On the Semantics of Parsing Actions

Hayo Thielecke

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

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.

Keywords: semantics, abstract machines, continuations, substructural types, parser generators, left recursion elimination

Email address: `H.Thielecke@cs.bham.ac.uk` (Hayo Thielecke)

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:

$$[\![\, E_1 \texttt{-} E_2 \,]\!] = [\![\, E_1 \,]\!] - [\![\, E_2 \,]\!]$$

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 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 $E :- E \texttt{-} E$, or the usual grammar for lambda calculus with the rule for application as juxtaposition, $M :- M\, M$. 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:

$$E \;:\!-\; \mathtt{1} \tag{1}$$

$$E \;:\!-\; E - E \tag{2}$$

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:

$$E \;:\!-\; \mathtt{1}\, E' \tag{3}$$

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

$$E' \;:\!-\; \mathtt{-}\, E\, E' \tag{4}$$

$$E' \;:\!-\; \varepsilon \tag{5}$$

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:

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

(For the notational conventions we will use regarding letters, arrows, etc, see Figure 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:

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

This operation will be used to characterise leftmost derivations. Suppose we have a grammar rule $A :- \alpha$. By multiplying it with some string w not containing any non-terminals on the left, and a string β (possibly containing non-terminals), we get a leftmost derivation step:

$$(w\,A\,\beta) \longrightarrow (w\,\alpha\,\beta)$$

Leftmost and rightmost derivations characterise the two main classes of parser generators, LL and LR [2]. The former include for example ANTLR [7], while the latter include LALR(1) parsers generators such as yacc.

Usually it is more convenient to calculate with lambda terms rather than translate them into their categorical semantics in a Cartesian closed category. For our purposes here, however, diagram chases are convenient and perspicuous. Since the meaning of a string to be parsed is generated compositionally from its derivation, we can take the derivation, translate it into the corresponding semantic diagram, and chase morphisms to reason about semantic equality.

Background and prerequisites

This paper assumes only basic familiarity with grammars and parsing, at the level of an undergraduate compiling course. To the extent that it is needed, some category theory will be presented as a form of simply-typed functional programming. Familiarity with the semantics of lambda calculus in a Cartesian closed category will be useful (as covered in programming language semantics texts, e.g. by Gunter [8]). The category theory used in this paper is elementary and closely related to functional programming. Readers who prefer type theory can safely view it as a form of type system that emphasizes sequential composition and lends itself to proof by diagram chase.

Outline of the paper

Section 2 describes parser generators in terms of simple abstract machines. Section 3 casts the notion of leftmost derivation, which is fundamental for LL parsers, into a mathematical form adapted from premonoidal categories. The semantics of derivations is then defined compositionally from their associated parsing actions in Section 4. With the framework in place, Section 5 then uses it on the leading example: the elimination of left recursion induces a semantic transformation. We prove the correctness of the semantic transformation in Section 6 and reconstruct it as a form of continuation passing in Section 7. A calculus that idealises parser generators is defined in Section 8, and the left-recursion elimination transform is adapted to the calculus in Section 9. The calculus is given an operational semantics by way of an abstract machine in Section 10. Finally, Section 11 concludes with a discussion of related work and directions for further research.

An earlier version of the present paper has appeared in PPDP 2012 [9]. It was left to further work at the end of the conference version whether the language for writing semantic actions could be made more realistic. The present paper addresses this question in the new sections on the L-calculus and L^2 machine, Section 8, 9, and 10.

2. Parser generators

We recall some definitions, as can be found in any compiling text. A grammar is of the form

$$\mathcal{G} = (\mathbf{T}_\mathcal{G}, \mathbf{N}_\mathcal{G}, \mathbf{S}_\mathcal{G}, :-)$$

where $\mathbf{T}_\mathcal{G}$ is a finite set of terminal symbols, $\mathbf{N}_\mathcal{G}$ is a finite set of non-terminal symbols, $\mathbf{S}_\mathcal{G}$ is the start symbol, and :– is a finite binary relation between non–terminal symbols and strings of symbols.

LL parsers attempt to construct a leftmost derivation of the string they are parsing [2]. If we leave open the question of *how* the parser decides on its next move, we can formulate LL parsers as a (remarkably simple) abstract machine, albeit a non-deterministic one. Given a grammar, an LL parser is an abstract machine of the following form. Configurations are pairs $\langle w, \sigma \rangle$. Here w is the remaining input string, consisting of terminal symbols, and σ is the parsing stack, consisting of a sequence of grammar symbols (each of which can be terminal or non-terminal). We write the top of the parsing stack on the left. The parser has two kinds of transitions: matching and predicting. In a matching transition, an input symbol a is removed both from the stack and the input. In a predicting transition, a non-terminal A at the top of the parsing stack is popped off and replaced by the right-hand-side of a grammar rule $A :- \alpha$ for it.

$$\begin{array}{rcl}
\langle a\, w, a\, \sigma \rangle & \rightsquigarrow & \langle w, \sigma \rangle \\
\langle w, A\, \sigma \rangle & \rightsquigarrow & \langle w, \alpha\, \sigma \rangle \\
 & & \text{if there is a rule } A :- \alpha
\end{array}$$

The initial configuration consists of the initial input w and the parsing stack holding start symbol S of the grammar, that is $\langle w, S \rangle$. The accepting configuration has both an empty input and an empty parsing stack $\langle \varepsilon, \varepsilon \rangle$. As defined here, the machine is highly non-deterministic, since for a given A at the top of the parsing stack there may be many different rules $A :- \alpha_1$, ..., $A :- \alpha_n$ with different right-hand sides. Given bad choices, the machine may get stuck in a state of the form $\langle a\, w, b\, \sigma \rangle$ for some $a \neq b$.

A, B, C, E, L	:	nonterminal symbols of a grammar
a, b	:	terminal symbols of a grammar
f, g, k	:	functions, variables of function type
x, y, z, p	:	variables
X, Y, X_1	:	types, objects in a category
$X \otimes Y$	:	product type, premonoidal functor
(x, y)	:	pair of type $X \otimes Y$
`()`	:	empty tuple of type **Unit**
$\alpha, \beta, \gamma, \delta, \varphi, \psi$	:	strings of terminals or nonterminals
ε	:	empty string
λ	:	lambda-abstraction; not used for strings
v, w, w_1, w_2	:	strings of terminal symbols
$\alpha\,\beta$	:	concatenation of strings α and β
$A :- \alpha$	:	grammar rule for replacing A by α
$f : X \to Y$	:	f is a morphism from X to Y
$\alpha \to_{\mathcal{G}} \beta$	:	derivation of string β from α (a special case of a morphism)
$\rightsquigarrow$	:	abstract machine transition step
$\langle w, \sigma, k \rangle$	:	abstract machine configuration
$f \cdot g$	:	composition of morphisms
		in diagrammatic order: first$f$, then g
$[\,]$	:	empty list or path
$[\![\, \alpha \,]\!]$	:	semantics of a string α as a type or set
$[\![\, d \,]\!]$	:	semantics of a derivation d as a function
$M[x \mapsto N]$	:	substitution in M of x by N

Figure 1: Notational conventions

In parser construction, the problem is how to make the above non-deterministic machine deterministic. For LL parsers, the choice between rules is made using lookahead, leading to parser classes such as LL(1) or LL(k), using 1 or some larger numbers of symbols in the input w. In particular, some modern parser generator such as ANTLR [7] use large lookaheads where required.

Here we are not concerned with the parser construction; on the contrary, we assume that there is a parser generator that successfully generates a parser for the grammar at hand. In other words, the above abstract machine is *assumed* to be deterministic, for example by looking at the first k symbols of the input.

The problem that this paper aims to address is how to bridge the gap between the parsing abstract machine and the semantics of the language that is being parsed. We extend the parsing abstract machine with a third component that iteratively computes meaning during the parsing. The extended machine makes a transition

$$\langle w, A\,\sigma, k_1\rangle \quad\rightsquigarrow\quad \langle w, \alpha\,\sigma, k_2\rangle$$

whenever the parsing machine decides to make a prediction transition using a rule $A :- \alpha$. (In ANTLR, semantic information can also be used to guide the choice of predictive transition, in addition to lookahead. We do not model this feature.) Each such rule has an associated semantic action $[\![\, A :- \alpha \,]\!]$ that tells us how to compute the new semantic component k_2 from the preceding one k_1. As usual in programming language theory, the semantic actions will be idealised using λ-calculus. The semantic actions should arise in a compositional way from the syntactic transitions of the parser constructing a leftmost derivation. Ideally, the connection between syntax and semantics should be maintained even if the grammar needs to be transformed. Grammars may have to be rewritten to make them suitable for parsing. Specifically, left recursion is a problem for parsers relying on lookahead [2]. There is a standard grammar transformation, left recursion elimination, that removes the problem for many useful grammars (such as expressions in arithmetic). It is easy enough to understand if we are only interested in the language as a set of strings; but if we take a more fine-grained view of derivations and their semantic actions, the semantic transformation corresponding to the grammar refactoring is quite subtle. It involves introducing explicit dependence on context, which we show to be a form of a widely studied semantic transformation, continuation-passing style (CPS) [10, 11, 12].

3. From grammars to L-categories

For each grammar, we define a category with some structure that will be useful for defining the semantic actions.

Definition 3.1 (L-category of a grammar) Given a grammar $\mathcal{G}$, we define a directed graph $\mathbf{Graph}(\mathcal{G})$ as follows:

- The nodes of $\mathbf{Graph}(\mathcal{G})$ are strings of grammar symbols α.
- Whenever there is a grammar rule $A :\!- \alpha$, a terminal string w and a string β, $\mathbf{Graph}(\mathcal{G})$ has an edge from $w\,A\,\beta$ to $w\,\alpha\,\beta$.

The L-category for $\mathcal{G}$ is defined as the path category [13] for the graph $\mathbf{Graph}(\mathcal{G})$. Morphisms in the L-category are called leftmost derivations.

A note on notation: composition of morphisms f and g is written as $f \cdot g$ in diagrammatic order: first f, then g. This is the opposite order compared to function composition, written as $g \circ f$. Having the same notation for composition of morphisms and sequences is convenient for our purposes here.

We write morphisms in the L-category as lists of pairs of strings of the form $(w\,A\,\beta, w\,\alpha\,\beta)$. Composition of morphisms is by list concatenation. The identity morphism $\mathsf{id}_\alpha : \alpha \longrightarrow \alpha$ is given by the empty list. Each morphism in a category has a unique domain and codomain. As they are not evident in case of the empty list, we need to tag each morphism with its domain and codomain. However, as long as they are evident from the context, we omit these additional tags and represent morphisms as lists.

The reason for defining leftmost derivations via L-categories rather than in the style found in most compiler construction texts [2] is that they allow us to introduce some extra structure.

For each string of grammar symbols α of $\mathcal{G}$, the L-category has an endofunctor, written as $- \otimes \alpha$. That is to say, for each leftmost derivation $d : \beta \to \gamma$, there is a leftmost derivation

$$(d \otimes \alpha) : (\beta \otimes \alpha) \to (\gamma \otimes \alpha)$$

On objects, $- \otimes \alpha$ is string concatenation, that is,

$$\beta \otimes \alpha \overset{\text{def}}{=} \beta\,\alpha$$

On morphisms, $- \otimes \alpha$ extends each step in the derivation with α. Writing :: for list cons, as in ML, we define:

$$\begin{aligned} [\,] \otimes \alpha &\overset{\text{def}}{=} [\,] \\ ((w\,A\,\gamma, w\,\delta\,\beta)::p) \otimes \alpha &\overset{\text{def}}{=} ((w\,A\,\gamma\,\alpha, w\,\delta\,\beta\,\alpha)::(p \otimes \alpha) \end{aligned}$$

Notice that we do not generally define a functor $\alpha \otimes -$ that works symmetrically. If α contains non-terminal symbols, then adding it *on the left* of a leftmost derivation does not produce a leftmost derivation. However, for string w consisting entirely of terminal symbols, the leftmost character of a derivation is preserved if we add such a string everywhere on the left. Hence we define an endofunctor $w \otimes -$ for each terminal string w. For a leftmost derivation $d : \beta \to \gamma$, we have

$$(w \otimes d) : (w\,\beta) \to (w\,\gamma)$$

defined by

$$\begin{aligned} w \otimes [\,] &\overset{\text{def}}{=} [\,] \\ w \otimes ((w_2\,A\,\gamma, w_2\,\delta\,\beta)::p) &\overset{\text{def}}{=} ((w\,w_2\,A\,\gamma\,\alpha, w\,w_2\,\delta\,\beta\,\alpha) \\ &\qquad ::(w \otimes p) \end{aligned}$$

The definition of L-category is inspired of Power and Robinson's premonoidal categories [3, 14] and by Lambek's syntactic calculus [15]. In particular, the idea of a functor $- \otimes \alpha$ for each object comes from premonoidal categories, whereas the distinction between left and right functors is reminiscent of the non-commutative contexts in Lambek's calculus.

It is worth noting that L-categories have a very substructural flavour, as it were. We do not even assume a symmetry isomorphism

$$\alpha \otimes \beta \cong \beta \otimes \alpha$$

In premonoidal categories, which are intended for modelling programming languages with effects, a symmetry is a natural ingredient, as it models swapping two values. In an L-category for an arbitrary grammar, there is no reason to expect a leftmost derivation leading from $\alpha\,\beta$ to $\beta\,\alpha$.

$$\frac{A \coloneq \alpha}{[(A,\alpha)] : A \to \alpha}$$

$$\frac{}{[\,] : \alpha \to \alpha} \qquad \frac{d_1 : \alpha \to \beta \qquad d_2 : \beta \to \gamma}{d_1 \cdot d_2 : \alpha \to \gamma}$$

$$\frac{d : \alpha \to \beta}{w \otimes d : w\,\alpha \to w\,\beta} \qquad \frac{d : \beta \to \gamma}{d \otimes \alpha : \beta\,\alpha \to \gamma\,\alpha}$$

Figure 2: Type system for leftmost derivations

$$\begin{tikzcd}
X_1 \arrow[r, "f"] & X_2
\end{tikzcd}$$

$$\begin{tikzcd}
Y_1 \arrow[d, "g"] & X_1 \otimes Y_1 \arrow[r, "f \otimes Y_1"] \arrow[d, "X_1 \otimes g"] & X_2 \otimes Y_1 \arrow[d, "X_2 \otimes g"] \\
Y_2 & X_1 \otimes Y_2 \arrow[r, "f \otimes Y_2"] & X_2 \otimes Y_2
\end{tikzcd}$$

Figure 3: A morphism f is called *central* if the square commutes for all g

Readers who prefer types to categories may refer to Figure 2, presenting the L-category construction as types for derivations. The first rule lifts grammar rules to derivations. The next two rules construct paths, and the final two rules introduce the two tensors. As the leftmost derivations are lists, we can always break a morphism down into its derivation steps. Each such step gives us a grammar rule $A \coloneq f$ together with the left context w and the right context γ in which it was applied. We will use this decomposition for giving semantics to morphisms in L-categories.

4. Semantic actions

A grammar only defines a language, in the sense of a set of strings. It does not say what those strings mean. To give meaning to the strings in the language, we need to associate a semantic action to each grammar rule, so that the meaning of a string can be constructed by parsing the string.

The language in which the semantic actions are written is a simply-typed lambda calculus with finite products. We will use this lambda calculus as a semantic meta-language, without committing to any particular model. For simply-typed lambda calculus, the set-theoretic semantics is so straightforward that we think of any semantic action

$$[\![d]\!] : [\![\beta]\!] \to [\![\alpha]\!]$$

term as just a function between sets $[\![\beta]\!]$ and $[\![\alpha]\!]$.

However, it is not necessary that the lambda calculus is pure. It could be a call-by-value lambda calculus with side effects, such as state. Such a lambda calculus could be interpreted using the Kleisli category of a monad [4] or a Freyd category [16]. The details of categorical semantics are beyond the scope of this paper, but there is one aspect that is immediately relevant here: the notion of *central* morphisms [3]. A morphism f is called central if for all g, the two ways of composing f and g given in Figure 3 are the same. Programs with effects are usually not central. For example, suppose f writes to some shared variable and g reads from it: then the order of f or g coming first is observable, and the square does not commute. When lambda abstractions are interpreted in a Freyd category via the adjunction

$$\frac{f : (X \otimes Y) \to Z}{\lambda f : X \to (Y \to Z)}$$

then the λf is always central. This reflects the situation in call-by-value or computational lambda calculi, where a lambda abstraction $\lambda x.M$ is always a value [11, 4].

For each non-terminal symbol A, we assume a type $[\![A]\!]$ for the semantic values of strings derived from that symbol.

Definition 4.1 (Semantic action for a grammar) Given a grammar $\mathcal{G}$, a semantic action $[\![-]\!]$ consists of

$$\begin{tikzcd}
\langle w, S, \lambda x.x \rangle \arrow[d, "\rightsquigarrow^*"] & \llbracket S \rrbracket \\
\langle w_1, A\,\sigma_1, k_1 \rangle \arrow[d, "\rightsquigarrow"] & \llbracket A \rrbracket \otimes \llbracket \sigma_1 \rrbracket \arrow[u, "\llbracket d_1 \rrbracket = k_1"'] \\
\langle w_1, \alpha\,\sigma_1, (\llbracket A :- \alpha \rrbracket \otimes \llbracket \sigma_1 \rrbracket) \cdot k_1 \rangle \arrow[d, "\rightsquigarrow^*"] & \llbracket \alpha \rrbracket \otimes \llbracket \sigma_1 \rrbracket \arrow[u, "\llbracket A :- \alpha \rrbracket \otimes \llbracket \sigma_1 \rrbracket"'] \\
\langle \varepsilon, \varepsilon, k_2 \rangle &
\end{tikzcd}$$

Figure 4: WSK machine move for a grammar rule $A :- \alpha$ with semantic invariant

- For each non-terminal symbols A of $\mathcal{G}$, a type $\llbracket A \rrbracket$. For a terminal symbol a, its semantic type is the unit type containing only the empty tuple:
$$\llbracket a \rrbracket \stackrel{\text{def}}{=} \mathbf{Unit} = \{()\}$$
The types are extended to strings of symbols $X_1 \dots X_n$ by taking the product of the types of the X_j:
$$\llbracket X_1 \dots X_n \rrbracket \stackrel{\text{def}}{=} \llbracket X_1 \rrbracket \otimes \cdots \otimes \llbracket X_n \rrbracket$$

- For each grammar rule $A :- \alpha$, there is a function (going in the opposite direction) of type
$$\llbracket A :- \alpha \rrbracket : \llbracket \alpha \rrbracket \longrightarrow \llbracket A \rrbracket$$

Definition 4.2 (Semantic action of a derivation) Let $\mathcal{G}$ be a grammar with a semantic action $\llbracket - \rrbracket$. For each derivation $d : \alpha \rightarrow_{\mathcal{G}} \beta$, we define the semantic action of d as morphism $\llbracket d \rrbracket : \llbracket \beta \rrbracket \longrightarrow \llbracket \alpha \rrbracket$ as follows:

- If d is the empty derivation of a string α from itself, $d = \llbracket [\,] \rrbracket : \alpha \rightarrow_{\mathcal{G}} \alpha$, then its semantic action is the identity function on the semantic of the string α:
$$\llbracket d \rrbracket \stackrel{\text{def}}{=} \mathsf{id}_{\llbracket \alpha \rrbracket}$$

- If d is not empty, we decompose it into its first derivation step using some rule $A :\!- \alpha$ and the remaining derivation d_1:

$$d = ((w\, A\, \beta, w\, \alpha\, \beta)::d_1$$

  Then $[\![\, d\,]\!]$ is defined as follows:

$$[\![\, d\,]\!] \overset{\text{def}}{=} [\![\, d_1\,]\!] \cdot (([\![\, w\,]\!]) \otimes [\![\, A :\!- \alpha\,]\!] \otimes [\![\, \beta\,]\!])$$

Note that Definition 4.2 is just the evident inductive extension of a semantic action from rules to all derivations. We could even have omitted the induction and just appealed to the initial property of the path category [13]. Also note that a derivation is already identified by the sequence of grammar rules $A :\!- \alpha$ that it uses. The reason for explicitly including the left context w and the right context β in Definition 3.1 is that it makes it easier to define the action of a derivation in Definition 4.2. The values of types $[\![\, w\,]\!]$ and $[\![\, \beta\,]\!]$ are simply carried along when we define the meaning of a grammar rule application in this context:

$$[\![\, (w\, A\, \beta, w\, \alpha\, \beta)\,]\!] : ([\![\, w\,]\!] \otimes [\![\, \alpha\,]\!] \otimes [\![\, \beta\,]\!]) \longrightarrow ([\![\, w\,]\!] \otimes [\![\, A\,]\!] \otimes [\![\, \beta\,]\!])$$

Since the semantics of terminal symbols is the unit type, $[\![\, a\,]\!] \overset{\text{def}}{=} \mathbf{Unit}$, it follows that $[\![\, w\,]\!] \cong \mathbf{Unit}$, so that $[\![\, w\,]\!] \otimes -$ contributes little to a semantics action.

Given that leftmost derivations are in one-to-one correspondence with parse trees [2], the semantics becomes easier to visualize by depicting it in a parse tree: see Figure 5. The benefit of having defined a semantic action $[\![\, d\,]\!]$ for each derivation d is this: we can now return to the abstract machines from Section 2 and extend them with a semantic component that computes the effect of all the parsing moves.

Like the SECD [17] and CEK [18] machines, the WSK machine is named for its components: a remaining input word w, a stack s, and a semantic continuation k. The latter is a continuation in the sense that it represents a function to the final answer, as the K in the CEK machine.

For the definition of the WSK machine, we will need the isomorphism of the unit type given by adding and deleting an empty tuple `()`:

$$\begin{array}{lclcccc}
\mathbf{unitleft} & = & (\lambda(\texttt{()}, x).x) & : & (\mathbf{Unit} \otimes [\![\, \sigma\,]\!]) & \longrightarrow & [\![\, \sigma\,]\!] \\
\mathbf{unitleft}^{-1} & = & (\lambda x.(\texttt{()}, x)) & : & [\![\, \sigma\,]\!] & \longrightarrow & (\mathbf{Unit} \otimes [\![\, \sigma\,]\!])
\end{array}$$

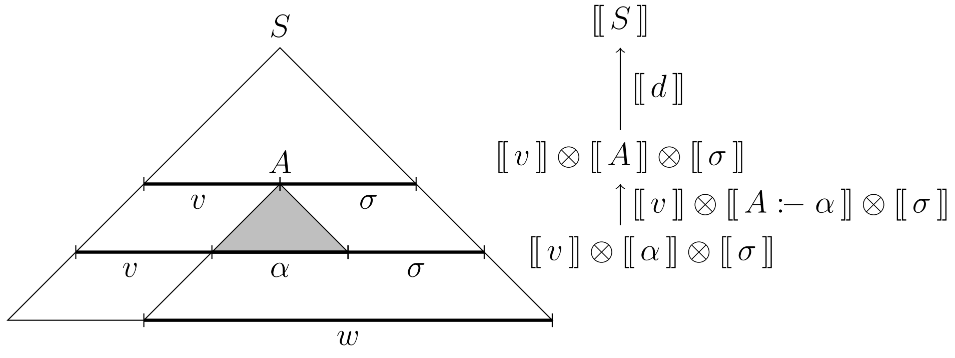
S
A
v
σ
v
α
σ
w
$[\![S]\!]$
$[\![d]\!]$
$[\![v]\!] \otimes [\![A]\!] \otimes [\![\sigma]\!]$
$[\![v]\!] \otimes [\![A :- \alpha]\!] \otimes [\![\sigma]\!]$
$[\![v]\!] \otimes [\![\alpha]\!] \otimes [\![\sigma]\!]$

Figure 5: The semantics of derivations goes from the bottom to the top of the parse tree; a derivation step $A :- \alpha$ corresponds to the shaded subtree

Definition 4.3 (WSK machine) Given a grammar $\mathcal{G}$ with a semantic action $[\![-]\!]$, the parsing machine with semantic actions is defined as follows:

- The machine has configurations of the form

  $$\langle w, \sigma, k \rangle$$

  Here w is the remaining input (a string of terminals), σ is the parsing stack (a string of grammar symbols), and k is a function.

- The initial configuration of the machine for a given input w is

  $$\langle w, S, \lambda x.x \rangle$$

  The initial parsing stack consists only of the start symbol S of $\mathcal{G}$, and the initial semantic action is the identity $\lambda x.x$.

- An accepting configuration of the machine is of the form

  $$\langle \varepsilon, \varepsilon, k \rangle$$

  with an empty input and parsing stack.

- The transitions of the machine are predicting and matching moves:

  $$\begin{array}{rcl} \langle w, A\,\sigma, k \rangle & \rightsquigarrow & \langle w, \alpha\,\sigma, ([\![A :- \alpha]\!] \otimes [\![\sigma]\!]) \cdot k \rangle \\ \langle a\,w, a\,\sigma, k \rangle & \rightsquigarrow & \langle w, \sigma, \mathbf{unitleft}^{-1} \cdot k \rangle \end{array}$$

As an example of machine transitions, consider a predict move using the rule $E :\!-\ \mathtt{1}$, followed by a match move:

$$\begin{array}{ll} & \langle \mathtt{1}\, w, E\,\sigma, k\rangle \\ \rightsquigarrow & \langle \mathtt{1}\, w, \mathtt{1}\,\sigma, ([\![\, E :\!-\ \mathtt{1}\,]\!] \otimes [\![\, \sigma \,]\!]) \cdot k\rangle \\ \rightsquigarrow & \langle w, \sigma, \mathbf{unitleft}^{-1} \cdot ([\![\, E :\!-\ \mathtt{1}\,]\!] \otimes [\![\, \sigma \,]\!]) \cdot k\rangle \end{array}$$

The predict move pushes the terminal symbol $\mathtt{1}$ onto the parsing stack, and the match moves then pops the symbol off the stack. The predict move extends the K component of the machine with the semantics $[\![\, E :\!-\ \mathtt{1}\,]\!]$ of the rule. The match move does not significantly change the K component, apart from a small adjustment of its type reflecting the change of the parsing stack.

Theorem 4.4 (WSK machine correctness) For each input string w, if the WSK machine accepts the input with final configuration $\langle \varepsilon, \varepsilon, k\rangle$, then it has constructed a leftmost derivation $d : S \to w$ and the final answer $k = [\![\, d \,]\!]$.

Proof We prove an invariant that holds for each configuration reachable from the initial configuration. The syntactic invariant is as follows: for each reachable configuration $\langle w_1, \sigma_1, k_1\rangle$, there is a leftmost derivation

$$d_1 : S \to_{\mathcal{G}} v_1\, \sigma_1$$

where v_1 represents the input that the machine has already consumed, so that $w = v_1\, w_1$. Moreover $k = [\![\, d_1 \,]\!]$. The proof is by induction on the length of d_1. The most interesting inductive step is a predict move, as depicted in Figure 4.

Note that when the machine terminates with a configuration of the form

$$\langle \varepsilon, \varepsilon, k\rangle$$

by Theorem 4.4, the type of the final action is $k : [\![\, \varepsilon \,]\!] \to [\![\, S \,]\!]$. Hence we have an element of the semantic type $[\![\, S \,]\!]$ of the start symbols as the final answer computed by the parser using the actions given by $[\![\, - \,]\!]$ for the grammar.

As discussed in Section 2, we assume that the parser generator turns the non-deterministic WSK machine for a given grammar and its action into a deterministic parser. For example, ANTLR guides the predict moves by computing appropriate lookaheads, and it implements the parsing rules as

Java methods, so that both the parsing stack σ and the semantic k are managed by the call stack of the programming language. The same holds if a recursive descent parser is written by hand.

Example 4.5 Consider the following grammar $\mathcal{G}$:

$$\begin{array}{rcl} A & :- & B\,\alpha \\ A & :- & B\,\beta \end{array}$$

Because both rules for A start with the same B, this grammar has a FIRST-/FIRST conflict, causing problems for LL parsers relying on lookahead [2]. A simple solution is to refactor the grammar using a fresh non-terminal C. The resulting grammar $\mathcal{F}$ has these rules:

$$\begin{array}{rcl} A & :- & B\,C \\ C & :- & \alpha \\ C & :- & \beta \end{array}$$

But what about the semantic actions? The new symbol C depends on some left context given by B, so we add an argument to pass in the corresponding semantic information:

$$[\![\, C \,]\!] \overset{\text{def}}{=} [\![\, B \,]\!] \to [\![\, A \,]\!]$$

For all other symbols X, we let $[\![\, X \,]\!] = [\![\, X \,]\!]$, and we write simply $[\![\, X \,]\!]$ for both. Let

$$\begin{array}{rcl} f & \overset{\text{def}}{=} & [\![\, A :- B\,\alpha \,]\!] : ([\![\, B \,]\!] \otimes [\![\, \alpha \,]\!]) \to [\![\, A \,]\!] \\ g & \overset{\text{def}}{=} & [\![\, A :- B\,\beta \,]\!] : ([\![\, B \,]\!] \otimes [\![\, \beta \,]\!]) \to [\![\, A \,]\!] \end{array}$$

We define the semantic actions $[\![\, - \,]\!]$ in the new grammar $\mathcal{F}$ as follows:

$$\begin{array}{rcl} [\![\, C :- \alpha \,]\!] & \overset{\text{def}}{=} & \lambda x.\lambda y.f(y, x) \\ & : & [\![\, \alpha \,]\!] \to ([\![\, B \,]\!] \to [\![\, A \,]\!]) \\ \\ [\![\, C :- \beta \,]\!] & \overset{\text{def}}{=} & \lambda x.\lambda y.g(y, x) \\ & : & [\![\, \beta \,]\!] \to ([\![\, B \,]\!] \to [\![\, A \,]\!]) \\ \\ [\![\, A :- B\,C \,]\!] & \overset{\text{def}}{=} & \lambda x.\lambda f.f(x) \\ & : & ([\![\, B \,]\!] \otimes (\underbrace{[\![\, B \,]\!] \to [\![\, A \,]\!]}_{[\![\, C \,]\!]})) \to [\![\, A \,]\!] \end{array}$$

Original rules for L:

$$\begin{array}{rcl} L & :- & \psi_1 \\ & \vdots & \\ L & :- & \psi_m \\ L & :- & L\,\varphi_1 \\ & \vdots & \\ L & :- & L\,\varphi_n \end{array}$$

Transformed rules:

$$\begin{array}{rcl} L & :- & \psi_1\,L' \\ & \vdots & \\ L & :- & \psi_m\,L' \\ L' & :- & \varphi_1\,L' \\ & \vdots & \\ L' & :- & \varphi_n\,L' \\ L' & :- & \varepsilon \end{array}$$

Figure 6: Transformation of grammar rules

To prove correctness of this transformation, we need to consider all leftmost derivations

$$\begin{array}{rcl} d_1 & : & A \rightarrow_{\mathcal{G}} w \\ d_2 & : & A \rightarrow_{\mathcal{F}} w \end{array}$$

and show that $[\![\, d_1 \,]\!] = [\![\, d_2 \,]\!]$.

5. Left recursion elimination

We recall the standard definition of (immediate) left-recursion elimination as presented in compiling texts [2]. Left recursion could also occur indirectly via other symbols, but that indirection can be eliminated.

Definition 5.1 (Left recursion) Given a grammar $\mathcal{G}$ and a non-terminal symbol L of $\mathcal{G}$, a non-terminal L is called *immediately left-recursive* if there is a rule of the form $L :- L\,\varphi_i$ for some string φ_i.

Left-recursion elimination for L in $\mathcal{G}$ produces a new grammar $\mathcal{F}$. Let the left-recursive rules for L in $\mathcal{G}$ be

$$L :- L\,\varphi_i$$

and let the remaining rules for L in $\mathcal{G}$ be

$$L :- \psi_j$$

Then $\mathcal{F}$ is constructed by adding a new non-terminal symbol L' and replacing the rules for L as given in Figure 6.

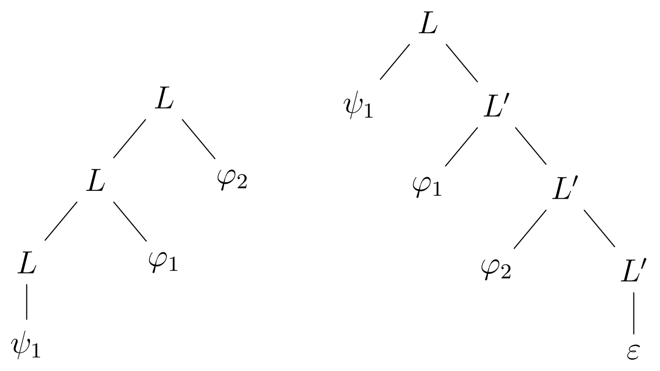

Figure 7: Parse trees for $L \to_{\mathcal{G}} \psi_1\,\varphi_1\,\varphi_2$ and $L \to_{\mathcal{F}} \psi_1\,\varphi_1\,\varphi_2$ in the original and transformed grammar

Note that since L' is chosen to be distinct from all other grammar symbols, it is at the right of the new rules and does not occur in φ_i or ψ_j, whereas L may occur anywhere in L or ψ_j, except in the left-most position of ψ_j.

The resulting grammar $\mathcal{F}$ generates the same language as the original grammar $\mathcal{G}$. The usual proof sketch for this fact points out that the rules for L' resemble the translation of Kleene star to grammars, in that L' can either terminate with the ε-rule, or go around the loop once more to generate another φ_i. One can then picture L as follows:

$$L \quad :- \quad \overbrace{(\psi_1 \mid \ldots \mid \psi_m)\;\underbrace{(\varphi_1 \mid \ldots \mid \varphi_n)^*}_{L'}}^{L}$$

This simple intuition for left-recursion elimination works for the language generated by the grammars, as the language is a set of strings, with no further structure. If we take a more intensional or fine-grained view of derivations or parse trees, the effect of the transformation is less easy to grasp. As shown in Figure 7, parse trees are rotated. If we compute the meaning of a string by way of a tree walk, as in compositional semantics, the semantics is transformed globally.

Example 5.2 The standard example of left-recursion elimination found in many compiling texts consists of expressions in arithmetic with various binary operators for addition, multiplication, subtraction, etc. For simplicity, we

$$\begin{tikzcd}
\llbracket \varphi_i \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L \rrbracket \arrow[d, "\cong"] & \llbracket \varphi_i \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[ddd, "\llbracket L' :- \varphi_i\, L' \rrbracket"] \\
(\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L \rrbracket \otimes \llbracket \varphi_i \rrbracket \arrow[d, "(\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L :- L\, \varphi_i \rrbracket"] & \\
(\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L \rrbracket \arrow[d, "\mathsf{app}"] & \\
\llbracket L \rrbracket & \llbracket L \rrbracket \to \llbracket L \rrbracket
\end{tikzcd}$$

Figure 8: From $\llbracket L :- L\, \varphi_i \rrbracket$ to $\llbracket L' :- \varphi_i\, L' \rrbracket$

$$\begin{tikzcd}
\llbracket \psi_j \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[d, "\llbracket L :- \psi_j \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket)"] & \llbracket \psi_j \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[ddd, "\llbracket L :- \psi_j\, L' \rrbracket"] \\
\llbracket L \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[d, "\cong"] & \\
(\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L \rrbracket \arrow[d, "\mathsf{app}"] & \\
\llbracket L \rrbracket & \llbracket L \rrbracket
\end{tikzcd}$$

Figure 9: From $\llbracket L :- \psi_j \rrbracket$ to $\llbracket L :- \psi_j\, L' \rrbracket$

take the subset given by a single binary operator `-` and a single constant `1`. This gives us one grammar rule with, and one without, left recursion. Let the grammar $\mathcal{G}$ be given by the following rules:

$$\begin{array}{rcl} E & :- & \mathtt{1} \\ E & :- & E - E \end{array}$$

In the notation of the left-recursion elimination construction, we have the following case:

$$L = E \qquad \psi_j = \mathtt{1} \qquad \varphi_i = \mathtt{-}\, E$$

Consequently, the construction adds a fresh non-terminal symbol E', and the rules of the transformed grammar $\mathcal{F}$ are as follows:

$$\begin{array}{rcl} E & :- & \mathtt{1}\, E' \\ E' & :- & \mathtt{-}\, E\, E' \\ E' & :- & \varepsilon \end{array}$$

When a parser generator parses a string, it does not just construct a derivation. It also performs semantic actions associated with each grammar rule. For the simple expression grammar, it is clear what these actions should be. For a rule

$$E :- E \;\mathtt{-}\; E$$

each of the occurrences of E on the right-hand side returns an integer, and the whole subtree (corresponding to the occurrence of E on the left-hand side of the rule) should return the difference of these two numbers.

Definition 5.3 (Transformation of semantic actions) Let $\mathcal{G}$ be a grammar with a semantic action $[\![-]\!]$. We assume that $\mathcal{G}$ contains left recursion for a non-teminal L. Let $\mathcal{F}$ be the grammar that is constructed from $\mathcal{G}$ by the left-recursion elimination transformation as in Definition 5.1. We define a semantic action $[\![-]\!]$ for $\mathcal{F}$ that matches how grammar rules are transformed in Figure 6.

- For the new non-terminal L', we define its semantic type as

$$[\![L']\!] \overset{\mathrm{def}}{=} [\![L]\!] \longrightarrow [\![L]\!]$$

- For a left-recursive grammar rule $L :\!- \, L\,\varphi_i$ in $\mathcal{G}$, let its action be

  $$f \stackrel{\text{def}}{=} [\![\, L :\!- \, L\,\varphi_i \,]\!] : ([\![\, L \,]\!] \otimes [\![\, \varphi_i \,]\!]) \longrightarrow [\![\, L \,]\!]$$

  As shown in Figure 8, we first construct a morphism

  $$([\![\, \varphi_i \,]\!] \otimes ([\![\, L \,]\!] \to [\![\, L \,]\!]) \otimes [\![\, L \,]\!]) \to [\![\, L \,]\!]$$

  Lambda abstraction of $[\![\, L \,]\!]$ gives us

  $$\begin{array}{rl} & [\![\, L' :\!- \, \varphi_i\, L' \,]\!] \\ : & ([\![\, \varphi_i \,]\!] \otimes ([\![\, L \,]\!] \to [\![\, L \,]\!])) \to ([\![\, L \,]\!] \to [\![\, L \,]\!]) \end{array}$$

  Equivalently, using lambda calculus syntax, we write

  $$[\![\, L' :\!- \, \varphi_i\, L' \,]\!] \stackrel{\text{def}}{=} \lambda(p,k).\lambda x.k(f(x,p))$$

  Here the types are:

  $$p : [\![\, \varphi_i \,]\!] \qquad k : [\![\, L \,]\!] \longrightarrow [\![\, L \,]\!] \qquad x : [\![\, L \,]\!]$$

- For a rule $L :\!- \, \psi_j$ in $\mathcal{G}$ that is not left-recursive, let its action be

  $$g \stackrel{\text{def}}{=} [\![\, L :\!- \, \psi_j \,]\!] : [\![\, \psi_j \,]\!] \to [\![\, L \,]\!]$$

  As shown in Figure 9, we add a parameter of type $[\![\, L \,]\!] \to [\![\, L \,]\!]$ and apply it to the result of g, which gives us a morphism

  $$\begin{array}{rl} & [\![\, L :\!- \, \psi_j\, L' \,]\!] \\ : & ([\![\, \psi_j \,]\!] \otimes ([\![\, L \,]\!] \to [\![\, L \,]\!])) \to [\![\, L \,]\!] \end{array}$$

  Writing the same with lambda terms, we have:

  $$[\![\, L :\!- \, \psi_j\, L' \,]\!] \stackrel{\text{def}}{=} \lambda(p,k).k(g(p))$$

- For the additional ε-rule for L', we take the isomorphism

  $$\mathbf{unitleft} : (\mathbf{Unit} \otimes [\![\, L \,]\!]) \to L$$

  and lambda-abstract, to give

  $$\begin{array}{rl} & [\![\, L' :\!- \, \varepsilon \,]\!] \\ : & [\![\, \varepsilon \,]\!] \to ([\![\, L \,]\!] \to [\![\, L \,]\!]) \end{array}$$

  Written as a lambda term, $[\![\, L' :\!- \, \varepsilon \,]\!] \stackrel{\text{def}}{=} (\lambda\,().(\lambda x.x))$.

- For all other non-terminals and grammar rules, the action of $\mathcal{F}$ is defined to be equal to that of $\mathcal{G}$.

Example 5.4 We continue Example 5.2 by extending it with semantic actions: we have $[\![E]\!] = \texttt{int}$ and the semantic functions as follows:

$$\begin{aligned}[\![E :\!- \texttt{1}]\!] &\overset{\text{def}}{=} \lambda(\texttt{()}).1 \\ [\![E :\!- E - E]\!] &\overset{\text{def}}{=} \lambda(x, \texttt{()}, y).x - y\end{aligned}$$

Left-recursion elimination introduces a fresh non-terminal E'. Its semantic type is:

$$[\![E']\!] \overset{\text{def}}{=} \texttt{int} \longrightarrow \texttt{int}$$

The semantic actions for the transformed grammar rules can be written as lambda-terms as follows:

$$\begin{aligned}[\![E :\!- \texttt{1}\ E']\!] &\overset{\text{def}}{=} \lambda(\texttt{()}, k).k(1) \\ &: (\textbf{Unit} \otimes (\texttt{int} \to \texttt{int})) \to \texttt{int} \\ [\![E' :\!- \texttt{-}\ E\ E']\!] &\overset{\text{def}}{=} \lambda(\texttt{()}, x, k).\lambda y.k(y - x) \\ &: (\textbf{Unit} \otimes \texttt{int} \otimes (\texttt{int} \to \texttt{int})) \to (\texttt{int} \to \texttt{int}) \\ [\![E' :\!- \varepsilon]\!] &\overset{\text{def}}{=} \lambda\texttt{()}.\lambda x.x \\ &: \textbf{Unit} \to (\texttt{int} \to \texttt{int})\end{aligned}$$

6. Correctness of the transformation

We first note that the transformed grammar simulates derivations of the original grammar. The simulation is then extended from syntax to semantics. The proof relies on the fact that the semantics $[\![-]\!]$ preserves the relevant structure, so that a diagram of derivations gives rise to a diagram of their semantics actions. Correctness amounts to showing that the resulting semantic diagram commutes.

Lemma 6.1 Let d be a derivation $d : \alpha \to_{\mathcal{F}} w$ where L' does not occur in α. Then one of the following holds:

- No rules for L or L' are used in d.

$$\begin{tikzcd}
\llbracket L \rrbracket & & \llbracket L \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "k"'] \\
\llbracket \psi_j \rrbracket \otimes \llbracket L' \rrbracket \arrow[u, "\llbracket L :- \psi_j\, L' \rrbracket"'] & & \llbracket \psi_j \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "\llbracket \psi_j \rrbracket \otimes \lambda k"'] \arrow[u, "\llbracket L :- \psi_j \rrbracket \otimes \llbracket \alpha \rrbracket"'] \\
\llbracket w \rrbracket \otimes \llbracket L' \rrbracket \arrow[u, "\llbracket d_1 \rrbracket \otimes \llbracket L' \rrbracket"'] & & \llbracket w \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "\llbracket w \rrbracket \otimes \lambda k"'] \arrow[u, "\llbracket d_1^{\triangle} \rrbracket \otimes \llbracket \alpha \rrbracket"']
\end{tikzcd}$$

Figure 10: Simulation for a $L :- \psi_j L'$ step

- The derivation d contains a derivation of a terminal string from L. Moreover, this sub-derivation is of the form $L \rightarrow_{\mathcal{F}} w\, L'$ followed by $L' :- \varepsilon$.

Consider a leftmost derivation where the next step is a rule for L:

$$w\, L\, \gamma \rightarrow_{\mathcal{F}} w\, \psi_j\, L'\, \gamma$$

The only way we can reach a string not containing L' from $w\, \psi_j\, L'\, \gamma$ is via the rule $L' :- \varepsilon$. In a leftmost derivation, all non-terminals to the left of the occurrence of L' must first have been eliminated by way of a derivation $d : \psi_j \rightarrow_{\mathcal{F}} w_2$. This gives us a derivation

$$w\, L\, \gamma \rightarrow_{\mathcal{F}} w\, \psi_j\, L'\, \gamma \rightarrow_{\mathcal{F}} w\, w_2\, L'\gamma \rightarrow_{\mathcal{F}} w\, w_2\, \gamma$$

Lemma 6.2 Derivations in $\mathcal{F}$ can be simulated by those in $\mathcal{G}$ while the semantics is preserved, in the following sense.

1. For each derivation $d : \alpha \rightarrow_{\mathcal{F}} w$ where L' does not occur in α, there is a derivation $d^{\triangle} : \alpha \rightarrow_{\mathcal{G}} w$ such that $\llbracket d^{\triangle} \rrbracket = \llbracket d \rrbracket$.
2. For each derivation $d : L \rightarrow_{\mathcal{F}} w\, L'$ there is a derivation $d^{\triangle} : L \rightarrow_{\mathcal{G}} w$ such that for all $k : \llbracket L \rrbracket \longrightarrow \llbracket L \rrbracket$, the following diagram commutes:

$$\begin{tikzcd}
\llbracket L \rrbracket & & \llbracket L \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "k"'] \\
\llbracket w_1 \rrbracket \otimes \llbracket L' \rrbracket \arrow[u, "\llbracket d_1 \rrbracket"'] & & \llbracket L \rrbracket \otimes \llbracket \varphi_i \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[u, "\llbracket L :- L\, \varphi_i \rrbracket"'] \\
\llbracket w_1 \rrbracket \otimes \llbracket \varphi_i \rrbracket \otimes \llbracket L' \rrbracket \arrow[u, "\llbracket L' :- \varphi_i\, L' \rrbracket"'] & & \llbracket w_1 \rrbracket \otimes \llbracket \varphi_i \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "\lambda k"'] \arrow[u, "\llbracket d_1^{\triangle} \rrbracket"'] \\
\llbracket w_1 \rrbracket \otimes \llbracket w_2 \rrbracket \otimes \llbracket L' \rrbracket \arrow[u, "\llbracket d_2 \rrbracket"'] & & \llbracket w_1 \rrbracket \otimes \llbracket w_2 \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "\lambda k"'] \arrow[u, "\llbracket d_2^{\triangle} \rrbracket"']
\end{tikzcd}$$

Figure 11: Simulation for a derivation d_1 followed by a $L' :- \varphi_i\, L'$ step

$$\begin{tikzcd}
\llbracket L \rrbracket & & \llbracket w \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[ll, "\llbracket d \rrbracket"'] \\
\llbracket L \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[u, "k"] & & \llbracket w \rrbracket \otimes \llbracket \alpha \rrbracket \arrow[ll, "\llbracket d^{\triangle} \rrbracket \otimes \llbracket \alpha \rrbracket"'] \arrow[u, "\llbracket w \rrbracket \otimes \lambda k"']
\end{tikzcd}$$

Proof We prove both statements of the lemma by simultaneous induction on the length of the derivation. Details are omitted, except for the three most difficult inductive steps, using rules involving L'.

1. Assume we have a derivation $d_1 : L \to_{\mathcal{F}} w\, L'$, and we extend it to a derivation d by using the rule $L' :- \varepsilon$. By the induction hypothesis, we have a derivation $d_1^{\triangle}$ making the square above commute for all k. In particular, it commutes for k being the isomorphism $\llbracket L \rrbracket \otimes$ **Unit** $\cong \llbracket L \rrbracket$. Hence the following diagram commutes by the definition of $\llbracket L' :- \varepsilon \rrbracket$ in Definition 5.3:

$$\begin{tikzcd}
\llbracket L \rrbracket & & \llbracket w \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \arrow[ll, "\llbracket d_1 \rrbracket"'] \\
\llbracket L \rrbracket \otimes \llbracket \varepsilon \rrbracket \arrow[u, "\cong"] & & \llbracket w \rrbracket \otimes \llbracket \varepsilon \rrbracket \arrow[ll, "\llbracket d_1^{\triangle} \rrbracket \otimes \llbracket \varepsilon \rrbracket"'] \arrow[u, "\llbracket w \rrbracket \otimes \llbracket L' :- \varepsilon \rrbracket"']
\end{tikzcd}$$

We define $d^{\triangle} \overset{\text{def}}{=} d_1^{\triangle}$. From the diagram, we have $\llbracket d^{\triangle} \rrbracket = \llbracket d \rrbracket$, as required.

2. Consider a derivation of the form $d : L \to_{\mathcal{F}} w\, L'$. The first step of all such derivations must be $L :- \psi_j\, L'$. This must be followed by a derivation $d_1 : \psi_j \to_{\mathcal{F}} w$ for some word w. We apply the induction hypothesis to d_1, which gives us derivation $d_1^{\triangle}$ with the same semantics, $\llbracket d_1^{\triangle} \rrbracket = \llbracket d_1 \rrbracket$. Figure 10 gives the property we need to prove for d. The right rectangle commutes because $\llbracket d^{\triangle} \rrbracket = \llbracket d \rrbracket$. The left rectangle commutes due to the construction of $L :- \psi_j\, L'$ in Figure 9).
3. Now suppose we have a derivation $d_1 : L \to w_1 L'$, and the next derivation step uses the rule $L' :- \varphi_i\, L'$. In a leftmost derivation, we must now take φ_i to some terminal string w_2 via a derivation d_2. We apply the induction hypothesis to the derivations d_1, giving $d_1^{\triangle}$ and to d_2, giving $d_2^{\triangle}$. Overall we have the following derivations:

$$\begin{tikzcd}
L \arrow[d, "d_1"] & L\,\alpha \arrow[d, "L :- L\,\varphi_i"] \\
w_1\, L' \arrow[d, "L' :- \varphi_i\, L'"] & L\,\varphi_i\,\alpha \arrow[d, "d_1^{\triangle}"] \\
w_1\,\varphi_i\, L' \arrow[d, "d_2"] & w_1\,\varphi_i\,\alpha \arrow[d, "d_2^{\triangle}"] \\
w_1\, w_2\, L' & w\, w_2\,\alpha
\end{tikzcd}$$

The property we need to prove is given in Figure 11. By the induction hypothesis, $\llbracket d_2^{\triangle} \rrbracket = \llbracket d_2 \rrbracket$. The diagram commutes because

$$\lambda((\llbracket L :- L\,\varphi_i \rrbracket) \otimes \llbracket \alpha \rrbracket) \cdot k) = \llbracket L' :- \varphi_i\, L' \rrbracket \cdot (\llbracket \varphi_i \rrbracket \otimes (\lambda k))$$

and $\llbracket L' :- \varphi_i\, L' \rrbracket$ is central due to its definition via λ (see Figure 8).

Given Lemma 6.2, we have the desired result for complete derivations, which correspond to accepting computations of the WSK machine.

Theorem 6.3 Derivations $S \to_{\mathcal{F}} w$ in $\mathcal{F}$ (after left-recursion elimination) can be simulated by those in the original grammar $\mathcal{G}$ such that the semantics is preserved.

The converse of Theorem 6.3 does not hold. There could be derivations in the original grammar that do not correspond to any after transformation. This is due to the fact that the transformation can reduce the ambiguity of the grammar. For example, the grammar in Example 5.2 can parse `1 - 1 - 1` in two different ways, giving two different parse trees, or equivalently, two different leftmost derivations. But removing the ambiguity is beneficial [2]; after all, we would not want a compiler to choose randomly how to evaluate expressions.

7. Relation to continuation passing style

If one thinks of continuations in terms of control operators such as `call/cc` in Scheme, it may be surprising to find them in left-recursion elimination. In this section, some parallels to continuations, as expressed via transformations on lambda calculi, are drawn.

7.1. Left recursion elimination and continuations

Consider Example 5.4 again, and compare it to a CPS transform for the semantics of the original grammar. An expression M is transformed into $\overline{M}$ as follows:

$$\begin{array}{rcl} \overline{1} & = & \lambda k.k\,1 \\ \overline{M_1 - M_2} & = & \lambda k.\overline{M_1}\,(\lambda x.\overline{M_2}(\lambda y.k(x-y))) \end{array}$$

For each transformed M, its type is

$$\overline{M} : (\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$$

To evaluate a term in CPS to a number, we supply the identity as the top-level continuation $\overline{M}\,(\lambda x.x)$.

Reading Example 5.4 in continuation terms, note that the semantics of the fresh non-terminal E' is that of continuations for $[\![\, E \,]\!] = \mathtt{int}$:

$$[\![\, E' \,]\!] \;=\; \mathtt{int} \longrightarrow \mathtt{int}$$

Up to some uncurrying and reordering of parameters, the above CPS transform gives us essentially the semantic actions for the transformed grammar:

$$\begin{aligned}
[\![\, E :- \texttt{1}\; E' \,]\!] &= \lambda(\,(),k).k(1) \\
[\![\, E' :- \texttt{-}\; E\; E' \,]\!] &= \lambda(\,(),x,k).\lambda y.k(y-x) \\
[\![\, E' :- \varepsilon \,]\!] &= \lambda\,().\lambda x.x
\end{aligned}$$

For the last rule, we know that the ε-rule is applied at the end of a subderivation when we are done with expanding the E' and switch back to the original grammar by deleting E'. At this point in the derivation, the continuation-passing style ends, which is a form of control delimiter. A variety of such control delimiters, with subtle differences, have been defined and investigated, for instance by Danvy and Filinksi [19] and Felleisen et. al. [20]. A common feature is that the identity $\lambda x.x$ is supplied when expressions in continuation-passing style are interfaced with those in direct style.

More generally, we revisit the rotation of parse trees from Figure 7 in terms of semantic actions in Figure 12. Here the actions of the original rules for L are assumed to be g_1, f_1, and f_2. To compensate for the rotation, the transformed semantic actions take a parameter k that works as a continuation. Corresponding to the syntactic rotation of the parse tree, on the semantic actions we have the “inside-out” composition typical of continuation passing. In the first tree, f_2 is added to the top of the tree, in direct style. In the second tree, it is added near the bottom, in the continuation. Since the remainder of the tree will apply its continuation k in tail position, the two ways of adding f_2 are equivalent. The continuations move semantic actions around the tree, but they are not used for control effects like copying or discarding the current continuation: they are *linearly used* [21].

The equations that we need to prove correctness of the semantic transform lend themselves to answer type polymorphism [22]. Recall the transformation of semantic actions from Definition 5.3. For an action in the original grammar,

$$f = [\![\, L :- L\, \varphi_i \,]\!]$$

the transformed grammar has the action

$$[\![\, L' :- \varphi_i\, L' \,]\!] = \lambda(p,k).\lambda x.k(f(x,p))$$

So far, we have typed this term with $k : [\![\, L \,]\!] \longrightarrow [\![\, L \,]\!]$. But the answer type of the continuation need not be $[\![\, L \,]\!]$. It could be a type variable χ. We give

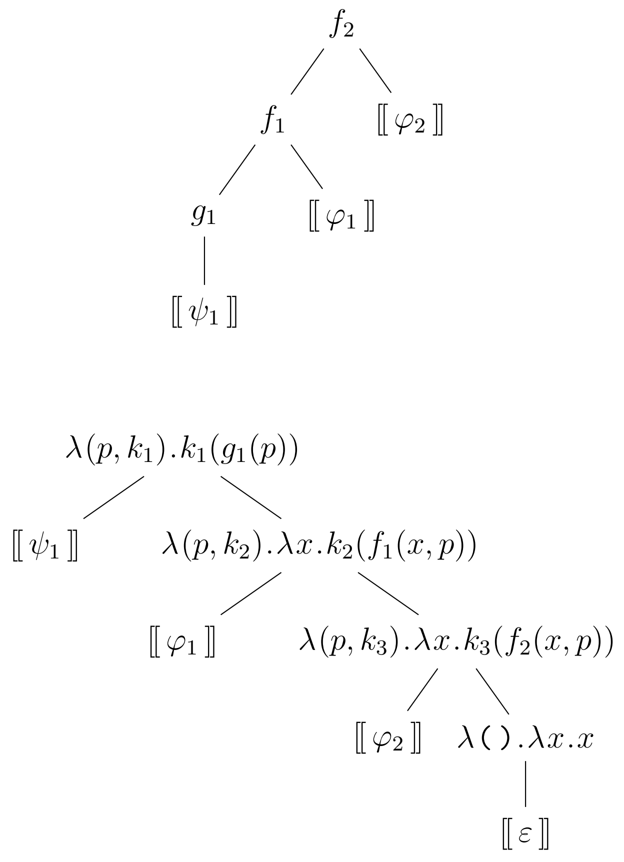
f_2
f_1
$[\![\, \varphi_2 \,]\!]$
g_1
$[\![\, \varphi_1 \,]\!]$
$[\![\, \psi_1 \,]\!]$
$\lambda(p, k_1).k_1(g_1(p))$
$[\![\, \psi_1 \,]\!]$
$\lambda(p, k_2).\lambda x.k_2(f_1(x, p))$
$[\![\, \varphi_1 \,]\!]$
$\lambda(p, k_3).\lambda x.k_3(f_2(x, p))$
$[\![\, \varphi_2 \,]\!]$
$\lambda(\,).\lambda x.x$
$[\![\, \varepsilon \,]\!]$

Figure 12: Semantic actions for original and transformed parse trees for $\psi_1\ \varphi_1\ \varphi_2$

the term the following polymorphic type that is parametric in the answer type χ:

$$\lambda(p, k).\lambda x.k(f(x, p)) : \forall\chi.([\![\, \varphi_i \,]\!] \otimes ([\![\, L \,]\!] \to \chi)) \to \chi$$

Just from the type, we can infer that the action satisfies certain equations by using parametricity reasoning in the style that Wadler calls "Theorems for free" [23]. Developing answer type parametricity for the transformation is beyond the scope of the present paper, but it could give an alternative proof technique to the diagram chases in Figures 10 and 11. In continuation terms, the commuting diagrams in the simulation of transformed derivations can be explained as the relation between a direct-style program and its continuation passing equivalent. Supplying a continuation k as a value (via the λk on the right) to the continuation passing program is the same as post-composing k to the program in direct style (via the k on the left of the diagram).

7.2. The left-corner transformation and continuation passing

Another technique for eliminating left recursion from a grammar is given by the left-corner transformation [24], which yields more compact grammars for natural-language grammars than the transform classically used in compiler texts. The left-corner transformation has been implemented and investigated by several researchers in the context of parsing combinators in functional programming [25, 26, 27].

In brief, the left-corner transform introduces new grammar symbols, written as $(C{-}X)$, where C is a non-terminal symbol, and X is a grammar symbol. New rules are introduced using these symbols, as follows:

1. the rule $A :\!-\; a\,(A{-}a)$ for any non-terminal A and terminal a;
2. the rule $(C{-}X) :\!-\; \beta\,(C{-}A)$ whenever there is a rule $A :\!-\; X\,\beta$ in the old grammar;
3. the rule $(A{-}A) :\!-\; \varepsilon$.

To see the connection to the classic left-recursion elimination transformation, it may help to think of its new grammar symbol L' as $(L{-}L)$ in the notation of the left-corner transformation. The first rule resembles $L\;:\!-\;\psi_j\,L'$, which coincides with it when $\psi_j = a$. The second case generalizes $L' :\!-\; \beta\,L'$. The third is essentially the ε-production $L' :\!-\; \varepsilon$.

It may be conjectured that these new rules can again be reconstructed as continuation passing style analogues of the old rules, in parallel to our view of $L' :\!-\; \beta\,L'$ as a continuation passing version of $L :\!-\; L\,\beta$. Following Brink, Holdermans and Löh [27], we define the semantic types of the new symbols $(C{-}X)$ as

$$[\![\,(C{-}X)\,]\!] \overset{\text{def}}{=} [\![\,X\,]\!] \to [\![\,C\,]\!]$$

Here $[\![\,C\,]\!]$ can be seen as an answer type of continuations. The left-corner transform is more complex than the classic left-recursion elimination in that it employs multiple local answer types. Each of the new symbols $(C{-}X)$ has the answer type $[\![\,C\,]\!]$, as opposed to a single new symbol L' with a global answer type $[\![\,L\,]\!]$. As in Definition 5.3, the semantics of a rule

$$(C{-}X) :\!-\; \beta\,(C{-}A)$$

has the type

$$(\llbracket \beta \rrbracket \otimes (\llbracket A \rrbracket \to \llbracket C \rrbracket)) \to (\llbracket X \rrbracket \to \llbracket C \rrbracket)$$

The same proof technique as in Lemma 6.2, suitably adapted for the multiple answer types, may be applicable to proving the correctness of the transformation of the semantic actions [27] in the left-corner transformation.

8. An idealized L-attribute parsing calculus

So far, we have used lambda calculus for defining the semantic actions of grammar rules. In this section, we define a language that corresponds more closely to the input language of parser generators such as ANTLR [28, 7]. Specifically, the language lets one define a parsing function for each non-terminal of the grammar, such that parsing functions take parameters and have return values.

A term N in the L-calculus can be of the following forms:

- $\texttt{let}\, x = B(P); N_1$ calls the parsing function for the non-terminal B with parameter P and then binds the result of the call to x in N_1.
- $\ulcorner a \urcorner; N_1$ matches the terminal symbol a in the input and then proceeds with N_1.
- The parameters and return values of parsing functions are given by terms P that evaluate to some value V, without directly affecting the parsing.

Each non-terminal A gives rise to a parsing function with parameters of type $\langle\!\langle A \rrbracket$ and return type $\llbracket A \rangle\!\rangle$. Calling such a function causes some input word $w \in \mathcal{L}(A)$ to be consumed and parsed. We will regard this reading of input as a *computational effect*, in the sense of monads as notions of computation [4] and effect systems [29]. For instance, a term N may have some type τ, but in addition it may have the effect of calling some parsing functions, say B and then C. We write its type and effect in a judgement of the form:

$$\Gamma \Vdash N : \tau \,!\, B\, C$$

The parsing effects are substructural, in the sense of substructural logics like linear [30] or separation logic [31, 32]. The effect $B\,C$ in the above judgement for N is to be distinguished from that in the following judgements:

$$\begin{array}{lcl} \Gamma & \Vdash & N_1 : \tau \; ! \; C \\ \Gamma & \Vdash & N_2 : \tau \; ! \; B\,B\,C \\ \Gamma & \Vdash & N_3 : \tau \; ! \; C\,B \end{array}$$

Calling B twice is not the same as calling it only once or not at all (so the effect is linear). Moreover, the order in which B and C are called matters as well, so that the effect system does not even permit the exchange rule, just like Lambek's syntactic calculus [15].

When we have a judgement $\Gamma \Vdash N : \tau \; ! \; B\,C$, the parameters of the call of C in N may depend on the result of the call to B, but not conversely. This left-to-right-dependency corresponds to L-attribute grammars [28].

The L-calculus has two kinds of terms: pure terms P may refer to temporary variables and may perform calculations on them, whereas general terms N may also have parsing effects. The typing judgements of pure terms are those of a very simple functional language, for example

$$x : \texttt{int}, y : \texttt{int} \vdash (x - y) : \texttt{int}$$

Strictly speaking, the L-calculus is parametric in the choice of a basic language of pure terms P. Here only a minimal set of constructs, as sufficient for the examples, will be considered.

Definition 8.1 (L-calculus syntax and type system) The syntax of terms N, pure terms P and values V is given by the grammar:

$$\begin{array}{lcl} N & ::= & \texttt{let}\, x = B(P); N \mid \ulcorner a \urcorner;\, N \mid P \\ P & ::= & x \mid P - P \mid (P, P) \mid n \\ V & ::= & n \mid (V, V) \end{array}$$

Types are given by this grammar:

$$\tau ::= \texttt{int} \mid \tau \otimes \tau \mid \tau_1 \to \tau_2$$

Terms N with syntactic effects may call parsing functions corresponding to grammar symbols. The typing judgements for such terms are of the form

$$\Gamma \Vdash N : \tau \; ! \; \alpha$$

$\boxed{\Gamma \vdash P : \tau}$

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \qquad \frac{\Gamma \vdash P_1 : \texttt{int} \qquad \Gamma \vdash P_2 : \texttt{int}}{\Gamma \vdash (P_1 - P_2) : \texttt{int}}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash P_1 : \tau_1 \qquad \Gamma \vdash P_2 : \tau_2}{\Gamma \vdash (P_1, P_2) : \tau_1 \otimes \tau_2}$$

$$\frac{\Gamma \vdash P_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash P_2 : \tau_1}{\Gamma \vdash P_1(P_2) : \tau_2}$$

$\boxed{\Gamma \Vdash N : \tau \,!\, \alpha}$

$$\frac{\Gamma \vdash P : \langle\!\langle\, B \,]\!] \qquad \Gamma, x : [\![\, B \,\rangle\!\rangle \Vdash N : \tau \,!\, \alpha}{\Gamma \Vdash (\texttt{let}\, x = B(P); N) : \tau \,!\, B\,\alpha}$$

$$\frac{\Gamma \vdash P : \tau}{\Gamma \Vdash P : \tau \,!\, \varepsilon} \qquad \frac{\Gamma \Vdash N : \tau \,!\, \alpha}{\Gamma \Vdash (\ulcorner b \urcorner; N) : \tau \,!\, b\,\alpha}$$

Figure 13: L-calculus typing rules for pure terms P and effectful terms N

The typing rules for the L-calculus are given in Figure 13. The context Γ types temporary variables. The parsing effect α records which parsing functions are called inside the term. A pure term P can be lifted to an effectful term with the empty string ε as its parsing effect.

A *program* in the L-calculus consists of:

1. A signature for the non-terminals of the grammar, giving for each A its domain $\langle\!\langle\, A \,]\!]$ and codomain $[\![\, A \,\rangle\!\rangle$.
2. A typing context Γ_A for each non-terminal A.
3. A set of augmented grammar rules of the form $\langle\!\langle\, A :\!-\, \alpha \,\rangle\!\rangle = M$, giving an L-calculus term M as the semantics for a grammar rule $A :\!-\, \alpha$.

Definition 8.2 A program is *well formed* if for each augmented grammar rule of the form

$$\langle\!\langle\, A :\!-\, \alpha \,\rangle\!\rangle = N$$

there are variables $x_1, \ldots, x_n$ and types $\tau_1, \ldots, \tau_n$ such that:

$$\begin{array}{rcl} \Gamma_A & \equiv & x_1 : \tau_1, \ldots, x_n : \tau_n \\ \langle\!\langle\, A \,]\!] & = & \tau_1 \otimes \ldots \otimes \tau_n \\ \Gamma_A & \Vdash & N : [\![\, A \,\rangle\!\rangle \,!\, \alpha \end{array}$$

As an example of a program in the L-calculus, we revisit the running example 5.2 of a fragment of arithmetic expressions. To illustrate the passing of parameters to parsing functions, we add another grammar rule for variables; for simplicity we assume that there is just a single terminal symbol `a` representing variables. The interesting point about adding variables to the program is that they require environments e to be passed to the parsing function for E. Environments are functions from variables to integers, although they could equally be modelled as a non-functional data structure, such as a list of pairs.

Example 8.3 (Expressions with environments in the L-calculus)

$$\begin{array}{rcl} \langle\!\langle\, E \,]\!] & = & \texttt{env} \\ [\![\, E \,\rangle\!\rangle & = & \texttt{int} \end{array}$$

$$\Gamma_E \;=\; e : \texttt{env}$$

$$\begin{array}{rcl} \langle\!\langle\, E :\!-\; E \;\texttt{-}\; E \,\rangle\!\rangle & = & \texttt{let}\, x = E(e); \ulcorner\texttt{-}\urcorner; \texttt{let}\, y = E(e); x - y \\ \langle\!\langle\, E :\!-\; \texttt{1} \,\rangle\!\rangle & = & \ulcorner\texttt{1}\urcorner; 1 \\ \langle\!\langle\, E :\!-\; \texttt{(}\, E \,\texttt{)} \,\rangle\!\rangle & = & \ulcorner\texttt{(}\urcorner; \texttt{let}\, x = E(e); \ulcorner\texttt{)}\urcorner; x \\ \langle\!\langle\, E :\!-\; \texttt{a} \,\rangle\!\rangle & = & \ulcorner\texttt{a}\urcorner;\; e(\texttt{a}) \end{array}$$

In ANTLR, Example 8.3 corresponds to a single parsing function for E, where the rules correspond to the alternatives in the body of the parsing function. The parameter name and type, as well as the return type `int`, are given in ANTLR by the explicitly-typed syntax for parsing functions.

The basic grammar, without the semantics of the parsing functions, is given by the parsing effects of the parsing functions. For instance, the type and effect judgement

$$e : \texttt{env} \Vdash \texttt{let}\, x = E(e); \ulcorner\texttt{-}\urcorner; \texttt{let}\, y = E(e); x - y : \texttt{int} \,!\, E \;\texttt{-}\; E$$

corresponds to the grammar rule $E :\!-\; E$ `-` E.

$$\boxed{\Gamma_1 \Vdash \mathcal{L} : \Gamma_2 \,!\, \alpha}$$

$$\frac{}{\Gamma_1 \Vdash \bigcirc : - \,!\, \varepsilon} \qquad \frac{\Gamma_1 \Vdash \mathcal{L} : \Gamma_2 \,!\, \alpha}{\Gamma_1 \Vdash (\ulcorner b \urcorner; \mathcal{L}) : \Gamma_2 \,!\, b\,\alpha}$$

$$\frac{\Gamma_1 \Vdash \mathcal{L} : \Gamma_2 \,!\, \alpha \qquad \Gamma_1 \vdash P : \langle\!\langle\, B \,]\!]}{\Gamma_1 \Vdash (\texttt{let}\, x = B(P); \mathcal{L}) : \Gamma_2, x : [\![\, B \,\rangle\!\rangle \,!\, B\,\alpha}$$

Figure 14: Typing rules for L-calculus contexts $\mathcal{L}$

9. Transformation of L actions

In order to adapt the semantic left-recursion elimination transform, it will be useful to have a notion of left context in the L-calculus. The idea of using one-hole contexts as a syntactic representation of continuations is due to Felleisen [18].

Definition 9.1 Left contexts $\mathcal{L}$ for the L-calculus are defined as:

$$\begin{array}{rcl} \mathcal{L} & ::= & \bigcirc \\ & | & \ulcorner a \urcorner; \mathcal{L} \\ & | & \texttt{let}\, x = B(P); \mathcal{L} \end{array}$$

The $\bigcirc$ represents the hole in the context into which a term may be plugged. We write $\mathcal{L}$; N for the term that is the result of plugging N into the hole in $\mathcal{L}$. A context can bind variables in the term plugged into it. To keep track of the types, we introduce typing judgements for contexts in Figure 14. In a judgement

$$\Gamma_1 \Vdash \mathcal{L} : \Gamma_2 \,!\, \alpha$$

the first Γ_1 contains the free variables the context may depend on, whereas Γ_2 types the variables it binds. As for the L-calculus, α stands for the parsing effect of the context.

Lemma 9.2 (Left context decomposition) There is a one-to-one correspondence between:

- terms N of the L-calculus such that

$$\Gamma \Vdash N : \tau \,!\, \alpha_1\,\alpha_2$$

- contexts $\mathcal{L}$ together with terms N_1 such that $N \equiv \mathcal{L};\ N_1$ and

$$\Gamma \Vdash \mathcal{L} : \Gamma_2 \;!\; \alpha_1 \text{ and } \Gamma, \Gamma_2 \Vdash N_1 : \tau \;!\; \alpha_2$$

Proof Both properties follow from a straightforward induction on the length of α_1.

Given the decomposition of a term, we can now define the semantic transformation for left-recursion elimination. For simplicity, we assume that Γ_L contains only a single variable d, so that $\Gamma_L = d : \langle\!\langle\, L \,]\!]$.

Definition 9.3 (Left-recursion elimination for the L-calculus) The domain and codomain of the fresh grammar symbol L' are defined as follows:

$$\begin{aligned} \langle\!\langle\, L' \,]\!] &\overset{\text{def}}{=} \langle\!\langle\, L \,]\!] \otimes [\![\, L \,\rangle\!\rangle \\ [\![\, L' \,\rangle\!\rangle &\overset{\text{def}}{=} [\![\, L \,\rangle\!\rangle \end{aligned}$$

The typing context $\Gamma_{L'}$ is defined by extending that for L with a fresh variable c:

$$\Gamma_{L'} \overset{\text{def}}{=} d : \langle\!\langle\, L \,]\!], c : [\![\, L \,\rangle\!\rangle$$

The new augmented rules involving L' as defined in terms of those for L in the original grammar.

1. Let $\langle\!\langle\, L :\!-\, L\,\varphi_i \,\rangle\!\rangle = N$ and decompose $N \equiv (\mathtt{let}\, c = L(P_1); \mathcal{L}_{\varphi_i};\ P_2)$. Then we define the semantics of the transformed rule as follows:

$$\langle\!\langle\, L' :\!-\, \varphi_i\, L' \,\rangle\!\rangle \overset{\text{def}}{=} (\mathcal{L}_{\varphi_i};\ \mathtt{let}\, z = L'(P_1, P_2); z)$$

   where z is a fresh variable.
2. Let $\langle\!\langle\, L :\!-\, \psi_j \,\rangle\!\rangle = N$ and decompose $N \equiv \mathcal{L}_{\psi_j};\ P$. Then we define:

$$\langle\!\langle\, L :\!-\, \psi_j\, L' \,\rangle\!\rangle \overset{\text{def}}{=} (\mathcal{L}_{\psi_j};\ \mathtt{let}\, z = L'(d, P); z)$$

3. For the ε-rule $L' :\!-\, \varepsilon$, we define

$$\langle\!\langle\, L' :\!-\, \varepsilon \,\rangle\!\rangle \overset{\text{def}}{=} c$$

The variable d stands for domain $\langle\!\langle\, L \,]\!]$ of the parsing function for L. The variable c stands for the codomain $[\![\, L \,\rangle\!\rangle$.

The presence of continuation passing in Definition 9.3 is less evident than in Section 7, but still discernible. The new symbol L' in the rightmost position acts as a linearly-used continuation. The continuation is inlined statically, rather than being passed into the parsing function at runtime.

The transformation in Definition 9.3 is well typed: if the original rules for L are well-typed in the sense of Definition 8.2, then so are the new rules generated from them. Thus the transformation is not just between L-calculus terms N, but between typing judgements $\Gamma \Vdash N : \tau \mathrel{!} \alpha$.

Lemma 9.4 (Typing of left-recursion elimination) The new rules for L' are well-typed:

1. When $L :\!- L\,\varphi_i$ is transformed into $L' :\!- \varphi_i\,L'$ then the judgement

$$d : \langle\!\langle\, L \,]\!] \Vdash \mathtt{let}\, c = L(P_1); \mathcal{L}_{\varphi_i};\ P_2 : [\![\, L \,\rangle\!\rangle \mathrel{!} L\,\varphi_i$$

is transformed into

$$d : \langle\!\langle\, L \,]\!], c : [\![\, L \,\rangle\!\rangle \Vdash \mathcal{L}_{\varphi_i}; \mathtt{let}\, z = L'(P_1, P_2); z : [\![\, L \,\rangle\!\rangle \mathrel{!} \varphi_i\,L'$$

2. When $L :\!- \psi_j$ is transformed into $L :\!- \psi_j\,L'$ then the judgement

$$d : \langle\!\langle\, L \,]\!] \Vdash \mathcal{L}_{\psi_j};\ P : [\![\, L \,\rangle\!\rangle \mathrel{!} \psi_j$$

is transformed into

$$d : \langle\!\langle\, L \,]\!] \Vdash \mathcal{L}_{\psi_j}; \mathtt{let}\, z = L'(d, P); z : [\![\, L \,\rangle\!\rangle \mathrel{!} \psi_j\,L'$$

3. For the ε-rule $L' :\!- \varepsilon$, we have the judgement

$$d : \langle\!\langle\, L \,]\!], c : [\![\, L \,\rangle\!\rangle \Vdash c : [\![\, L \,\rangle\!\rangle \mathrel{!} \varepsilon$$

Example 9.5 (Transformation of expressions in the L-calculus) We apply the transformation from Definition 9.3 to Example 8.3.

Consider the rule

$$\langle\!\langle\, E :\!- E\ \mathtt{-}\ E \,\rangle\!\rangle = \mathtt{let}\, x = E(e); \ulcorner\mathtt{-}\urcorner; \mathtt{let}\, y = E(e); x - y$$

We decompose it as follows:

$$\mathtt{let}\, x = E(P_1); \mathcal{L};\ P_2$$

where

$$\begin{array}{rcl}
\mathcal{L} & \equiv & \ulcorner\texttt{-}\urcorner;\, \texttt{let}\, y = E(e);\, \bigcirc \\
P_1 & \equiv & e \\
P_2 & \equiv & x - y \\
L & \equiv & E
\end{array}$$

Then the transformed semantics for $E' :\!-\ \texttt{-}\ E\ E'$ is:

$$\begin{array}{rl}
& \mathcal{L};\ \texttt{let}\, z = L'(P_1, P_2);\, z \\
\equiv & \mathcal{L};\ \texttt{let}\, z = E'(e, x - y);\, z \\
\equiv & \ulcorner\texttt{-}\urcorner;\, \texttt{let}\, y = E(e);\, \texttt{let}\, z = E'(e, x - y);\, z
\end{array}$$

The transformed grammar is as follows:

$$\begin{array}{rcl}
\langle\!\langle\, E \,]\!] & = & \texttt{env} \\
[\![\, E \,\rangle\!\rangle & = & \texttt{int} \\
\Gamma_E & = & e : \texttt{env}
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle\, E :\!-\ \texttt{1}\ E' \,\rangle\!\rangle & = & \ulcorner\texttt{1}\urcorner;\, \texttt{let}\, z = E'(e, 1);\, z \\
\langle\!\langle\, E :\!-\ \texttt{a}\ E' \,\rangle\!\rangle & = & \ulcorner\texttt{a}\urcorner;\, \texttt{let}\, z = E'(e, e(\texttt{a}));\, z \\
\langle\!\langle\, E :\!-\ \texttt{(}\ E\ \texttt{)}\ E' \,\rangle\!\rangle & = & \ulcorner\texttt{(}\urcorner;\, \texttt{let}\, x = E(e);\, \ulcorner\texttt{)}\urcorner;\, \texttt{let}\, z = E'(e, x);\, z
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle\, E' \,]\!] & = & \texttt{env} \otimes \texttt{int} \\
[\![\, E' \,\rangle\!\rangle & = & \texttt{int} \\
\Gamma_{E'} & = & e : \texttt{env}, x : \texttt{int}
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle\, E' :\!-\ \texttt{-}\ E\ E' \,\rangle\!\rangle & = & \ulcorner\texttt{-}\urcorner;\, \texttt{let}\, y = E(e);\, \texttt{let}\, z = E'(e, x - y);\, z \\
\langle\!\langle\, E' :\!-\ \varepsilon \,\rangle\!\rangle & = & x
\end{array}$$

A detailed proof of the correctness of the transformation in Definition 9.3 is beyond the scope of the paper. However, it is not necessary to build such a proof from scratch, as results above can be re-used. In brief, one can define a semantics $\langle\!\langle\, N \,\rangle\!\rangle$ of L-calculus along the same lines as the semantics

$[\![\, M \,]\!]$ in Definition 4.2. The only significant difference is the translation of the new grammar symbol L', while for all other symbols, the two semantics coincide. In the semantics $[\![\, - \,]\!]$, the type $[\![\, L' \,]\!]$ may be a higher-order function, whereas in $\langle\!\langle\, L' \,\rangle\!\rangle$, such higher-order functions are avoided, and only additional arguments are added to $\langle\!\langle\, L \,\rangle\!\rangle$. For example, consider a grammar with expressions where parsing functions take an environment parameter of type `env`, as in Example 9.5.

$$\begin{array}{ll} & [\![\, L' \,]\!] \\ = & [\![\, L \,]\!] \to [\![\, L \,]\!] \\ = & (\langle\!\langle\, L \,]\!] \to [\![\, L \,\rangle\!\rangle) \to (\langle\!\langle\, L \,]\!] \to [\![\, L \,\rangle\!\rangle) \\ = & (\mathtt{env} \to \mathtt{int}) \to (\mathtt{env} \to \mathtt{int}) \end{array}$$

$$\begin{array}{ll} & \langle\!\langle\, L' \,\rangle\!\rangle \\ = & (\langle\!\langle\, L \,]\!] \otimes [\![\, L \,\rangle\!\rangle) \to [\![\, L \,\rangle\!\rangle \\ = & (\mathtt{env} \otimes \mathtt{int}) \to \mathtt{int} \end{array}$$

We can define a conversion function ℓ:

$$\begin{array}{lcl} \ell & : & [\![\, L' \,]\!] \to \langle\!\langle\, L' \,\rangle\!\rangle \\ \ell & = & \lambda f.\lambda(x,y).f\,(\lambda z.y)\,x \end{array}$$

The conversion between the two semantics may be easiest to grasp by considering the new ε-rule introduced by the left-recursion elimination. Its semantics is the identity and the second projection, respectively.

$$\begin{array}{rcl} [\![\, L' :\!- \varepsilon \,]\!] & = & \mathsf{id} : \langle\!\langle\, L \,]\!] \to [\![\, L \,\rangle\!\rangle) \to (\langle\!\langle\, L \,]\!] \to [\![\, L \,\rangle\!\rangle) \\ \langle\!\langle\, L' :\!- \varepsilon \,\rangle\!\rangle & = & \pi_2 : (\langle\!\langle\, L \,]\!] \otimes [\![\, L \,\rangle\!\rangle) \to [\![\, L \,\rangle\!\rangle \end{array}$$

Then ℓ connects the two semantic function, since $\ell(\mathsf{id}) = \pi_2$.

Given the conversion ℓ, the correctness of the semantic transformation for $\langle\!\langle\, N \,\rangle\!\rangle$ follows from Theorem 6.3 for the transformation of $[\![\, M \,]\!]$ from Definition 5.3.

10. An abstract machine for the L-calculus

To give meaning to the L-calculus, we define an abstract machine for it, in the style of the SECD machine [17] or the CEK machine [18]. The

L-calculus is more restricted than a lambda calculus, corresponding to the stylized language supported by parser generators in which each non-terminal corresponds to a parsing function. The generated parser may for instance consist of C functions, so that one cannot assume first-class functions or closures as given.

The L^2 machine interprets a language without first-class lambda abstractions, so allocating closures on a heap can be avoided. The only closures that need to be dynamically allocated are continuation closures, which can be kept on a stack [33]. Even if heap-allocated closures are supported, it can be advantageous to stack-allocate closures whenever possible, as in Leroy's Zinc machine for CAML [34]. As in the SECD and CEK machines, a central feature of the L^2 machine is a stack of frames. Each frame corresponds to a call site in one of the parsing functions. The parameter of the frame is the return value of the function call.

The domain $\langle\!\langle\, S \,]\!]$ of the start symbol S is assumed to be the unit type, so that the machine can always start by calling the parsing function with an empty parameter $()$.

Definition 10.1 (L^2 machine) The configurations of the L^2 machine are 3-tuples written as

$$\langle w, N, s \rangle$$

Here w is a terminal string, N is a term of the L-calculus, and s is a stack. Stacks are lists of continuation frames of the form $(\curlywedge x.N)$ separated by $\cdot$. The empty stack, causing the machine to stop, is written as $\blacksquare$. So stacks are defined inductively as:

$$s \;::=\; \blacksquare \mid (\curlywedge x.N) \cdot s$$

The transitions of the L^2 machine are given in Figure 15.

We assume that there is a (big-step) semantics of the form $P \Downarrow V$ for evaluating pure terms P to some value V. For the fragment of arithmetical expressions, the big-step rules are:

$$\frac{P_1 \Downarrow n_1 \qquad P_2 \Downarrow n_2}{P_1 - P_2 \Downarrow n_1 - n_2} \qquad \frac{P_1 \Downarrow n_1 \qquad P_2 \Downarrow n_2}{(P_1, P_2) \Downarrow (n_1, n_2)} \qquad \frac{}{n \Downarrow n}$$

For a given input string w, the initial configuration of the machine is

$$\langle w, \texttt{let}\, x = S(); x, \blacksquare \rangle$$

$$\begin{array}{rcl}
\langle w, \mathtt{let}\, x = B(V); N, s\rangle & \leadsto & \langle w, N_2[y \mapsto V], (\measuredangle x.N)\cdot s\rangle \\
 & \text{if} & \langle w, B\,\sigma\rangle \leadsto \langle w, \alpha\,\sigma\rangle \\
 & \text{and} & \langle\!\langle\, B :\!-\ \alpha \,\rangle\!\rangle = (\measuredangle y.N_2) \\[1em]
\langle a\,w, \ulcorner a \urcorner; N, s\rangle & \leadsto & \langle w, N, s\rangle \\
 & \text{if} & \langle a\,w, a\,\sigma\rangle \leadsto \langle w, \sigma\rangle \\[1em]
\langle w, \mathtt{let}\, x = B(P); N, s\rangle & \leadsto & \langle w, \mathtt{let}\, x = B(V); N, s\rangle \\
 & \text{if} & P \Downarrow V \\[1em]
\langle w, P, s\rangle & \leadsto & \langle w, V, s\rangle \\
 & \text{if} & P \Downarrow V \\[1em]
\langle w, V, (\measuredangle x.N)\cdot s\rangle & \leadsto & \langle w, N[x \mapsto V], s\rangle
\end{array}$$

Figure 15: L^2 machine transitions

calling the parsing function for the start symbol S with the empty continuation $\blacksquare$. An accepting configuration of the machine is of the form

$$\langle \varepsilon, V, \blacksquare\rangle$$

having consumed all input and returning a final value V with the empty continuation.

In a machine configuration $\langle w, N, s\rangle$, N represents the body of the current parsing function, or more accurately the part of it that has not been processed yet. The stack consists of pending calls to parsing functions. The rules in Figure 15 can be read as follows:

1. If the next parsing function to be called is B, then the parser generator determines which rule $B :\!-\ \alpha$ will be used in the predict move. The corresponding branch of the parsing function is selected, and the parameter V is passed to it; this parameter passing is modelled by substitution. The calling context of $B(V)$ is pushed onto that stack s.
2. If the next grammar symbol in the current expression is a non-terminal a, the machine matches it in the input.

$$
\begin{array}{lll}
 & E & \\
\to_{\mathcal{F}} & \texttt{1}\ E' & \text{by } E :- \texttt{1}\ E' \\
\to_{\mathcal{F}} & \texttt{1 -}\ E\ E' & \text{by } E' :- E\ E' \\
\to_{\mathcal{F}} & \texttt{1 - a}\ E'\ E' & \text{by } E :- \texttt{a}\ E \\
\to_{\mathcal{F}} & \texttt{1 - a}\ E' & \text{by } E' :- \varepsilon \\
\to_{\mathcal{F}} & \texttt{1 - a} & \text{by } E' :- \varepsilon
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \texttt{1 - a}, \texttt{let}\, x = E(e); x, s\rangle \\
(1) \rightsquigarrow & \langle \texttt{1 - a}, \ulcorner\texttt{1}\urcorner; \texttt{let}\, z = E'(e,1); z, s_1\rangle \\
(2) \rightsquigarrow & \langle \texttt{- a}, \texttt{let}\, z = E'(e,1); z, s_1\rangle \\
(3) \rightsquigarrow & \langle \texttt{- a}, \ulcorner\texttt{-}\urcorner; \texttt{let}\, y = E(e); \texttt{let}\, z = E'(e,1-y); z, s_1\rangle \\
(4) \rightsquigarrow & \langle \texttt{a}, \texttt{let}\, y = E(e); \texttt{let}\, z = E'(e,1-y); z, s_1\rangle \\
(5) \rightsquigarrow & \langle \texttt{a}, \ulcorner\texttt{a}\urcorner; \texttt{let}\, z = E'(e,3); z, (\nwarrow y.\texttt{let}\, z = E'(e,1-y); z)\cdot s_1\rangle \\
(6) \rightsquigarrow & \langle \varepsilon, \texttt{let}\, z = E'(e,3); z, (\nwarrow y.\texttt{let}\, z = E'(e,1-y); z)\cdot s_1\rangle \\
(7) \rightsquigarrow & \langle \varepsilon, 3, (\nwarrow y.\texttt{let}\, z = E'(e,1-y); z)\cdot s_1\rangle \\
(8) \rightsquigarrow & \langle \varepsilon, \texttt{let}\, z = E'(e,1-3); z, s_1\rangle \\
(9) \rightsquigarrow & \langle \varepsilon, \texttt{let}\, z = E'(e,-2); z, s_1\rangle \\
(10) \rightsquigarrow & \langle \varepsilon, -2, (\nwarrow z.z)\cdot s_1\rangle \\
(11) \rightsquigarrow & \langle \varepsilon, -2, (\nwarrow x.x)\cdot s\rangle \\
(12) \rightsquigarrow & \langle \varepsilon, -2, s\rangle
\end{array}
$$

where $s_1 \equiv (\nwarrow x.x)\cdot s$

Figure 16: L^2 machine run for parsing `1 - a` and evaluating it to -2

3. The parameter P of a parsing function B is evaluated to some value V which can then be passed to the parsing function.
4. If the current expression is a pure expression P, it is evaluated to some V. The value can then be passed to the continuation.
5. If the current expression has been reduced to a value V, then a frame is popped from the continuation and V is passed to it, again modelled by substitution.

The stack frames $(\curlywedge x.N)$ resemble lambda abstractions to some extent, in that they can be invoked with a value V. However, they are not first-class values and only reside on the stack of the machine. In an implementation, such as an ANTLR-generated parser in C, $(\curlywedge x.N)$ is implemented by the stack pointer. In the abstract machine, returning from a parsing function is modelled by substituting the return value V into the frame, in $N[x \mapsto V]$. In an implementation, this is implemented by the calling convention of the implementation language, such as placing V into a register.

Example 10.2 (L^2 machine run) As an example of how the L^2 machine works, consider the transformed grammar in Example 9.5. Let the input string be `1 - a`. Given an environment e, we would expect this string to be parsed and evaluated to some integer. Suppose that $e(\mathtt{a}) = 3$. Then the final result should be -2. See Figure 16 for the machine run, with a leftmost derivation of the initial input given for comparison. The initial stack s can be any stack; if it is the empty stack $\blacksquare$, then -2 is the final result of the whole machine run.

The machine steps (1), (3), (5), (7), and (10) correspond to derivation steps; the machine simulates them by calling parsing functions. Step (9) evaluates a function argument. The last two machine steps pop the continuation stack to return from a parsing function called earlier.

For understanding an abstract machine, it is useful to know its typing invariant.

Definition 10.3 Stacks of the L^2 machine are typed as follows:

$$\frac{}{\blacksquare : \neg[\![\, S \,\rangle\!\rangle} \qquad \frac{x : \tau_1 \Vdash N : \tau_2 \mathbin{!} \alpha \qquad s : \neg\tau_2}{((\curlywedge x.N) \cdot s) : \neg\tau_1}$$

In a reachable machine configuration $\langle w, N, s\rangle$, the current expression N is always a closed term of type $[\![\, A \,\rangle\!\rangle$, where A is the parsing function whose remaining body N represents. The stack s is of type $\neg\, [\![\, A \,\rangle\!\rangle$.

11. Conclusions

The immediate motivation for this paper was the desire to relate the *syntactic* transformation of left-recursion elimination to the well-understood *semantic* transformation of continuation passing. The intuitive idea of left-recursion elimination (as well as other grammar transformations) is to introduce a fresh non-terminal L' that represents *part* of some rule for L. The intuitive idea of continuations is that they represent the meaning of the *rest* of the computation [12]. To bridge the gap between syntax and semantics, it helps to have a framework that presents them in analogous terms.

The diagram chases emphasise what properties of the language for semantic actions are needed. It need not be a pure lambda calculus, but can also be a call-by-value lambda calculus with effects. Parsing actions often refer to a symbol table, which represents a global mutable state.

11.1. Related work

The fact that the semantics of the language needs to be systematically adapted when left recursion is eliminated appears to be part of compiling folklore. For instance, Aho, Sethi and Ullman [2] discuss L-attribute grammars, and Appel [35] gives an example of a hand-written LL(1) parser in Java where the parsing methods have been adapted by taking additional arguments, similar to Example 9.5.

Functional programming for parser construction is a well established stream of research, going back at least to the 1960s. However, here the idea is not to construct a parser. Instead, the parser should ideally be generated automatically. We are only concerned *that* the grammar can be parsed, rather than *how*. Research on constructing parsers in Haskell [36] or recent work on parsing with derivatives [37] is in general only distantly related to the aims of this paper.

Lohmann, Riedewald and Stoy [38] show how to transform semantic rules during left recursion elimination. They use the traditional compiler construction framework of Attribute Grammars [39], so that their semantics of grammar rules consists of equations between attributes. Some of their observations about moving such attribute rules across the transformed trees are closely related to the present paper (particularly Figure 12). In this paper, however, we use structures from the semantics of programming languages, such as categorical semantics, premonoidal categories and continuations, so that the results are quite different to theirs. Roughly speaking, inherited attributes push information down the parse tree towards the leaves, whereas synthesised attributes push information up towards the roots [2]. In the framework of L-categories, the semantics of a node $[\![A :- \alpha]\!]$ takes a value of type $[\![\alpha]\!]$ and returns value $[\![A]\!]$, thereby pushing information towards the root. If $[\![A]\!]$ is itself a function type, its parameter pushes information down the parse tree. ANTLR as well as some other modern parser generators are not based on attribute grammars, but on grammar rules having arguments and return values, which is closer to the approach of the present paper of modelling semantic actions as functions. Attribute grammars have been related to functional parsing actions [40]. In the context of parsing combinators in Haskell [41], Swierstra and coauthors have defined typed transformations of semantic actions [25], including left-recursion elimination via the left-corner transformation [24]. A major challenge they solve in their work is to accommodate grammar transformations in the Haskell type system, which is orthogonal to the aims of the present paper. Other authors have used dependent types in their formalization of the left-corner transform, which are expressive enough for correctness properties [27, 26].

The categorical framework, based on premonoidal categories [3], defined in the present paper may be applicable to proving the correctness of other constructions involving more general grammar transformations in addition to left recursion elimination. Unlike parsing combinators in Haskell, the semantics of parsing actions presented here is geared towards call-by-value languages with effects (not all morphisms are required to be central, permitting the actions to have computational effects such as updating state). Such call-by-value parsing actions are arguably closer to parser generators such as ANTLR [28, 7]. On the other hand, since arrows [5, 6] have been related to Freyd-categories [16], they may provide a link to parsing in Haskell. Atkey's recent work on parsing actions, while using different techniques, has similar aims in making the semantics less ad-hoc by using "techniques from programming language theory" [42].

The idea to seek analogues of continuation passing in syntax was inspired by continuations in natural language (see Barker's survey [43]) and Lambek's pioneering syntactic calculus [15], where left and right $\otimes$ are not symmetric. In addition to the left/right distinction, Lambek's calculus is also linear. Since the new grammar symbol L' introduced by left recursion elimination occurs only in a right-linear position, it is yet another instance of linearly used continuations [21], as well as answer type polymorphism [22].

11.2. Directions for further research

The category theory used in this paper was kept deliberately naive on a "need-to-know" basis, so to speak. It should be possible to give a more axiomatic definition of L-category along the same lines as premonoidal category [3], and the semantic action $[\![-]\!]$ as a contravariant functor to a Freyd category [16] preserving the premonoidal $\otimes$. Likewise, coherence isomorphisms have been glossed over in the present paper and could be addressed more formally. For syntax, the operation $\otimes$ is strictly associative, but its semantic counterpart is only associative up to isomorphism. Coherence as developed for premonoidal categories could be adapted to the latter situation.

The present paper is in the tradition of call-by-value programming languages with computational effects, using mathematical tools such as premonoidal and Freyd categories, continuations, type and effect systems, and abstract machines. The most widely studied effects in the literature, each neatly giving rise to a monad and a premonoidal category, are: state, continuations and, to a lesser degree, exceptions. Here we have developed parsing as another such effect, in the hope of building bridges between programming language theory and formal language theory (including parsing, but traditionally using automata-theoretic tools). In this regard, the L-calculus and L^2 machine from Sections 8 and 10 may have independent interest and be developed further.

As sketched in Section 7.2, the left-corner transformation, when extended to semantic actions, appears as a form of continuation passing, much as the simpler left-recursion elimination transform we have concentrated on. While Brink, Holdermans and Löh [27] define the left-corner transformation on semantics actions, they prove only language inclusion for the original grammar $\mathcal{G}$ and the transformed grammar $\mathcal{G}'$, that is, $\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{G}')$. They leave it for future work to "expand the proof of the language-inclusion property into a full correctness proof". It may be possible to combine their formalization in Agda with the semantic techniques in this paper to prove the full correctness of the left-corner transformation, including correctness of the transformed semantic actions.

References

[1] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright, Initial algebra semantics and continuous algebras, Journal of the ACM 24 (1977) 68–95.

[2] A. V. Aho, R. Sethi, J. D. Ullman, Compilers - Principles, Techniques and Tools, Addison Wesley, 1985.

[3] J. Power, E. Robinson, Premonoidal categories and notions of computation, Mathematical Structures in Computer Science 7 (1997) 453–468.

[4] E. Moggi, Computational lambda calculus and monads, in: Proceedings, Fourth Annual Symposium on Logic in Computer Science (LICS), 1989, pp. 14–23.

[5] J. Hughes, Generalising monads to arrows, Science of Computer. Programming 37 (2000) 67–111.

[6] R. Paterson, A new notation for arrows, in: ICFP, ACM, 2001.

[7] T. Parr, K. Fisher, LL(*): the foundation of the ANTLR parser generator, in: PLDI, ACM, 2011.

[8] C. A. Gunther, Semantics of Programming Languages, MIT Press, 1992.

[9] H. Thielecke, Functional semantics of parsing actions, and left recursion elimination as continuation passing, in: PPDP '12 Proceedings of the 14th symposium on Principles and practice of declarative programming, ACM Press, 2012, pp. 91–102. doi:10.1145/2370776.2370789.

[10] G. Steele, Rabbit: A Compiler for Scheme, Technical Report AI TR 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1978.

[11] G. D. Plotkin, Call-by-name, call-by-value, and the λ-calculus, Theoretical Computer Science 1 (1975) 125–159.

[12] C. Strachey, C. P. Wadsworth, Continuations: A Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, 1974.

[13] S. Mac Lane, Categories for the Working Mathematician, Springer Verlag, 1971.

[14] J. Power, H. Thielecke, Environments, continuation semantics and indexed categories, in: M. Abadi, T. Ito (Eds.), Proceedings TACS'97, number 1281 in LNCS, Springer Verlag, 1997, pp. 391–414.

[15] J. Lambek, The mathematics of sentence structure, American Mathematical Monthly 65 (1958) 154–170.

[16] J. Power, H. Thielecke, Closed Freyd- and kappa-categories, in: J. Wiedermann, P. van Emde Boas, M. Nielsen (Eds.), Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP), number 1644 in LNCS, Springer Verlag, 1999, pp. 625–634.

[17] P. J. Landin, The mechanical evaluation of expressions, The Computer Journal 6 (1964) 308–320.

[18] M. Felleisen, D. P. Friedman, Control operators, the SECD-machine, and the λ-calculus, in: M. Wirsing (Ed.), Formal Description of Programming Concepts, North-Holland, 1986, pp. 193–217.

[19] O. Danvy, A. Filinski, Representing control, a study of the CPS transformation, Mathematical Structures in Computer Science 2 (1992) 361–391.

[20] D. Sitaram, M. Felleisen, Control delilmiters and their hierarchies, Lisp and Symbolic Computation 3 (1990) 67–99.

[21] J. Berdine, P. W. O'Hearn, U. Reddy, H. Thielecke, Linear continuation passing, Higher-order and Symbolic Computation 15 (2002) 181–208.

[22] H. Thielecke, From control effects to typed continuation passing, in: Principles of Programming Languages (POPL'03), ACM, 2003, pp. 139–149.

[23] P. Wadler, Theorems for free!, in: Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'89), London, 11–13 September 1989, ACM Press, New York, 1989, pp. 347–359.

[24] R. Moore, Removing left recursion from context-free grammars, in: Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference, Association for Computational Linguistics, 2000, pp. 249–255.

[25] A. Baars, S. Doaitse Swierstra, M. Viera, Typed transformations of typed grammars: The left corner transform, Electron. Notes Theor. Comput. Sci. 253 (2010) 51–64.

[26] D. Devriese, F. Piessens, Explicitly recursive grammar combinators, Practical Aspects of Declarative Languages (PADL'11) (2011) 84–98.

[27] K. Brink, S. Holdermans, A. Löh, Dependently typed grammars, in: Mathematics of Program Construction (MPC'10), Springer, 2010, pp. 58–79.

[28] T. J. Parr, R. W. Quong, ANTLR: A predicated- LL(k) parser generator, Software, Practice and Experience 25 (1995) 789–810.

[29] J. M. Lucassen, D. K. Gifford, Polymorphic effect systems, in: Principles of Programming Languages (POPL '88), ACM, 1988, pp. 47–57.

[30] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1987) 1–102.

[31] P. W. O'Hearn, On bunched typing, Journal of Functional Programming 13 (2003) 747–796.

[32] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Logic in Computer Science (LICS), IEEE, 2002, pp. 55–74.

[33] D. P. Friedman, M. Wand, C. T. Haynes, Essentials of Programming Languages, MIT Press, 1992.

[34] X. Leroy, The ZINC experiment : an economical implementation of the ML language, Technical Report RT-0117, Inria, Institut National de Recherche en Informatique et en Automatique, 1990.

[35] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, Cambridge University Press, 1997.

[36] G. Hutton, E. Meijer, Monadic parsing in Haskell, Journal of Functional Programming 8 (1998) 437–444.

[37] M. Might, D. Darais, D. Spiewak, Parsing with derivatives: a functional pearl, in: ICFP, ACM, 2011, pp. 189–195.

[38] W. Lohmann, G. Riedewald, M. Stoy, Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars, Electronic Notes in Theoretical Compututer Science 110 (2004) 133–148.

[39] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (1968) 127–145.

[40] K. B. Oege de Moor, S. D. Swierstra, First class attribute grammars, Informatica: An International Journal of Computing and Informatics 24 (2000) 329–341. Special Issue: Attribute grammars and their Applications.

[41] S. D. Swierstra, Combinator parsers: a short tutorial, in: A. Bove, L. Barbosa, A. Pardo, , J. Sousa Pinto (Eds.), Language Engineering and Rigorous Software Development, volume 5520 of *LNCS*, Spinger, 2009, pp. 252–300.

[42] R. Atkey, The semantics of parsing with semantic actions, in: Symposium on Logic in Computer Science (LICS'12), IEEE, 2012, pp. 75–84.

[43] C. Barker, Continuations in natural language, in: Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04), University of Birmingham Computer Science technical report CSR-04-1, 2004.