Feature-driven Movement as Delimited Control

Christina Unger (UiL-OTS, Universiteit Utrecht)
christina.unger@let.uu.nl

Lambda Calculus and Formal Grammar
Sept 18-19 2007, Nancy
Background

- there is a close relation between ideas of Minimalist syntax and Categorial Grammars
- this connection has been shed light on mainly by developing Minimalist ideas in Multimodal Categorial Grammars (Amblard, Cornell, Lecomte, Retoré, Vermaat a.o.)
Correspondences:

- categorial features = slash types, categorial feature checking = Modus Ponens (/, \-elimination)
- formal features = unary operators ♦, □
  formal feature checking = ♦□A → A
- movement in order to check features = restructuring of grammatical material in order to apply the relevant inferences
- functional nodes (v, T, C) = composition modes
Goal

- explore the other direction of the mapping, i.e. from MMCG to Stabler’s Minimalist Grammars
- more specifically: exploit parallels to MMCG for the design of a semantics for MG
- and even more specifically: by using control operators in analogy to unary connectives
1 Motivation

2 Setting the stage
   - Minimalist Grammars
   - Control operators

3 MG with control
   - Semantics using control and prompt
   - Semantics using $C$

4 Toy fragment

5 Conclusion
Minimalist Grammars

A Minimalist Grammar is a four-tuple $G = \langle V, Cat, Lex, F \rangle$ where

- $V$, the vocabulary, is a finite non-empty set
- $Cat$ is the set of syntactic features
- $Lex$, the lexicon, is a finite set of simple expressions built from $V$ and $Cat$
- $F = \{\text{merge}, \text{move}\}$ is the set of structure-building operations

A language $L(G)$ is the closure of $Lex$ under the operations.
Minimalist expressions as trees

**Simple and complex expressions:**

\[
exp ::= (phon, cat, lic, sem) \mid (phon, cat, lic, sem, exp, exp)
\]
Minimalist expressions as trees

**Simple and complex expressions:**

\[ \text{exp} ::= (\text{phon}, \text{cat}, \text{lic}, \text{sem}) \mid (\text{phon}, \text{cat}, \text{lic}, \text{sem}, \text{exp}, \text{exp}) \]

**Category features:**

\[ \text{cat} = (\text{sel}^*) \text{base} \quad \text{where} \]

- \[ \text{sel} ::= =\text{base} \]
- \[ \text{base} ::= \text{NP} \mid \text{V} \mid \text{v} \mid \text{T} \mid \text{C} \mid \text{Det} \mid \text{N} \]
Minimalist expressions as trees

Simple and complex expressions:

\[ exp ::= (phon, cat, lic, sem) \mid (phon, cat, lic, sem, exp, exp) \]

Category features:

\[ cat = (sel^*) \text{ base} \quad \text{where} \]

- \[ sel ::= = \text{base} \]
- \[ base ::= \text{NP} \mid \text{V} \mid \text{v} \mid \text{T} \mid \text{C} \mid \text{Det} \mid \text{N} \]

Formal features:

\[ lic ::= +L \mid -L \quad \text{where} \quad L = \text{case} \mid \text{wh} \mid ... \]
Structure-building operations 1: merge

Merge cancels selector and selected features.

**First merge** of a simple head $\alpha$ and a simple (complex) complement $\beta$:

$$\text{merge1 } \alpha @ (\text{alpha, } = f \gamma, F_1, M) \beta @ (\text{beta, } f, F_2, N (, e_1, e_2)) = (\text{alpha beta, } \gamma, F_1, (M \ N), \alpha', \beta')$$
Structure-building operations 1: merge

Merge cancels selector and selected features.

**First merge** of a simple head $\alpha$ and a simple *(complex)* complement $\beta$:

$\text{merge1 } \alpha@(\text{alpha, } = f\gamma, F_1, M) \beta@(\text{beta, } f, F_2, N (, e_1, e_2)) = (\text{alpha beta, } \gamma, F_1, (M N), \alpha', \beta')$

**Second merge** of a simple *(complex)* specifier $\beta$ and a complex expression $\alpha$ containing a selecting head:

$\text{merge2 } \beta@(\text{beta, } f, F_2, N (, e_1, e_2)) \alpha@(\text{alpha, } = f\gamma, F_1, M, e_3, e_4) = (\text{beta alpha, } \gamma, F_1, (M N), \beta', \alpha')$
Structure-building operations 2: move

Move cancels licensor and licensee feature.

The unary **move operation** applies to a complex expression with a head containing a licensor feature:

\[
\text{move } f \ (\alpha, \gamma_1, F_1, M, e_1, e_2) = \\
(b \alpha, \gamma_1, F_1 \{+f\}, [\text{move } M], \beta', e_1 e'_2)
\]

for a feature \( f \) such that \( +f \in F_1 \) and \( e_2 \) contains exactly one expression \( \beta \ast (b, \gamma_2, F_2, N) \) with \( -f \in F_2 \).
Example

**Numeration:**

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>gilgamesh</td>
<td>NP</td>
<td>-case</td>
</tr>
<tr>
<td>whom</td>
<td>NP</td>
<td>-wh</td>
</tr>
<tr>
<td>liked</td>
<td>=NP</td>
<td>V</td>
</tr>
<tr>
<td>⍵</td>
<td>=V</td>
<td>=NP v</td>
</tr>
<tr>
<td>⍵</td>
<td>=v</td>
<td>T +case</td>
</tr>
<tr>
<td>⍵</td>
<td>=T</td>
<td>C +wh</td>
</tr>
</tbody>
</table>
**Continuation-passing style**

**Continuations**: functional representations of the rest of the computation

**Delimited continuations**: functional representations of a part of the rest of the computation

**Continuation-passing style** (CPS) transformations make control transfers (such as jumps or procedure calls) explicit; that’s why CPS is one of the standard frameworks for understanding, comparing, implementing and reasoning about control operators.
Continuation-based operators for delimited control

- $C, \mathcal{F}$ (Felleisen & Friedman)
- control and prompt (Felleisen & Sitaram)
- shift and reset (Danvy & Filinsky)
- cupto and set (Gunter, Rémy & Riecke)
- ...
Call-by-value λ-calculus extended with control operators:

\[ V ::= x \mid \lambda x. E \]

\[ E ::= V \mid (E E) \mid \text{control } f \ E \mid \text{prompt } E \]

Evaluation contexts:

\[ C[] ::= [] \mid C[(E [])] \mid C([], V) \]

\[ M[] ::= [] \mid M[\text{prompt } C[]] \]

Control captures the current continuation, which is delimited by the closest enclosing prompt.
control and prompt: Operational semantics

\[ M[(\text{prompt } V)] \triangleright M[V] \]
\[ M[(\text{prompt } C[(\text{control } h \ E)])] \triangleright M[(\text{prompt } E')] \]

where \( E' = E \{ h \mapsto \lambda x. C[x] \} \)
control and prompt: Operational semantics

\[
M[(\text{prompt } V)] \triangleright M[V] \\
M[(\text{prompt } C[(\text{control } h E)])] \triangleright M[(\text{prompt } E')] \\
\text{where } E' = E\{h \mapsto \lambda x. C[x]\}
\]

Example:

\[
(+ 5 (\text{prompt } (* 2 (\text{control } h (+ 1 (h 3)))))) \\
\triangleright (+ 5 (\text{prompt } (+ 1 (* 2 3)))) \\
\triangleright (+ 5 (\text{prompt } 7)) \\
\triangleright (+ 5 7)
\]
First idea

- merge = functional application (= modus ponens)
- \(-f = \text{control}_f\) ('lock') (analogous to \(\Box_f\))
- \(+f = \text{prompt}_f\) ('key') (analogous to \(\Diamond_f\))
- move = cancelling \(\text{prompt}_f\) and \(\text{control}_f\) against each other (analogous to \(\Diamond_f \Box_f A \rightarrow A\))
- functional categories serve as 'glue' for the composition
meanings of lexical items are not completely natural

\[ \text{gilgamesh NP -case (control}_{\text{case}} \ h (\lambda \overline{x}.(\overline{x} \text{gilgamesh}) \ h)) \]
\[ \epsilon = v \ T + \text{case} \ \lambda A. (\text{prompt}_{\text{case}} (\lambda \overline{v} \lambda k. (\overline{v} \ k) \ A)) \]

- types for control and prompt
- to limit the invoked context by prompt is actually not necessary, because move always applies at toplevel
Semantic expressions

Semantic values of lexical items will be expressions of a CPS call-by-value $\lambda$-calculus extended with a family of control operators.

**Semantic expressions:**

$$V ::= x \mid k \mid \lambda x.E \mid \lambda k.E$$

$$E ::= V \mid (E E) \mid (C_L \lambda k.E)$$

$C$ is a syntactic constructor that takes an $A$-computation (value) and gives an $A$-computation (term).

**Local evaluation contexts:**

$$C[] ::= [] \mid C[(E C[])] \mid C[(C[] V)] \mid C[C_L C[]]$$
The denotation of an expression of category $A$ is of type $(A \rightarrow R) \rightarrow R = C_A$ (an $A$-computation).

$$\text{john \ NP } \lambda \overline{x}.(\overline{x} \text{john})$$
The denotation of an expression of category $A$ is of type $(A \rightarrow R) \rightarrow R = C_A$ (an $A$-computation).

$$\text{john} \ 	ext{NP} \ \lambda x. (\overline{x} \text{john})$$

The denotation of an expression of category $\text{=A B}$ is of type $C_A \rightarrow C_B$.

$$\text{likes} = \text{NP} \ \text{V} \ \lambda \overline{\text{NP}} \lambda \overline{R}. (\overline{\text{NP}} \ \lambda x. (\overline{R} \ (\text{like } x)))$$

$$\epsilon = \text{V} = \text{NP} \ \text{v} \ \lambda \overline{\text{V}} \lambda \overline{\text{NP}} \lambda k. (\overline{\text{V}} \ \lambda R. (\overline{\text{NP}} \ \lambda x. (k \ (R \ x))))$$

Systematically, we get a direct correspondence between syntactic categories and semantic types.
Licensee features $-f$ introduce $C_f$:
If $\llbracket \alpha \rrbracket = M$, then $\llbracket \alpha - f \rrbracket = (C_f M)$.

\[
\text{john NP -case } (C_{\text{case}} \lambda \overline{x}.(\overline{x} \text{john}))
\]

Licensor features $+f$ have no semantic effect.
Semantics of merge and move

- merge is functional application:

\[
[\text{merge } \alpha \beta] = ([\alpha] [\beta]) \quad (\text{where } \alpha \text{ selects } \beta)
\]
Semantics of merge and move

- **merge** is functional application:
  \[
  \llbracket \text{merge } \alpha \beta \rrbracket = (\llbracket \alpha \rrbracket \llbracket \beta \rrbracket) \quad \text{(where } \alpha \text{ selects } \beta)\]

- **move** provides a toplevel reduction rule for \( C \):
  \[
  \llbracket \text{move } f \ C[(C_f \ E)]] = \lambda k.(E \ \lambda x.(C[\lambda \bar{x}.(\bar{x} \ x)] \ k))
  \]
  where \( x \) is a fresh variable

  e.g.: \( \llbracket C \rrbracket (\llbracket T \rrbracket (\llbracket \nu \rrbracket (\llbracket \text{likes} \rrbracket C_{wh}[\text{whom}])) \llbracket \text{gilgamesh} \rrbracket)))\)

  \( \triangleright \lambda k.(\llbracket \text{whom} \rrbracket \ \lambda x.(\llbracket C \rrbracket (\llbracket T \rrbracket (\llbracket \nu \rrbracket (\llbracket \text{likes} \rrbracket \lambda \bar{x}.(\bar{x} \ x)))) \llbracket \text{gilgamesh} \rrbracket) k)\)
Lexicon

Enkidu:  \textit{gilgamesh} NP -case \\
\((C_{\text{case}} \lambda x.(\overline{x} \textit{gilgamesh}))\)

smiles:  \textit{smiles} V \\
\(\lambda R.((\overline{R} \textit{smile}))\)

\(\nu^0: \quad \epsilon =V =\text{NP} \ \nu \)
\(\lambda \overline{V} \lambda \overline{NP} \lambda k.((\overline{V} \lambda R.((\overline{NP} \lambda x.(k (R x)))))\)

\(T^0: \quad \epsilon =v \ T +\text{case} \)
\(\lambda \overline{V} \lambda k.(\overline{V} \ k)\)

\(C^0: \quad \epsilon =T \ C \)
\(\lambda \overline{T} \lambda k.(k (\overline{T} \lambda s.s))\)
+\textit{f} feature is semantically void

moving phrases would require an additional rule like:

\[
((C_f M) N) \triangleright (C_f (M N))
\]

variation of evaluation time and place possible with a more fine-grained feature system
Extension: Feature system

A more fine-grained feature system could distinguish:

- strong vs weak features
- interpretable vs uninterpretable features

This would allow for the following possibilities:

- immediate evaluation in base position (uninterpretable features)
- delayed evaluation in base position (weak interpretable feature)
- delayed evaluation at landing site (strong interpretable feature)