Science of Computer Programming 84 (2014) 52–76

Contents lists available at ScienceDirect

Science of Computer Programming

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

On the semantics of parsing actions

Hayo Thielecke *

School of Computer Science, University of Birmingham, Birmingham B15 2TT, United Kingdom

HIGHLIGHTS

- We develop a categorical semantics for parsing actions.
- Via the semantics, left-recursion elimination (in syntax) is shown to lead to continuation passing (in semantics).
- We define a type-and-effect calculus idealizing parser generators together with its abstract machine semantics.

ARTICLE INFO

Article history:
Received 17 December 2012
Received in revised form 22 March 2013
Accepted 30 April 2013
Available online 11 May 2013

Keywords:
Semantics
Abstract machines
Continuations
Parser generators
Left recursion elimination

ABSTRACT

Parsers, whether constructed by hand or automatically via a parser generator tool, typically need to compute some useful semantic information in addition to the purely syntactic analysis of their input. Semantic actions may be added to parsing code by hand, or the parser generator may have its own syntax for annotating grammar rules with semantic actions. In this paper, we take a functional programming view of such actions. We use concepts from the semantics of mostly functional programming languages and adapt them to give meaning to the actions of the parser. Specifically, the semantics is inspired by the categorical semantics of lambda calculi and the use of premonoidal categories for the semantics of effects in programming languages. This framework is then applied to our leading example, the transformation of grammars to eliminate left recursion. The syntactic transformation of left-recursion elimination leads to a corresponding semantic transformation of the actions for the grammar. We prove the semantic transformation correct and relate it to continuation passing style, a widely studied transformation in lambda calculi and functional programming. As an idealization of the input language of parser generators, we define a call-by-value calculus with first-order functions and a type-and-effect system where the effects are given by sequences of grammar symbols. The account of left-recursion elimination is then extended to this calculus.

1. Introduction

When writing interpreters or denotational semantics of programming languages, one aims to define the meaning of an expression in a clean, compositional style. As usually understood in computer science, the principle of compositionality states that the meaning of an expression arises as the meaning of its constituent parts.

For example, in a compositional semantics for arithmetic expressions, the semantic definitions may even appear trivial and no more than a font change:

$$[\![E_1 - E_2]\!] = [\![E_1]\!] - [\![E_2]\!]$$

Of course, the simplicity of such rules is due to the fact that the semantic operations (in this case subtraction $-$) and the syntax (in this case -) that they interpret are chosen to be as similar as possible. A more formal statement of this idea is initial

* Tel.: +44 121 414 2986.
E-mail address: H.Thielecke@cs.bham.ac.uk.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.04.010

algebra semantics [1]. The constructors for the syntax trees form the initial algebra, so that any choice of corresponding semantic operations induces a unique algebra homomorphism. Intuitively, the initial algebra property means that we take our syntax tree for a given expression, replace syntactic operation (say, –) everywhere by their semantic counterpart (subtraction in this case), and then collapse the resulting tree by evaluating it to an integer.

The simple picture of semantics as tree node replacement followed by evaluation assumes that parsing has been taken care of, in a reasonable separation of concerns. If, however, parsing is taken into account, the situation becomes more complicated. We would still hope any semantics to be as compositional as possible (sometimes called "syntax-directed" in compiling [2]), but now the grammar must be suitable for the parsing technology at hand. Many of the grammars widely used in semantics are not, including the example above, assuming a grammar rule $E \coloneq E\text{-}E$, or the usual grammar for lambda calculus with the rule for application as juxtaposition, $M \coloneq M\,M$. Both these rules exhibit left recursion (and also result in an ambiguous grammar). There are standard techniques for transforming grammars to make them more amenable to parsing [2]. If we take compositionality seriously, the semantics should also change to reflect the new grammar; moreover the transformation should be correct relative to the old grammar and its compositional semantics. The old grammar, while unsuitable for parsing, may give a more direct meaning to the syntactic constructs where the intended meaning may be more evident than for the transformed grammar and its semantics.

Our running example of left recursion elimination and its semantics will be a simple expression grammar. We first discuss it informally, in the hope that it already provides some intuition of continuations arising in parsing.

Example 1.1. Consider the following grammar rules, where the rule (2) has an immediate left-recursion for E:

$$E \coloneq 1 \tag{1}$$
$$E \coloneq E - E \tag{2}$$

We eliminate the left recursion from this grammar using the standard technique as found in compiling texts [2]. This construction involves the introduction of a new grammar symbol E' (together with some rules for it), and replacing (1) with a new grammar rule that uses the new symbol in the rightmost position:

$$E \coloneq 1\ E' \tag{3}$$

The new symbol E' has the following rules, which replace rule (2) above:

$$E' \coloneq -\ E\ E' \tag{4}$$
$$E' \coloneq \varepsilon \tag{5}$$

The semantics of E' needs an argument for receiving the value of its left context. For example in (3), the value 1 needs to be passed to the semantic action for E'. Now compare the original (1) to the transformed (3). The original rule is in direct style, in the sense that 1 is returned as the value of the occurrence of E. By contrast, the transformed rule (3) is in continuation passing style, in that 1 is not *returned* here; rather, it is passed to its continuation, given by E'.

As research communities, parsing and semantics can be quite separated, with (to put it crudely) the former using formal language theory and the latter lambda calculi. The contribution of the present paper is in bridging the gap between parsing a language and its semantics. To do so, we use formal tools that were originally developed on the semantic side. One such semantic tool is category theory. Due to its abstract nature, it can capture both syntax and semantics.

One of these formal tools will be premonoidal categories [3], which were originally developed as an alternative to Moggi's monads as notions of computation [4] for the semantics of functional languages with effects (such as ML). Much as monads in category theory are related to monads in Haskell, premonoidal categories have a functional programming analogue, Hughes's arrows [5,6].

The idea of the "tensor" $\otimes$ in premonoidal categories is easy to grasp from a functional programming point of view. Suppose we have a function $f : X_1 \to X_2$. Then we can still run the same function while carrying along an additional value of type Y. That gives us two new functions, by multiplying f with Y from the left or right, as it were:

$$\begin{aligned} f \otimes Y &= \lambda(x : X_1, y : Y).(f(x), y)) \\ &: (X_1 \otimes Y) \longrightarrow (X_2 \otimes Y) \\ Y \otimes f &= \lambda(y : Y, x : X_1).(y, f(x)) \\ &: (Y \otimes X_1) \longrightarrow (Y \otimes X_2) \end{aligned}$$

(For the notational conventions we will use regarding letters, arrows, etc., see Fig. 1.) While category theory can be used to structure functional programs and to reason about their meaning algebraically, categories are not restricted to morphisms being functions. Syntactic structures, such as strings, sequences, paths, traces, etc., can also be used to construct categories. For our purposes here, it will be useful to define a $\otimes$ that simply concatenates strings:

$$w \otimes \beta \stackrel{\text{def}}{=} w\,\beta$$