island-specific water depends on rules following the island, thus it is tailored to the context in which the island is used — it is not general.

In this paper we propose a new technique for island parsing: *bounded seas* [2]. Bounded seas are composable, reusable, robust and easy to use. The key idea of bounded seas is that specialized water is defined for each particular island (depending on the context of the island) so that an island can be embedded into any rule. To achieve such composability, water is not allowed to consume any input that would be consumed by a following rule.

To prevent fragility and to improve reusability, we compute water automatically, without user interaction. To prove feasibility, we integrate bounded seas into Petit Parser [3], a PEG-based [4] (see Appendix A) parser combinator [5] framework.

In addition to our previous work [2] we evaluate the usability of bounded seas in two case studies, we present a performance study, and we provide more details about the implementation. The contributions of the paper are:

- the definition of bounded seas, a composable, reusable, robust and easy method of island parsing;
- a formalization of bounded seas for PEGs;
- an implementation of bounded seas in a PEG-based parser combinator framework; and
- case studies of semi-parsing of Java and Ruby using bounded seas.

Structure: Section 2 motivates this work by presenting the limitations of island grammars with an example. Section 3 presents our solution to overcoming these limitations by introducing bounded seas. Section 4 introduces a sea operator for PEGs, which creates a bounded sea from an arbitrary PEG expression. Section 5 presents our implementation of bounded seas in PetitParser. Section 6 discusses the applicability of bounded seas to GLL parsers, design decisions and some limitations of bounded seas. Section 7 analyzes how well bounded seas perform compare to other island parsers. Section 8 analyzes usability of bounded seas for context-sensitive grammars, particularly for indentation-sensitive grammars. Section 9 surveys other semi-parsing techniques and highlights similarities and differences between them and bounded seas. Finally, Section 10 concludes this paper with a summary of the contributions.

2. Motivating example

Let us consider the source code in Listing 1 written in some proprietary object-oriented language. We don't have a grammar specification for the code, because the parser was written using *ad hoc* techniques, and we do not have access to its implementation. Let us suppose that our task is to extract class and method names. Classes may be contained within other classes and we need to keep track of which class each method belongs to.

Listing 1. Source code of the `Shape` class in a proprietary language.

```
class Shape
    Color color;

    method getColor {
        return color;
    }
    int uid = UIDGenerator.newUID;
endclass
```

2.1. Why not use regular expressions?

To extract a flat list of method names, we could use regular expressions. We need, however, to keep track of the nesting of classes and methods within classes. Regular expressions are only capable of keeping track of finite state, so are formally too weak to analyze our input. To deal with nested structures, we need at least a context-free parser.

Modern implementations of regular expression frameworks can parse more than regular languages (*e.g.*, using recursive patterns[2]). Such powerful frameworks can handle our rather simple task. However regular expressions are not meant to specify complex grammars since they tend to be hard to maintain when the complexity of the grammar grows.

[2] http://perldoc.perl.org/perlre.html