

Extracting Monadic Parsers from Proofs

Ulrich Berger, Alison Jones, Monika Seisenberger

Swansea University

Abstract

This talk outlines a proof-theoretic approach to developing correct and terminating monadic parsers. Using a modified realisability interpretation, we extract provably correct and terminating programs from formal proofs. If the proof system is proven to be correct, then any extracted program is guaranteed to be so. By extracting parsers, we can ensure that they are correct, complete and terminating for any input. The work is ongoing, and is being carried out in the interactive proof system MINLOG.

Introduction

Parsing any language can be problematic. With natural language, the problems are exacerbated by its inherent ambiguities and irregularities. In Linguistics, syntax can be analysed using several different formalisms, none of which is universally agreed upon. At the semantic level there is even less consensus. This sets natural languages apart from more formal languages encountered by parsers. Despite these differences, however, there are key properties that all parsers should ideally have; they should be complete and correct (with respect to a given grammar), and they should terminate with any input. The importance of termination is demonstrated by the well-known left-recursion problem.

In this talk, we focus on monadic parsers, which allow us to generate parse trees as side-effects in pure functional programming languages such as Haskell. Another advantage is that monadic parser combinators naturally accommodate non-determinism and ambiguous input. Monadic parsers can be directly implemented in Haskell. This is the approach taken in much of the literature (see Hutton, Wadler). Testing these parsers could check (to a degree) that they work as expected. A more rigorous approach would be to verify them by formalising the implementations and proving that they work as expected. A third option, and the one explored here, is to extract correct parsers directly from the formal proofs.

The Parsing Problem

For any parser $P \subseteq S \times A$ the parsing problem can be stated as follows: Given a string $s \in S$, find all a such that $(s, a) \in P$.

In the *partial parsing problem*, we only require an initial segment of s to be parsed: given a string $s \in S$, find all pairs (a, r) such that s is equal to $s' \upharpoonright r$ and $(s', a) \in P$. Formally, this partial parsing problem can be represented by a *parsing function* $[P] : S \rightarrow \mathbb{P}(A \times S)$, defined by:

$$[P](s) := \{(a, r) \mid \exists s'. s = s' \upharpoonright r \wedge (s', a) \in P\}$$

I.e., the input string s is parsed into a result a and the remaining unparsed string r .

Usually one is interested in parsers with parsing functions that always yield a finite set of results. We call such parsers *finitely branching*:

$$\text{FB}_{S,A}(P) := \forall s \in S. [P](s) \in \mathbb{P}_{<\omega}(A \times S) \quad (1)$$

Extracting Parsing Programs

The reason why we are interested in finitely branching parsers is that, from a constructive proof that a parser P is finitely branching, we can extract a program $\mathbf{p}$ that implements the parsing function $[P]$. Using Haskell notation, $\mathbf{p}$ will have the type:

`p :: String -> [(a, String)]`

where $\mathbf{a}$ is the type of representations of parsing results for P . Note that the type of $\mathbf{p}$ is the type of monadic parsers. Due to the way in which finite sets have been inductively defined, the set of results is represented by a finite list of results at the program level. Essentially, we extract monadic parsers in the style of Hutton, but with the additional benefit of a formal proof of correctness, completeness and termination. Instead of building complex parsers from simple ones at the *program* level, we will do so at the *proof* level.

The considerations above should be viewed as initial steps towards a library of proof tactics supporting a combinatorial style of proving that a parser is finitely branching which can be automatized. For example, take the following definition and function for the monadic bind:

$$P \gg= Q = \{(s \upharpoonright r, b) \mid \exists a. (s, a) \in P \wedge (r, b) \in Q_a\}$$

$$[P \gg= Q](s) = \{(b, t) \mid \exists a, r. (a, r) \in [P](s) \wedge (b, t) \in [Q_a](r)\}$$

Using (1), we can extract a corresponding program by proving the following:

$$\text{FB}_{S,A}(P) \rightarrow \forall a \in A. \text{FB}_{S,B}(Q_a) \rightarrow \text{FB}_{S,B}(P \gg= Q)$$

Conclusion

A fundamental benefit of treating monadic parsers at the proof level is that we can often work directly with parsers (that is, parser relations) as opposed to their parsing functions. This abstraction extends to our extraction of parser programs from proofs of set-theoretic properties, as they are not dependent on our implementation of sets in MINLOG.