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

Hayo Thielecke
University of Birmingham, UK

PPDP 2012
Contributions

1. categorical model of parsing actions $\llbracket - \rrbracket$
2. killer app of the model: left-recursion elimination as continuation passing style

Outline: first vertical $\llbracket - \rrbracket$, then horizontal transformations

Grammar $\xrightarrow{\text{Left recursion elimination}}$ Transformed grammar

$\llbracket - \rrbracket$ $\downarrow$ $\llbracket - \rrbracket$ (extended)

Meaning $\xrightarrow{\text{Continuation passing}}$ Transformed meaning

Compare MacLane: in order to define natural transformation, had to define functor.
Some notation, following Dragon Book where possible

$A, B, C, E, L$ : nonterminal symbols of a grammar

$\alpha, \beta, \gamma, \delta, \varphi, \psi$ : strings of terminal or nonterminals

$\varepsilon$ : empty string

$\lambda$ : lambda-abstraction; not used for strings

$v, w, w_1, w_2$ : strings of terminal symbols

$A \rightarrow \alpha$ : grammar rule for replacing $A$ by $\alpha$

$X \rightarrow Y$ : morphism from $X$ to $Y$ including:
functions for semantics and derivations for syntax

$[[\alpha]]$ : semantics of a string $\alpha$ as a type or set

$[[d]]$ : semantics of a derivation $d$ as a function
Motivation: “Idealized ANTLR”

Left recursion is a problem for LL parsers

- hand-written recursive descent parsers
- LL parser generators, like ANTLR: widely used tools, not just in functional programming

Example of direct left-recursion and its elimination

1. \[ E :\!\!\!\!\!:\!\!\!\!\!: 1 \] (1)
2. \[ E :\!\!\!\!\!:\!\!\!\!\!: E - E \] (2)

New symbol \( E' \) and rules:

3. \[ E :\!\!\!\!\!: 1 E' \] (3)
4. \[ E' :\!\!\!\!\!:\!\!\!\!: - E E' \] (4)
5. \[ E' :\!\!\!\!\!:\!\!\!\!: \varepsilon \] (5)

What is the meaning of things like \(- E E'\), and why?
A category of leftmost derivations ($\cong$ LL parsing)

Objects: strings $\alpha$

Morphisms: leftmost derivations

Two functors on derivations $d$:

- $d \otimes \alpha$ for strings $\alpha$
- $w \otimes d$ for strings $w$ of terminal symbols; $A \otimes d$ not allowed

The functors $\otimes$ generalize premonoidal categories (see also Arrows)

\[
\dfrac{A ::= \alpha}{[(A, \alpha)]: A \to \alpha}
\]

\[
\dfrac{d_1: \alpha \to \beta \quad d_2: \beta \to \gamma}{d_1 \cdot d_2: \alpha \to \gamma}
\]

\[
\dfrac{d: \beta \to \gamma}{d \otimes \alpha: \beta \alpha \to \gamma \alpha}
\]

\[
\dfrac{d: \alpha \to \beta}{w \otimes d: w \alpha \to w \beta}
\]
Functional semantics of parsing actions

Parsing actions (≈ input to parser generator)

- For each non-terminal symbols $A$ there is a type $[A]$.
  Extended to strings:
  \[
  [X_1 \ldots X_n] \overset{\text{def}}{=} [X_1] \otimes \cdots \otimes [X_n]
  \]

- For each grammar rule $A ::= \alpha$, there is a function of type
  \[
  [A ::= \alpha] : [\alpha] \longrightarrow [A]
  \]

Parsing actions extended to derivations (≈ runs of the parser)

- Each run of the parser generates a leftmost derivation
  \[
  d : \alpha \rightarrow \beta.
  \]
- Its semantics is a function $[d] : [\beta] \rightarrow [\alpha]$. 
Left-recursion elimination: syntactic transformation

Original rules for $L$:

$L$ :- $\psi_1$

$\vdots$

$L$ :- $\psi_m$

$L$ :- $L \ \varphi_1$

$\vdots$

$L$ :- $L \ \varphi_n$

Transformed rules with $L'$:

$L$ :- $\psi_1 \ L'$

$\vdots$

$L$ :- $\psi_m \ L'$

$L'$ :- $\varphi_1 \ L'$

$\vdots$

$L'$ :- $\varphi_n \ L'$

$L'$ :- $\varepsilon$
Semantic transformation = continuation passing

Reminder: continuation passing style transformation

\[
\bar{1} = \lambda k. k \, 1 \\
\bar{M_1 - M_2} = \lambda k. \bar{M_1} (\lambda x. \bar{M_2}(\lambda y. k(x - y)))
\]

Double negation on types: \( \bar{M} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \)

Transformation on types for left recursion elimination

\[
\begin{align*}
\llbracket L' \rrbracket & \overset{\text{def}}{=} \llbracket L \rrbracket \to \llbracket L \rrbracket \\
L & ::= L \varphi_i \quad \text{original rule} \\
L' & ::= \varphi_i L' \quad \text{transformed rule} \\
(\llbracket L \rrbracket \to \llbracket L \rrbracket) & \leftarrow (\llbracket \varphi_i \rrbracket \otimes (\llbracket L \rrbracket \to \llbracket L \rrbracket))
\end{align*}
\]
Example: parse tree for $\psi_1 \varphi_1 \varphi_2$ is turned inside out
Semantic actions for original and transformed parse trees

\[
\lambda(p, k_1).k_1(g_1(p))
\]

\[
\lambda(p, k_2).\lambda x.k_2(f_1(x, p))
\]

\[
\lambda(p, k_3).\lambda x.k_3(f_2(x, p))
\]

Linearly used continuations: \( \lambda k. \ldots k(\ldots) \)
Main result and proof technique

Simulation theorem
For each leftmost derivation in the transformed grammar

\[ d : \alpha \rightarrow \beta \text{ where } L' \text{ does not occur in } \alpha \text{ or } \beta \]

there is a derivation \( d^\Delta \) in the original grammar such that

\[ [d^\Delta] = [d] \]

Proof technique by diagram chase

- The proof uses the contravariant functor \([-]\) preserving \(\otimes\).
- Need induction hypothesis: continuation \(k\)
- Direct style: compose with \(k\)
- Continuation passing style: pass \(\lambda k\) as an argument
Simulation for a $L \vdash \psi_j L'$ step by a $L \vdash \psi_j$ step

\[
\begin{array}{c}
L & \xrightarrow{L \vdash \psi_j L'} & \psi_j \otimes L' \\
& & \xrightarrow{d_1 \otimes L'} & w \otimes L'
\end{array}
\]
Simulation for a $L :- \psi_j L'$ step by a $L :- \psi_j$ step

\[
\begin{align*}
L & \xrightarrow{L :- \psi_j L'} \psi_j \otimes L' \\
L & \otimes \alpha \xrightarrow{L :- \psi_j \otimes \alpha} \psi_j \otimes \alpha \\
\end{align*}
\]
Simulation for a $L : \vdash \psi_j L'$ step by a $L : \vdash \psi_j$ step
Simulation for a $L : \psi_j L'$ step by a $L : \psi_j$ step

\[
\begin{array}{c}
\left[ L \right] \quad \left[ \psi_j \right] \quad \left[ L' \right] \quad \left[ w \right] \\
\left[ L \right] \left[ \alpha \right] \quad \left[ \psi_j \right] \left[ \alpha \right] \quad \left[ w \right] \left[ \alpha \right]
\end{array}
\]
Related work

- Attribute motion for left-recursion elimination (Lohmann, Riedewald & Stoy)
- Parsing combinators in Haskell and transformations (Swierstra et al)
- Premonoidal categories and Freyd categories (Power & Robinson; Power & Thielecke)
- Linearly used continuations \((A \to \text{Ans}) \leadsto \ldots\) (Berdine, O’Hearn, Reddy & Thielecke)
- Syntax as a non-commutative substructural logic (Lambek 1958)
- Reg exp and parsing with derivatives (Sulzman & Lu; Might et al)
- With Asiri Rathnayake: regular expressions using abstract machines
Conclusions

- Syntax and semantics connected via functor
- Diagram chase as proof method
- Induction hypothesis on continuation $k$
- Possible applications: perform transform automatically for parser generators like ANTLR, correctness
- The ingredients are $\otimes$ and a contravariant negation, much like $\otimes\neg$-categories
- Continuations are everywhere, not just about call/cc in Scheme; discovered multiple times
The End

Thank you
Example grammar revisited

\[
\begin{align*}
\llbracket E' \rrbracket & \overset{\text{def}}{=} \mathbb{N} \rightarrow \mathbb{N} \\
\end{align*}
\]

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

\[
\begin{align*}
\begin{array}{ll}
\llbracket E :- 1 E' \rrbracket & \overset{\text{def}}{=} \lambda((), k). k(1) \\
& : (\text{Unit} \otimes (\mathbb{N} \rightarrow \mathbb{N})) \rightarrow \mathbb{N} \\
\end{array} \\
\begin{array}{ll}
\llbracket E' :- E E' \rrbracket & \overset{\text{def}}{=} \lambda((), x, k). \lambda y. k(y - x) \\
& : (\text{Unit} \otimes \mathbb{N} \otimes (\mathbb{N} \rightarrow \mathbb{N})) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \\
\end{array} \\
\begin{array}{ll}
\llbracket E' :- \varepsilon \rrbracket & \overset{\text{def}}{=} \lambda(). \lambda x. x \\
& : \text{Unit} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \\
\end{array}
\end{align*}
\]
Premonoidal $\otimes$ and central morphisms

$f$ is central iff for all $g$ the two compositions agree:

$$\begin{array}{c}
X_1 & \xrightarrow{f} & X_2 \\
\downarrow g & & \downarrow g \\
Y_1 & \xrightarrow{f \otimes Y_1} & \otimes Y_1 \\
\downarrow & & \downarrow \\
Y_2 & \xrightarrow{X_1 \otimes g} & \otimes Y_2 \\
\end{array}$$