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.)

Minimalism in MMCG

Correspondences:

- categorial features = slash types,
  categorial feature checking = Modus Ponens (/,\-elimination)
- formal features = unary operators $\Diamond, \Box$
  formal feature checking = $\Diamond\Box A \rightarrow A$
- 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

Minimalist Grammars

A Minimalist Grammar is a four-tuple $G = \langle V, Cat, Lex, \mathcal{F} \rangle$ 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
- $\mathcal{F} = \{\texttt{merge}, \texttt{move}\}$ 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:

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

Category features:

$cat = (sel^*)\, base$ where

- $sel ::= \ =base$
- $base ::= \ \texttt{NP} \mid \texttt{V} \mid \texttt{v} \mid \texttt{T} \mid \texttt{C} \mid \texttt{Det} \mid \texttt{N}$

Minimalist expressions as trees

Simple and complex expressions:

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

Category features:

$cat = (sel^*)\, base$ where

- $sel ::= {=}base$
- $base ::= \texttt{NP} \mid \texttt{V} \mid \texttt{v} \mid \texttt{T} \mid \texttt{C} \mid \texttt{Det} \mid \texttt{N}$

Formal features:

$lic ::= +L \mid -L$ where $L = \texttt{case} \mid \texttt{wh} \mid \ldots$

Structure-building operations 1: merge

Merge cancels selector and selected features.

First merge of a simple head α and a simple (complex) complement β:

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

Structure-building operations 1: merge

Merge cancels selector and selected features.

First merge of a simple head α and a simple (complex) complement β:

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

Second merge of a simple (complex) specifier β and a complex expression α containing a selecting head:

$\texttt{merge2}$
$\beta@(\texttt{beta}, \texttt{f}, F_2, N\ (, e_1, e_2)\ )\ \alpha@(\texttt{alpha}, =\texttt{f}\gamma, F_1, M, e_3, e_4) = (\texttt{beta}\,\texttt{alpha}, \gamma, F_1, (M\ N), \beta', \alpha')$

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:

$$\texttt{move f}\ (\texttt{alpha}, \gamma_1, F_1, M, e_1, e_2) = \\ (\texttt{beta alpha}, \gamma_1, F_1 \backslash \{\texttt{+f}\}, [\![\texttt{move}\ M]\!], \beta', e_1 e_2')$$

for a feature `f` such that `+f`$\in F_1$ and e_2 contains exactly one expression β@(`beta`, γ_2, F_2, N) with `-f`$\in F_2$

Example

Numeration:

gilgamesh	NP -case
whom	NP -wh
liked	=NP V
ϵ	=V =NP v
ϵ	=v T +case
ϵ	=T C +wh

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

- $\mathcal{C}$, $\mathcal{F}$ (Felleisen & Friedman)
- `control` and `prompt` (Felleisen & Sitaram)
- `shift` and `reset` (Danvy & Filinsky)
- `cupto` and `set` (Gunter, Rémy & Riecke)
- ...

`control` and `prompt`: Syntax

Call-by-value λ-calculus extended with control operators:

$$V ::= x \mid \lambda x.E$$
$$E ::= V \mid (E\ E) \mid \texttt{control}\ f\ E \mid \texttt{prompt}\ E$$

Evaluation contexts:

$$C[\,] ::= [\,] \mid C[(E\ [\,])] \mid C[([\,]\ V)]$$
$$M[\,] ::= [\,] \mid M[\texttt{prompt}\ C[\,]]$$

`Control` captures the current continuation, which is delimited by the closest enclosing `prompt`.

control and prompt: Operational semantics

$$M[(\texttt{prompt}\ V)] \rhd M[V]$$

$$M[(\texttt{prompt}\ C[(\texttt{control}\ h\ E)])] \rhd M[(\texttt{prompt}\ E')]$$

where $E' = E\{h \mapsto \lambda x.C[x]\}$

control and prompt: Operational semantics

$$M[(\texttt{prompt}\ V)] \triangleright M[V]$$

$$M[(\texttt{prompt}\ C[(\texttt{control}\ h\ E)])] \triangleright M[(\texttt{prompt}\ E')]$$

where $E' = E\{h \mapsto \lambda x.C[x]\}$

Example:

```
    (+ 5 (prompt (* 2 (control h (+ 1 (h 3))))))
▷   (+ 5 (prompt (+ 1 (* 2 3))))
▷   (+ 5 (prompt 7))
▷   (+ 5 7)
```

First idea

- `merge` = functional application (= modus ponens)
- `-f` = $\texttt{control}_\texttt{f}$ ('lock') (analogous to $\Box_f$)
- `+f` = $\texttt{prompt}_\texttt{f}$ ('key') (analogous to $\Diamond_f$)
- `move` = cancelling $\texttt{prompt}_\texttt{f}$ and $\texttt{control}_\texttt{f}$ against each other (analogous to $\Diamond_f \Box_f A \rightarrow A$)
- functional categories serve as 'glue' for the composition

But...

- meanings of lexical items are not completely natural

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

- 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 λ-calculus extended with a family of control operators.

Semantic expressions:

$$V ::= x \mid k \mid \lambda x.E \mid \lambda k.E$$
$$E ::= V \mid (E\ E) \mid (\mathcal{C}_L\ \lambda k.E)$$

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

Local evaluation contexts:

$$C[] ::= [] \mid C[(E\ C[])] \mid C[(C[]\ V)] \mid C[\mathcal{C}_L\ C[]]$$

Semantics of lexical items

- The denotation of an expression of category `A` is of type $(A \to R) \to R = C_A$ (an A-computation).

  `john NP` $\lambda\overline{x}.(\overline{x}\ \mathit{john})$

Semantics of lexical items

- The denotation of an expression of category `A` is of type $(A \to R) \to R = C_A$ (an A-computation).

  `john` NP $\lambda\overline{x}.(\overline{x}\ \mathit{john})$

- The denotation of an expression of category `=A B` is of type $C_A \to C_B$.

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

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

Systematically, we get a direct correspondence between syntactic categories and semantic types.

Semantics of lexical items

- Licensee features `-f` introduce $\mathcal{C}_{\mathtt{f}}$:
  If $[\![\alpha]\!] = M$, then $[\![\alpha\, -\mathtt{f}]\!] = (\mathcal{C}_{\mathtt{f}}\ M)$.

  `john NP -case` $(\mathcal{C}_{\mathtt{case}}\ \lambda\overline{x}.(\overline{x}\ \mathit{john}))$

- Licensor features `+f` have no semantic effect.

Semantics of merge and move

- merge is functional application:

$$[\![\texttt{merge}\ \alpha\ \beta]\!] = ([\![\alpha]\!]\ [\![\beta]\!]) \quad \text{(where } \alpha \text{ selects } \beta\text{)}$$

Semantics of merge and move

- merge is functional application:

  $[\![\texttt{merge}\ \alpha\ \beta]\!] = ([\![\alpha]\!]\ [\![\beta]\!])$ (where α selects β)

- move provides a toplevel reduction rule for $\mathcal{C}$:

  $[\![\texttt{move f}\ C[(\mathcal{C}_{\texttt{f}}\ E)]]\!] = \lambda k.(E\ \lambda x.(C[\lambda\overline{x}.(\overline{x}\ x)]\ k))$

  where x is a fresh variable

e.g.: $([\![C]\!]\ ([\![T]\!]\ (([\![v]\!]\ ([\![\texttt{likes}]\!]\ \mathcal{C}_{\texttt{wh}}[\![\texttt{whom}]\!]))\ [\![\texttt{gilgamesh}]\!])))$
$\triangleright\ \lambda k.([\![\texttt{whom}]\!]\ \lambda x.(([\![C]\!]\ ([\![T]\!]\ (([\![v]\!]\ ([\![\texttt{likes}]\!]\ \lambda\overline{x}.(\overline{x}\ x)))\ [\![\texttt{gilgamesh}]\!])\ k)$

Lexicon

Enkidu: `gilgamesh NP -case`
$(\mathcal{C}_{\mathsf{case}}\ \lambda\overline{x}.(\overline{x}\ \mathit{gilgamesh}))$

smiles: `smiles V`
$\lambda\overline{R}.(\overline{R}\ \mathit{smile})$

v^0: ϵ `=V =NP v`
$\lambda\overline{\overline{V}}\lambda\overline{\overline{NP}}\lambda k.(\overline{\overline{V}}\ \lambda R.(\overline{\overline{NP}}\ \lambda x.(k\ (R\ x))))$

T^0: ϵ `=v T +case`
$\lambda\overline{\overline{v}}\lambda k.(\overline{\overline{v}}\ k)$

C^0: ϵ `=T C`
$\lambda\overline{\overline{T}}\lambda k.(k\ (\overline{\overline{T}})\ \lambda s.s))$

Implications

- `+f` feature is semantically void
- moving phrases would require an additional rule like:

$$((\mathcal{C}_{\mathtt{f}}\ M)\ N) \rhd (\mathcal{C}_{\mathtt{f}}\ (M\ N))$$

- 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)