Functional Semantics of Parsing Actions, and Left Recursion Elimination as Continuation Passing

Hayo Thielecke
University of Birmingham, UK
H.Thielecke@cs.bham.ac.uk

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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Theory

Keywords Continuations; functional programming; lambda calculus; parsing; semantics; types; verification

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 [8]. 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 [1]), 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. If we take compositionality seriously, the sematics 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:

$$E \quad :- \quad \texttt{1} \qquad (1)$$

$$E \quad :- \quad E - E \qquad (2)$$

Eliminating left recursion from this grammar 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:

$$E \quad :- \quad \texttt{1}\; E' \qquad (3)$$

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 [22], which were originally developed as an alternative to Moggi's monads as notions of computation [17] 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 [10, 20].

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:

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

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 [1]. The former include for example ANTLR [19], 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 [9]). 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.

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 terminal 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

Figure 1. Notational conventions

Outline of the paper

Section 2 describes parser generators in terms of simple abstract machines. Section 3 casts the notion of leftmost derivation (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 (Section 6) and reconstruct it as a form of continuation passing in Section 7. Section 8 concludes with a discussion of related work and directions for further research.

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 [1]. 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 kind 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 $A \to \alpha_1, \ldots, A \to \alpha_n$ 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 $a \neq b$.

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 [19] 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 $A \to \alpha$. (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 [1]. 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) [21, 28, 29].

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 [16] for the graph $\mathbf{Graph}(\mathcal{G})$. Morphisms in the L-category are called leftmost derivations.

We write morphisms in the L-category as lists of pairs 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 [1] 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 list :: for list cons, as in ML, we define:

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

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

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

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

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

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

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

The definition of L-category is inspired of Power and Robinson's premonoidal categories [22, 23] and by Lambek's syntactic calculus [13]. 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 $\beta\,\alpha$.

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 $A \mathrel{:-} f$ 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 [17] or a Freyd category [24]. 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 [22]. 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 [17, 21].

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

- For each non-terminal symbols A of $\mathcal{G}$, a type $[\![A]\!]$. For a terminal symbol a, its semantic type is the unit type containing only the empty tuple:

  $$[\![a]\!] \overset{\text{def}}{=} \textbf{Unit} = \{\,()\,\}$$

  The types are extended to strings of symbols $X_1 \dots X_n$ by taking the product of the types of the X_j:

  $$[\![X_1 \dots X_n]\!] \overset{\text{def}}{=} [\![X_1]\!] \otimes \cdots \otimes [\![X_n]\!]$$

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

  $$[\![A \mathrel{:-} \alpha]\!] : [\![\alpha]\!] \longrightarrow [\![A]\!]$$

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

- If d is the empty derivation of a string α from itself, $d = [\![\,[\,]\,]\!] : \alpha \to_{\mathcal{G}} \alpha$, then its semantic action is the identity function on the semantic of the string α:

  $$[\![d]\!] \overset{\text{def}}{=} \mathsf{id}_{[\![\alpha]\!]}$$

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

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

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

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

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

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

$$\begin{tikzcd}
S \arrow[r, "d_1"] & v_1\,A\,\sigma \arrow[r, "v_1 \otimes (A :\!- \alpha) \otimes \sigma_1"] & v_1\,\alpha\,\sigma_1
\end{tikzcd}$$

$$\begin{tikzcd}
\llbracket S\rrbracket & \llbracket A\rrbracket \otimes \llbracket \sigma_1\rrbracket \arrow[l, "\llbracket d_1\rrbracket = k_1"'] & \llbracket \alpha\rrbracket \otimes \llbracket \sigma_1\rrbracket \arrow[l, "\llbracket A :\!- \alpha\rrbracket \otimes \llbracket \sigma_1\rrbracket"']
\end{tikzcd}$$

Figure 4. WSK machine move for a grammar rule $A :\!- \alpha$ with syntactic and semantic invariants

path category [16]. 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 Defintion 3.1 is that it makes it easier to define the action of a derivation in Definition 4.2. The values of types $\llbracket w\rrbracket$ and $\llbracket \beta\rrbracket$ are simply carried along when we define the meaning of a grammar rule application in this context:

$$\llbracket (w\,A\,\beta, w\,\alpha\,\beta)\rrbracket : (\llbracket w\rrbracket \otimes \llbracket \alpha\rrbracket \otimes \llbracket \beta\rrbracket) \longrightarrow (\llbracket w\rrbracket \otimes \llbracket A\rrbracket \otimes \llbracket \beta\rrbracket)$$

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

The benefit of having defined a semantic action $\llbracket d\rrbracket$ 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 [14] and CEK [7] 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.

Definition 4.3 (WSK machine) Given a grammar $\mathcal{G}$ with a semantic action $\llbracket - \rrbracket$, 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:
$$\begin{array}{rcl} \langle w, A\,\sigma, k\rangle & \rightsquigarrow & \langle w, \alpha\,\sigma, (\llbracket A :\!- \alpha\rrbracket \otimes \llbracket \sigma\rrbracket)\cdot k\rangle \\ \langle a\,w, a\,\sigma, k\rangle & \rightsquigarrow & \langle w, \sigma, \mathbf{unitleft}\cdot k\rangle \end{array}$$
Here **unitleft** is the left identity for the unit type:
$$\mathbf{unitleft} : (\mathbf{Unit} \otimes \llbracket \sigma\rrbracket) \longrightarrow \llbracket \sigma\rrbracket$$
As a term, $\mathbf{unitleft} = (\lambda((), x).x)$.

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 $k = \llbracket d\rrbracket$.

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 $k = \llbracket d_1\rrbracket$. 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 Lemma 4.4, the type of the final action is k : **Unit** $= \llbracket \varepsilon\rrbracket \to \llbracket S\rrbracket$. Hence we have an element of the semantic type $\llbracket S\rrbracket$ of the start symbols as the final answer computed by the parser using the actions given by $\llbracket - \rrbracket$ 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 [1]. 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:

$$\llbracket C\rrbracket \overset{\text{def}}{=} \llbracket B\rrbracket \to \llbracket A\rrbracket$$

For all other symbols X, we let $\llbracket X\rrbracket = \llbracket X\rrbracket$, and we write simply $\llbracket X\rrbracket$ for both. Let

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

Original rules for L:

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

Transformed rules:

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

Figure 5. Transformation of grammar rules

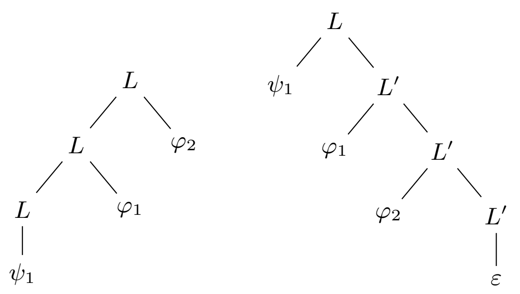

Figure 6. Parse trees for $L \rightarrow_{\mathcal{G}} \psi_1\,\varphi_1\,\varphi_2$ and $L \rightarrow_{\mathcal{F}} \psi_1\,\varphi_1\,\varphi_2$ in the original and transformed grammar

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

$$\begin{array}{rcl} [\![C :- \alpha]\!] & \overset{\text{def}}{=} & \lambda x.\lambda y.f(y,x) \\ & : & [\![\alpha]\!] \to ([\![B]\!] \to [\![A]\!]) \\[1em] [\![C :- \beta]\!] & \overset{\text{def}}{=} & \lambda x.\lambda y.g(y,x) \\ & : & [\![\beta]\!] \to ([\![B]\!] \to [\![A]\!]) \\[1em] [\![A :- B\,C]\!] & \overset{\text{def}}{=} & \lambda x.\lambda f.f(x) \\ & : & ([\![B]\!] \otimes \underbrace{([\![B]\!] \to [\![A]\!])}_{[\![C]\!]}) \to [\![A]\!] \end{array}$$

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 [1]. 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 symbols 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 5.

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 6, 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 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:

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

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:

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

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.

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

Figure 7. From $\llbracket L :- L\,\varphi_i \rrbracket$ to $\llbracket L' :- \varphi_i\, L' \rrbracket$

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

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

Definition 5.3 (Transformation of semantic actions) Let $\mathcal{G}$ be a grammar with a semantic action $\llbracket - \rrbracket$. 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 $\llbracket - \rrbracket$ for $\mathcal{F}$ that matches how grammar rules are transformed in Figure 5.

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

$$\llbracket L' \rrbracket \overset{\text{def}}{=} \llbracket L \rrbracket \longrightarrow \llbracket L \rrbracket$$

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

$$f \overset{\text{def}}{=} \llbracket L :- L\,\varphi_i \rrbracket : (\llbracket L \rrbracket \otimes \llbracket \varphi_i \rrbracket) \longrightarrow \llbracket L \rrbracket$$

  As shown in Figure 7, we first construct a morphism (in the middle) column

$$(\llbracket \varphi_i \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket) \otimes \llbracket L \rrbracket) \to \llbracket L \rrbracket$$

  Lambda abstraction of $\llbracket L \rrbracket$ gives us

$$\begin{array}{ll} & \llbracket L' :- \varphi_i\, L' \rrbracket \\ : & (\llbracket \varphi_i \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket)) \to (\llbracket L \rrbracket \to \llbracket L \rrbracket) \end{array}$$

  Equivalently, using lambda calculus syntax, we write

$$\llbracket L' :- \varphi_i\, L' \rrbracket \overset{\text{def}}{=} \lambda(p,k).\lambda x.k(f(x,p))$$

  Here the types are:

$$p : \llbracket \varphi_i \rrbracket \qquad k : \llbracket L \rrbracket \longrightarrow \llbracket L \rrbracket \qquad x : \llbracket L \rrbracket$$

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

$$g \overset{\text{def}}{=} \llbracket L :- \psi_j \rrbracket : \llbracket \psi_j \rrbracket \to \llbracket L \rrbracket$$

  As shown in Figure 8, we add a parameter of type $\llbracket L \rrbracket \to \llbracket L \rrbracket$ and apply it to the result of g, which gives us a morphism

$$\begin{array}{ll} & \llbracket L :- \psi_j\, L' \rrbracket \\ : & (\llbracket \psi_j \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket)) \to \llbracket L \rrbracket \end{array}$$

  Writing the same with lambda terms, we have:

$$\llbracket L :- \psi_j\, L' \rrbracket \overset{\text{def}}{=} \lambda(p,k).k(g(p))$$

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

$$\textbf{unitleft} : (\textbf{Unit} \otimes \llbracket L \rrbracket) \to L$$

  and lambda-abstract, to give

$$\begin{array}{ll} & \llbracket L' :- \varepsilon \rrbracket \\ : & \llbracket \varepsilon \rrbracket \to (\llbracket L \rrbracket \to \llbracket L \rrbracket) \end{array}$$

  Written as a lambda term, $\llbracket L' :- \varepsilon \rrbracket \overset{\text{def}}{=} (\lambda\,()\,.(\lambda x.x))$.
- 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 $\llbracket E \rrbracket = \mathbb{N}$ and the semantic functions as follows:

$$\begin{array}{rcl} \llbracket E :- \mathtt{1} \rrbracket & \overset{\text{def}}{=} & \lambda(()).1 \\ \llbracket E :- E - E \rrbracket & \overset{\text{def}}{=} & \lambda(x,(),y).x - y \end{array}$$

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

$$\llbracket E' \rrbracket \quad \overset{\text{def}}{=} \quad \mathbb{N} \longrightarrow \mathbb{N}$$

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

$$\begin{array}{rcl}
\llbracket E :- \mathtt{1}\ E' \rrbracket & \stackrel{\text{def}}{=} & \lambda((), k).k(1) \\
 & : & (\textbf{Unit} \otimes (\mathbb{N} \to \mathbb{N})) \to \mathbb{N} \\
\llbracket E' :- \mathtt{-}\ E\ E' \rrbracket & \stackrel{\text{def}}{=} & \lambda((), x, k).\lambda y.k(y - x) \\
 & : & (\textbf{Unit} \otimes \mathbb{N} \otimes (\mathbb{N} \to \mathbb{N})) \to (\mathbb{N} \to \mathbb{N}) \\
\llbracket E' :- \varepsilon \rrbracket & \stackrel{\text{def}}{=} & \lambda().\lambda x.x \\
 & : & \textbf{Unit} \to (\mathbb{N} \to \mathbb{N})
\end{array}$$

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

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

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

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 $d : \psi_j \to_{\mathcal{F}} w_2$. This gives us a derivation

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

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

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

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 $\llbracket L \rrbracket \otimes \textbf{Unit} \cong \llbracket L \rrbracket$. Hence the following diagram commutes by the definition of $\llbracket L' :- \varepsilon \rrbracket$ in Definition 5.3:

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

We define $d^{\triangle} \stackrel{\text{def}}{=} d_1^{\triangle}$. 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 9 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 8).
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:

$$\begin{tikzcd}
L \arrow[r, "d_1"] \arrow[dd, "L :- L\, \varphi_i"] & w_1\, L' \arrow[d, "w_1 \otimes (L' :- \varphi_i\, L')"] & \\
 & w_1\, \varphi_i\, L' \arrow[r, "w_1 \otimes d_2 \otimes L'"] & w_1\, w_2\, L' \\
L\, \varphi_i \arrow[r, "d_1^{\triangle} \otimes \varphi_i"] & w_1\, \varphi_i \arrow[r, "w_1 \otimes d_2^{\triangle}"] & w_1\, w_2
\end{tikzcd}$$

The property we need to prove is given in Figure 10. The right rectangle commutes because $\llbracket d_2^{\triangle} \rrbracket = \llbracket d_2 \rrbracket$. The left rectangle commutes because $\llbracket L' :- \varphi_i\, L' \rrbracket$ is central and due to its definition (see Figure 7).

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 [1]; af-

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

Figure 9. Simulation for a $L :- \psi_j L'$ step

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

Figure 10. Simulation for a derivation d_1 followed by a $L' :- \varphi_i\, L'$ step

ter 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.

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

$$\overline{M} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$$

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 $\llbracket E \rrbracket = \mathbb{N}$:

$$\llbracket E' \rrbracket \quad = \quad \mathbb{N} \longrightarrow \mathbb{N}$$

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

$$\begin{array}{rcl} \llbracket E :- \mathtt{1}\; E' \rrbracket & = & \lambda((),k).k(1) \\ \llbracket E' :- \mathtt{-}\; E\; E' \rrbracket & = & \lambda((),x,k).\lambda y.k(y-x) \\ \llbracket E' :- \varepsilon \rrbracket & = & \lambda().\lambda x.x \end{array}$$

For the last rule, we know that the ε-rule is applied at the end of a sub-derivation 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 [6] and Felleisen et. al. [27]. 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 6 in terms of semantic actions in Figure 11. 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* [5].

The equations that we need to prove correctness of the semantic transform lend themselves to answer type polymorphism [31]. Recall the transformation of semantic actions from Definition 5.3. For an action in the original grammar,

$$f = \llbracket L :- L\, \varphi_i \rrbracket$$

the transformed grammar has the action

$$\llbracket L' :- \varphi_i\, L' \rrbracket = \lambda(p,k).\lambda x.k(f(x,p))$$

So far, we have typed this term with $k : \llbracket L \rrbracket \longrightarrow \llbracket L \rrbracket$. But the answer type of the continuation need not be $\llbracket L \rrbracket$. It could be a type

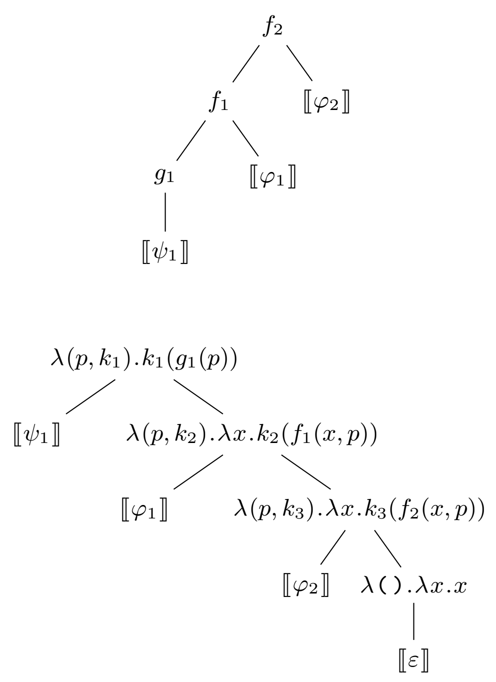

Figure 11. Semantic actions for original and transformed parse trees for $\psi_1\ \varphi_1\ \varphi_2$

variable χ. We give 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" [32]. 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 9 and 10. 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).

8. 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 [29]. 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.

8.1 Related work

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

Lohmann, Riedewald and Stoy [15] show how to transform semantic rules during left recursion elimination. They use the traditional compiler construction framework of Attribute Grammars [12], 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 11). 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 [1]. In the framework of L-categories, the semantics of a node $[\![A \coloneq \alpha]\!]$ 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 [18].

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 constructed automatically. We are only concerned about *that* the grammar can be parsed, rather than *how*, so that research on constructing parsers in Haskell [11] is only distantly related to the aims of this paper. In the context of parsing combinators in Haskell [30], Swierstra and coauthors have defined typed transformations of semantic actions [3], including left-recursion elimination and related transformations. The categorical framework defined in the present paper may be applicable to proving the correctness of their 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 tools such as ANTLR. On the other hand, since arrows [10, 20] have been related to Freyd-categories, they may provide a link to parsing in Haskell.

The idea to seek analogues of continuation passing in syntax was inspired by continuations in natural language (see Barker's survey [4]) and Lambek's pioneering syntactic calculus [13], 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 another instance of linearly used continuations [5].

8.2 Directions for further research

The use of lambda calculus for the semantic actions is not always realistic, even though lambda calculus is commonly used as a tractable idealisation of programming languages. While there are parser generator tools for functional languages, other widely used tools support Java or C, and one may wish to use C for efficiency. There are two ways the actions could be refined to be closer to such parser generator tools. The first possibility is to use a variant of lambda calculus, either a syntactic restriction or perhaps a substructural type system. The other possibility is to represent lambda terms in a way that is suitable for a language lacking first-class functions. This would essentially be a form of defunctionalization [26] and explicitly representing closures as data structures. For example, instead of a lambda term $\lambda y.x - y$, we would need to build a data structure recording both the current value of x and the fact that some future value y will need to be subtracted from it. Modern tools support Java as the language in which semantic actions are written, so that function closures could be simulated by way of objects.

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 [22], and the semantic action $[\![-]\!]$ as a contravariant functor to a Freyd category [24] preserving the functor $\otimes$. 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.

References

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

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

[3] A. Baars, S. Doaitse Swierstra, and M. Viera. Typed transformations of typed grammars: The left corner transform. *Electron. Notes Theor. Comput. Sci.*, 253:51–64, September 2010. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2010.08.031. URL `http://dx.doi.org/10.1016/j.entcs.2010.08.031`.

[4] 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.

[5] J. Berdine, P. W. O'Hearn, U. Reddy, and H. Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15 (2/3):181–208, 2002.

[6] O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4): 361–391, Dec. 1992.

[7] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986.

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

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

[10] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[11] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.

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

[13] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

[14] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964.

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

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

[17] E. Moggi. Computational lambda calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 14–23, 1989. ISBN 0-8186-1954-6.

[18] K. B. Oege de Moor and S. D. Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. ISSN ISSN 0350-5596. Special Issue: Attribute grammars and their Applications.

[19] T. Parr and K. Fisher. LL(*): the foundation of the ANTLR parser generator. In *PLDI*. ACM, 2011. ISBN 978-1-4503-0663-8.

[20] R. Paterson. A new notation for arrows. In *ICFP*. ACM, 2001. ISBN 1-58113-415-0.

[21] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[22] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5): 453–468, 1997.

[23] J. Power and H. Thielecke. Environments, continuation semantics and indexed categories. In M. Abadi and T. Ito, editors, *Proceedings TACS'97*, number 1281 in LNCS, pages 391–414. Springer Verlag, 1997.

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

[25] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25^{th} ACM National Conference*, pages 717–740. ACM, Aug. 1972.

[26] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprint of a conference paper [25].

[27] D. Sitaram and M. Felleisen. Control delilmiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

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

[29] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Jan. 1974.

[30] S. D. Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, , and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Spinger, 2009.

[31] H. Thielecke. From control effects to typed continuation passing. In *Principles of Programming Languages (POPL'03)*, pages 139–149. ACM, 2003.

[32] 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*, pages 347–359. ACM Press, New York, 1989.