On the Proper Treatment of Context in NL

Jan van Eijck
CWI & ILLC, Amsterdam, Uil-OTS, Utrecht

Abstract

The proper treatment of quantification in Natural Language proposed by Richard Montague some thirty years does not do proper justice to the fact that interpretation of texts both uses context and sets up new contexts. The dynamic turn in NL semantics is the attempt to model this basic fact, but the use of dynamically quantified variables introduces an undesirable element into this attempt. By extending a variable-free 'incremental dynamics' with a flexible system of type scheme patterns and type scheme pattern matching, we arrive at a Montague style architecture for NL semantics that provides a proper treatment both of quantification and of context use and context change.

1 The Key Issue

When processing natural language text, a *context* is incrementally built up of salient items, to be used as a domain for fixing anaphoric references (with constraints on where the referents are to be found). This context is *used* and *extended* at the same time, for introduction of new topics of conversation makes the context grow, and pronouns get interpreted (resolved) by linking them to the existing context.

```
txt txt txt him↑ txt txt she↑ txt
txt txt him↑ txt txt txt she↑ txt
txt a man↓ txt txt txt his↑ txt
txt [every woman↓ txt txt txt her↑]
txt txt a man↓ txt txt his↑ txt txt
another↑ man↓ txt txt txt his↑ txt
txt a boy↓ txt txt his↑ txt txt txt
a man↓ txt [txt txt himself↑] txt
txt txt
```

Legend:

- ↑: looking for a referent from surrounding text or outside context.
- ↓: adding a new referent to the existing context.
- [↓ txt txt txt] : local extension of context (the context is extended, but this extension has limited scope).
- [txt txt txt ↑] : locally extended context where antecedent may be found (the local extension of context provides additional possible referents).

The first theories that tried to give a systematic and more or less formal account of context dynamics were Discourse Representation Theory (Kamp, 1981) and File Change Semantics (Heim, 1982). Attempts to incorporate this work in mainstream Montague style natural language semantics can be found in (Groenendijk and Stokhof, 1990; Muskens, 1994; van Eijck, 1997; van Eijck and Kamp, 1997; Kohlhase et al., 1996). What these attempts at incorporation have in common is the representation of context as a list of variables, and use of dynamic quantification over these variables as context update. A serious disadvantage of dynamic quantification is that update of a variable y makes the previous value of y inaccessible: dynamic quantification over named variables brings the problem of destructive assignment in its wake.

One way of avoiding the destructive assignment problem is by dispensing with dynamic variable names and handling argument binding in a way reminiscent of the so-called De Bruijn indices from lambda calculus (de Bruijn, 1980). This leads to a redesign of the dynamic logic paradigm for handling context and context change, as follows. In incremental dynamics (van Eijck, 2000), context gets represented as a stack of n objects: referents for the anaphora $\mathrm{ANA}_0, \ldots, \mathrm{ANA}_{n-1}$. These contextually given objects need not all be different. Context gets processed dynamically: context extensions may be temporary. Of course, the anaphoric elements do not occur with an index in the text, as ANA_i; rather, the choice of appropriate indices for the anaphoric elements constitutes the process of anaphora resolution.

Text processing gets viewed as a process that adds a number of referents (say m) to the context: after processing the new text in a context of size n we have a new context of size $n + m$. Contextual elements that were mentioned too long ago may lose their salience (or: drop out of the context), but we will

not make this a central issue in the present paper.

Thus, the question *What do the indices in natural language texts like (1) mean?* has as its answer: just a gloss to indicate how we suppose the anaphoric reference resolution mechanism links the pronoun from the second sentence to the noun phrase introduced in the first.

1 *A man^i walked in. He$_i$ smiled.*

An anaphoric context is just a stack of n objects available as antecedents in future discourse. Introducing a new topic of conversation extends the anaphoric context by putting a new object on top of the context stack. The dynamic existential quantifier $\exists$ gets interpreted as the action of putting a new object on top of the context stack. If the size of the context is known, there is no need to indicate the register that gets bound by $\exists$. If the context is c, and its size is n, then c has the form $(c_0, \ldots, c_{n-1})$, and the next register — the one that gets bound by $\exists$ — is c_n.

A central idea is that the dynamic quantifier does not name the variable that it binds, but that dynamic quantification is always quantification in context. If a context is given, the interpretation of $\exists$ is just: introduce a next topic of conversation, and add it to the context.

2 Types for Incremental Dynamics

A typed version of DRT and DPL can be built on basic types e for entities and T for state transitions. The shift from t (truth values) to T (state transitions) constitutes the dynamic turn in natural language semantics. The state transitions can themselves be viewed as relations between states, so in a more fine-grained set-up, with basic types for t for truth values and s for states, state transitions are of type $s \to s \to t$.

A typed version of incremental dynamics (henceforth: ID) will use basic types e for entities, t for truth values, and types for contexts of arbitrary finite sizes. We view the natural number k in Von Neumann style as $k = \{0, \ldots, k-1\}$, and we use $[k]$ for the type of a context consisting of k elements. E.g., $5 \to [5]$ is the type of an index function into a context of length 5, and $[5] \to [8] \to t$ the type of a stack transition that extends a stack of length 5 by 3 positions to a stack of length 8. We will abbreviate this type as $(5, 3)$. Here is the definition of the type system:

$$\begin{array}{rcl} N & ::= & 0 \mid 1 \mid 2 \mid \cdots \\ \text{Type} & ::= & e \mid t \mid N \mid [N] \mid \text{Type} \to \text{Type} \end{array}$$

Abbreviation: use (N_1, N_2) for

$$[N_1] \to [N_1 + N_2] \to t.$$

3 Index Variables and Type Schemes

To represent a context of arbitrary size i index variables are used. $(i, 3)$ is the type scheme of a stack transition that extends a stack of length i by 3 positions. To represent extension of context by an indeterminate number of elements, we add numerical variables. $(i + 1, J)$ is the type scheme pattern of a stack transition that extends a non-empty context by J positions.

To generalize over stack transitions, we introduce type schemes. A type scheme is a type with index variables in it. Thus, $(i + 1, 2)$ is (abbreviated notation for) a type scheme. It describes the general form of a stack transition that increments a stack of at least size 1 by 2 positions. Examples:

- $i \to [i]$ is the type of an index (function) into a context,
- $i + 1 \to [i + 1]$ is the type of an index into a non-empty context,
- $i + 1 \to (i + 1, 2)$ the type of an index into a stack transformer with a non-empty input that puts 2 new items on the stack.

We use $\triangleright T$ as an abbreviation for an index into T, leaving the type of the index to be understood from the context of use.

4 Pattern Variables and Type Scheme Patterns

We distinguish between index variables and pattern variables. We use i, j, k for index variables, I, J, K for pattern variables.

A type containing pattern variables is called a *type pattern*. A type scheme containing pattern variables is called a *type scheme pattern*.

- $i + 1 \to (i + 1, J)$ is the type scheme pattern of an index into a stack transformer with a non-empty input that puts J new items on the stack.

Replacing the pattern variables in a type (scheme) pattern **instantiates** the type (scheme) pattern to a type (scheme).

- $K := 3$ instantiates the type pattern $(3, K)$ to the type $(3, 3)$.
- $J := 2$ instantiates the type scheme pattern $i + 1 \to (i + 1, J)$ to the type scheme $i + 1 \to (i + 1, 2)$.

5 Pattern Variables Versus Index Variables

- Pattern variables (Pvar) are variables that stand proxy for natural numbers. Full instantiation of a type (scheme) pattern involves replacement of all pattern variables by (names of) natural numbers.

- Type patterns and type scheme patterns are used to express patterns of types or patterns of type schemes. E.g., (i, J) is the pattern of all stack transitions that extend the stack by J elements. Instantiations of (i, J) are $(i, 0)$, $(i, 1)$, $(i, 2)$, and so on.
- Index variables (Ivar) are variables that may occur in the type schemes that result from fully instantiating the pattern variables in a type scheme pattern. $(i, 1)$ is the type scheme of a stack transition that extends the stack by one position.

6 Type Scheme Patterns: Definition

$$
\begin{array}{rcl}
N & ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\text{Type} & ::= & e \mid t \mid N \mid [N] \mid \text{Type} \to \text{Type} \\
\text{Num} & ::= & N \mid \text{Pvar} \mid \text{Num}_1 + \text{Num}_2 \\
\text{Nexp} & ::= & \text{Num} \mid \text{Ivar} + \text{Num} \\
\text{Tsp} & ::= & e \mid t \\
 & \mid & \text{Nexp} \mid [\text{Nexp}] \\
 & \mid & \text{Tsp} \to \text{Tsp}
\end{array}
$$

- A type (scheme) T is an instantiation of a type (scheme) pattern T' if there is a pattern variable substitution σ such that $T = T'\sigma$. In this case, σ will map all pattern variables in T' to natural numbers.
- A type scheme pattern T is more general than a scheme T' if there is a substitution σ for the index and pattern variables such that $T' = T\sigma$.
- Thus, $i \to (i, 2)$ is more general than $j + 1 \to (j+1, 2)$, for the substitution $\{i := j+1\}$ transforms the former into the latter.
- Substitution σ is an mgu (most general unifier) of type schemes T and T' if
  - there is a substitution σ such that $T\sigma = T'\sigma$,
  - every substitution θ such that $T\theta = T'\theta$ is such that $T\sigma$ is more general than $T\theta$.

7 Unification of Numerical Terms: Caution!

We are not interested in the term model generated from the natural numbers by the operation $+$, but in the natural numbers themselves. In the term model, $(1 + 4)$, $(4 + 1)$ and $(2 + 3)$ are all different, while in fact all these terms denote the same natural number.

If we work in the term model, we can unify $(N + M) \approx (K + L)$, with N, M, K, L all variables, by means of $N := K, M := L$, but for the natural numbers this substitution may well be wrong, for as we know, decomposition of natural numbers into summands is not unique.

We must conclude that unification of numerical terms will only work in special cases: we call terms that can be unified *unifying pairs*, pairs that cannot *failing pairs*. Caution: in-between cases exist!

8 Simplified Forms of Numerical Term Pairs

If Nexp_1 and Nexp_2 are numerical terms, then the pair $\text{Nexp}_1 \approx \text{Nexp}_2$ can be written in a canonical form as follows:

- Collect all natural numbers occuring in the lhs term and add them up, giving n.
- Collect all natural numbers occuring in the rhs term and add them up, giving m.
- Subtract the difference $|n - m|$ from both sides of the pair.
- If lhs and rhs both consist of more than a single variable, delete all variables that occur on both sides.

With this recipe, a pair $\text{Nexp}_1 \approx \text{Nexp}_2$ can always be simplified to one of the following forms (the variables v_i and w_j range over index and pattern variables, with each term containing at most one index variable; if lhs and rhs both consist of more than a single variable then they have no variables in common):

- $n \approx m$, with $n \geq 0, m \geq 0$.
- $v_1 + \cdots + v_n \approx k$, with $n > 0, k \geq 0$,
- $k \approx v_1 + \cdots + v_n$, with $n > 0, k \geq 0$,
- $v_1 + \cdots + v_n \approx w_1 + \cdots + w_m$, with $(n > 0, m > 0)$
- $v_1 + \cdots + v_n \approx w_1 + \cdots + w_m + k$, with $(n > 0, m > 0, k > 0)$
- $v_1 + \cdots + v_n + k \approx w_1 + \cdots + w_m$, with $(n > 0, m > 0, k > 0)$

9 Failing Pairs

- A pair of the form $n \approx m$ fails if $n \neq m$.
- A pair of the form $v \approx w_1 + \cdots + w_m + k$, with $k > 0$ fails if v occurs among the w_j.
- A pair of the form $v_1 + \cdots + v_n + k \approx w$, with $k > 0$ fails if w occurs among the v_i.

10 Unifying pairs, With Their Substitutions

We use ϵ for the empty substitution (the substitution that maps every term to itself).

$$\frac{v \approx w}{\epsilon}\ v \equiv w \qquad \frac{v \approx w}{\{v := w\}}\ v \not\equiv w$$

$$\frac{v \approx k}{\{v := k\}} \qquad \frac{k \approx w}{\{w := k\}}$$

$$\frac{v \approx w_1 + \cdots + w_m}{\{v := w_1 + \cdots + w_m\}}\ v \not\equiv w_j$$

$$\frac{v_1 + \cdots + v_m \approx w}{\{w := v_1 + \cdots + v_n\}}\ w \not\equiv v_i$$

$$\frac{v \approx w_1 + \cdots + w_m + k}{\{v := w_1 + \cdots + w_m + k\}}\ v \not\equiv w_j$$

$$\frac{v_1 + \cdots + v_m + k \approx w}{\{w := v_1 + \cdots + v_n + k\}}\ w \not\equiv v_i$$

$$\frac{v \approx w_1 + \cdots + v + \cdots + w_m}{\{w_1 := 0, \ldots, w_m := 0\}}$$

$$\frac{v_1 + \cdots + w + \cdots + v_n \approx w}{\{v_1 := 0, \ldots, v_n := 0\}}$$

11 Unification Algorithm for Tsp's (Sketch)

- e unifies with e, with mgu ϵ, t unifies with t, with mgu ϵ.
- Nexp_1 unifies with Nexp_2 and gives mgu according to the rules above.
- $[\text{Nexp}_1]$ unifies with $[\text{Nexp}_2]$ if Nexp_1 unifies with Nexp_2 and gives mgu according to the rules above.
- $\text{Tsp}_1 \to \text{Tsp}_2$ unifies with $\text{Tsp}_3 \to \text{Tsp}_4$ if Tsp_1 unifies with Tsp_3 to give mgu θ, and $\text{Tsp}_2\theta$ unifies with $\text{Tsp}_4\theta$ to give mgu σ, and gives mgu $\theta\sigma$.
- No other pairs of Tsp's unify.

12 Facts about Tsp Unification

- The algorithm always terminates.
- The algorithm is sound, but not complete (it will fail to find solutions in cases of comparison of numerical term pairs that are neither unifying nor failing).
- The algorithm will never introduce more than one index variable in a numerical expression. (This follows from a straightforward inspection of the rules.)

13 Example Type Scheme Patterns

Dynamic Exists

$$\exists \ :: \ (i+1, J) \to (i, J+1)$$

$\exists$ is a function that maps a stack transformer of type $(i+1, J)$, i.e., a transformer for a context with at least one element, to a stack transformer that expects a context with one element less, and increments this context by one element more. Note that $(i+1, J) \to (i, J+1)$ in fact specifies the **pattern** of a type scheme rather than a type scheme. For every choice of a natural number for J, we get a particular instantiation of the pattern to a scheme. In these type schemes for $\exists$ there occurs just one type variable, namely i.

Dynamic Negation

$$\neg \ :: \ (i, J) \to (i, 0)$$

$\neg$ maps a stack transformer to a test (a transformer that does not increment the stack).

Context Composition

$$(;) \ :: \ (i, J) \to (i + J, K) \to (i, J + K)$$

; takes as its arguments two stack transformers of which the second handles the output of the first, and combines these two into a new stack transformer, with increment given by the sum of the increments of the components.

14 Expressing properties in ID

Assume a constant for the property of being a man:

$$\text{man} \ :: \ e \to t$$

We can use this constant of type $e \to t$ to construct an index into a stack transformer, i.e, an object of type $i + 1 \to (i + 1, 0)$. This is the type of index functions into $[i+1] \to [i+1] \to t$, i.e., into tests on non-empty stacks.

$$\lambda j_{i+1} \lambda c_{[i+1]} \lambda c'_{[i+1]} \cdot \\ (\text{man}\ c_j \wedge c = c') \ :: \ i + 1 \to (i + 1, 0)$$

Abbreviation: if $c \ :: \ [n]$, and $i \ :: \ \boldsymbol{n}$, we use c_i for $(c\ i)$. Also, we abbreviate $\boldsymbol{i} + \mathbf{1} \to (i + 1, 0)$ as $\triangleright(i + 1, 0)$.

15 Definition of $\exists$

We extend the usual logic for extensional type theory with a constant $[] \ :: \ [0]$ and an operation $(\hat{}) \ :: \ [i] \to e \to [i+1]$. The constant [] denotes the empty stack, and the operation $\hat{}$ denotes 'extending a list by one element' or 'putting a new item on top of a stack'. We write $\hat{}$ with infix notation, so if $c \ :: \ [i]$ and $x \ :: \ e$ then $c\hat{}x \ :: \ [i + 1]$. We use this to give the definition of $\exists$, as follows. $\exists$ is an abbreviation of:

$$\lambda P_{(i+1,J)} \lambda c_{[i]} \lambda c'_{[i+J+1]} . \exists x_e ((P\ c\hat{}x)\ c').$$

Note: $i + J + 1$ is of the general form $i + K$, with i an index variable and K a numerical expression containing no index variables.

16 Definitions of $\neg$ and ;

$\neg$ is an abbreviation of:

$$\lambda P_{(i,J)}\lambda c_{[i]}\lambda c'_{[i]}.(\neg\exists c''_{[i+J]}((P\ c)\ c'') \wedge c = c').$$

; is an abbreviation of:

$$\lambda P_{(i,J)}\lambda Q_{(i+J,K)}\lambda c_{[i]}\lambda c'_{[i+J+K]}.\ \exists c''_{[i+J]}(((P\ c)\ c'') \wedge ((Q\ c'')\ c')).$$

We will write ; as an infix operator with association to the left, and we will omit superfluous brackets.

17 Linking Formulas to Their Type Schemes

Index variables can serve as a bridge between formulas and their type schemes. To talk about the final position of an arbitrary non-empty stack, we refer to the stack by means of the type $[i+1]$, and use index i in the formula to access its final position.

This is at the heart of the incremental dynamics of the indefinite determiner. A sentence starting with an indefinite determiner *a* has the general form [[a CN] VP]. Its semantics is a context transformation that takes an arbitary context, say of length i, adds one element to it to produce a new context of length $i+1$, makes sure that *that element* satisfies the CN and the VP, and performs the context transformations associated with the CN and the VP. We use index expression i in the formula to refer to *that element*.

18 Translating the Indefinite Determiner

a $\rightsquigarrow$
$\lambda P_{\triangleright(i+1,N)}\lambda Q_{\triangleright(i+N+1,M)}.\exists\ (Pi\ ;\ Qi)$
$:: \triangleright(i+1,N) \to \triangleright(i+N+1,M) \to (i, N+M+1).$

We need a pattern here because we do not know in advance how many referents will be introduced within the CN that goes with the indefinite determiner and how many in the predicate that follows.

Because $Pi :: (i+1,N)$ and $Qi :: (i+N+1,M)$, we must instantiate the type scheme of ; to $(i+1,N) \to (i+N+1,M) \to (i+1,N+M)$, and we get that $(Pi\ ;\ Qi) :: (i+1,N+M)$.

Since $\exists :: (i+1,J) \to (i,J+1)$ we must unify the schemes $(i+1,N+M)$ and $(i+1,J)$ to make the function fit the argument. This instantiates the type of $\exists$ to $(i+1,N+M) \to (i,N+M+1)$, and we get that $\exists\ (Pi\ ;\ Qi) :: (i,N+M+1)$.

19 Function Application with Unification

Function application may involve unification of type schemes, as follows:

$$\frac{\varphi :: T_1 \to T_2 \quad \psi :: T_3}{(\varphi\theta\ \psi\theta) :: T_2\theta}\ \theta \text{ mgu of } T_1, T_3$$

For example, let $T_1 = \triangleright(i+1,J)$, $T_2 = \triangleright(i,J+1)$, $T_3 = (k+1,0)$. Then

$$\lambda P_{\triangleright(i+1,J)}\lambda c\lambda c'\exists x.Pic\hat{}x\ c' :: T_1 \to T_2$$

applied to

$$\lambda j\lambda c\lambda c'Mc_j \wedge c = c' :: T_3$$

yields, under substitution $\theta = \{i := k, J := 0\}$:

$$(\lambda P_{\triangleright(k+1,0)}\lambda c\lambda c'\exists x.\ Pkc\hat{}x\ c)(\lambda j\lambda c\lambda c'Mc_j \wedge c = c') :: (k,1).$$

Note that the substitution $\theta = \{i := k, J := 0\}$ affects both the type scheme and the formula.

20 Toy Fragment: Determiners, Nouns and Intransitive Verbs

a $\rightsquigarrow$
$\lambda P_{\triangleright(i+1,J)}\lambda Q_{\triangleright(i+J+1,K)}.\exists(Pi;Qi)$
$:: \triangleright(i+1,J) \to \triangleright(i+J+1,K) \to (i, J+K+1)$

every $\rightsquigarrow$
$\lambda P_{\triangleright(i+1,J)}\lambda Q_{\triangleright(i+J+1,K)}.\neg\exists(Pi;\neg Qi)$
$:: \triangleright(i+1,J) \to \triangleright(i+J+1,K) \to (i,0)$

no $\rightsquigarrow$
$\lambda P_{\triangleright(i+1,j)}\lambda Q_{\triangleright(i+J+1,K)}.\neg\exists(Pi;Qi)$
$:: \triangleright(i+1,J) \to \triangleright(i+J+1,K) \to (i,0)$

man $\rightsquigarrow$
$\lambda j\lambda c_{[i+1]}\lambda c'_{[i+1]}.(\text{man}\ c_j \wedge c = c')\ :: \triangleright(i+1,0)$

smiled $\rightsquigarrow$
$\lambda j\lambda c_{[i+1]}\lambda c'_{[i+1]}.(\text{smile}\ c_j \wedge c = c')\ :: \triangleright(i+1,0)$

21 Iota Reduction

If A is a list of i elements and we append a new element B to the list, then we can retrieve this element by lookup at index $i+1$. This motivates the following notion of ι reduction:

- If $A :: [i]$ and $B :: e$, then $((A\hat{}B)\ i) \Rightarrow_\iota B$.

In abbreviated notation: $(A\hat{}B)_i \Rightarrow_\iota B$. We also allow ι reduction in context, and we use $\Rightarrow_\iota$ for one-step ι reduction. Here is an example:

$$\lambda x.((c_{[5]}\hat{}x)\ 5) \Rightarrow_\iota \lambda x.x$$

Or with variables:

$$\lambda x.((c_{[i]}\hat{}x)\ i) \Rightarrow_\iota \lambda x.x$$

22 Reduction to Normal Form

- Beta reduction: defined in the standard way.
- Iota reduction: see above.
- Beta-iota reduction is confluent, i.e, if $E \stackrel{\beta\iota}{\twoheadrightarrow} F$ and $B \stackrel{\beta\iota}{\twoheadrightarrow} F'$ then there is a G with $F \stackrel{\beta\iota}{\twoheadrightarrow} G$ and $F' \stackrel{\beta\iota}{\twoheadrightarrow} G$.
- Beta-iota reduction is strongly normalizing, i.e., every reduction sequence $E \stackrel{\beta\iota}{\longrightarrow} F \stackrel{\beta\iota}{\longrightarrow} G \ldots$ terminates.
- Thus, beta-iota reduction yields unique normal forms.

23 Example: 'a man smiled'

$$
\begin{array}{l}
\text{a man} \rightsquigarrow \\
(\lambda P_{\triangleright(i+1,J)} \lambda Q_{\triangleright(i+J+1,K)}.\exists(Pi;Qi)) \\
\quad (\lambda j \lambda c_{[i+1]} \lambda c'_{[i+1]}.(\text{man } c_j \wedge c = c')) \\
\Rightarrow_\beta \quad \lambda Q_{\triangleright(i+1,K)}.\exists((\lambda c_{[i+1]} \lambda c'_{[i+1]}. \\
\quad (\text{man } c_{i+1} \wedge c = c'));Qi) \\
\Rightarrow_\beta \quad \lambda Q_{\triangleright(i+1,K)}.\exists(\lambda c_{[i+1]} \lambda c'_{[i+K+1]}. \\
\quad (\text{man } c_{i+1} \wedge (((Qic)c'))) \\
\Rightarrow_\beta \quad \lambda Q_{\triangleright(i+1,K)}.\lambda c_{[i]} \lambda c'_{[i+K+1]}. \\
\quad \exists x_e(\text{man } (c\hat{}x)_{i+1} \wedge (((Q(i{+}1))c\hat{}x)c')) \\
\Rightarrow_\iota \quad \lambda Q_{\triangleright(i+1,K)}.\lambda c_{[i]} \lambda c'_{[i+K+1]}. \\
\quad \exists x_e(\text{man } x \wedge ((Qi)c\hat{}x)c')
\end{array}
$$

$$
\begin{array}{l}
\text{a man smiled} \rightsquigarrow \\
(\lambda Q_{\triangleright(i+1,K)}.\lambda c_{[i]} \lambda c'_{[i+K+1]}. \\
\quad \exists x_e(\text{man } x \wedge (((Qi)c\hat{}x)c'))) \\
\quad (\lambda j \lambda c_{[i+1]} \lambda c'_{[i+1]}.(\text{smile } c_j \wedge c = c')) \\
\Rightarrow_{\beta\iota} \quad \lambda c_{[i]} \lambda c'_{[i+1]}. \\
\quad \exists x_e(\text{man } x \wedge \text{smile } x \wedge c\hat{}x = c')
\end{array}
$$

24 Anaphora Resolution

Construction of an anaphora resolution engine is outside our scope. But the present framework makes it easy to specify **exactly** where anaphora resolution comes in. For every given anaphoric element, the framework specifies **the currently relevant context** for the resolution of that anaphoric element.

The anaphora resolution engine 'res' uses a context plus some unspecified further information to pick an index for that context.

$$\text{res} :: \ [i+1] \rightarrow \_ \rightarrow i+1$$

25 Toy Fragment: Pronouns and Transitive Verbs

$$
\begin{array}{l}
\text{he} \rightsquigarrow \\
\lambda P_{\triangleright(i+1,J)} \lambda c_{[i+1]} \lambda c'_{[i+J+1]}.(((P(\text{res } c \ \_))c)c') \\
:: \ \triangleright(i+1,J) \rightarrow (i+1,J)
\end{array}
$$

$$
\begin{array}{l}
\text{he}_k \rightsquigarrow \\
\lambda P_{\triangleright(i+1,J)} \lambda c_{[i+1]} \lambda c'_{[i+J+1]}.(((Pk)c)c') \\
:: \ \triangleright(i+1,J) \rightarrow (i+1,J)
\end{array}
$$

$$\text{him} \rightsquigarrow \ldots$$

$$\text{him}_k \rightsquigarrow \ldots$$

$$
\begin{array}{l}
\text{loves} \rightsquigarrow \\
\lambda \mathbb{P}_{\triangleright(i+1,J) \rightarrow (i+1,J)} \lambda s \lambda c \lambda c'. \\
(((\mathbb{P}(\lambda o \lambda c'' \lambda c'''(((\text{love } c''_o) \ c''_s) \wedge c'' = c''')))c)c') \\
:: \ (\triangleright(i+1,J) \rightarrow (i+1,J)) \rightarrow \triangleright(i+1,J)
\end{array}
$$

26 Example: 'he$_1$ loves her$_2$'

$$
\begin{array}{l}
\text{loves her}_2 \rightsquigarrow \\
\lambda \mathbb{P} \lambda s \lambda c \lambda c'. \\
(((\mathbb{P}(\lambda o \lambda c'' \lambda c'''(((\text{love } c''_o) \ c''_s) \wedge c'' = c''')))c)c') \\
(\lambda P \lambda c \lambda c'.(((P2)c)c')) \\
\quad \Rightarrow_\beta \quad \lambda s \lambda c \lambda c'.(((\lambda P \lambda c \lambda c'.(((P2)c)c')) \\
\lambda o \lambda c'' \lambda c'''(((\text{love } c''_o) \ c''_s) \wedge c'' = c''')))c)c' \\
\quad \Rightarrow_\beta \quad \lambda s \lambda c \lambda c'.(((\text{love } c_2) \ c_s) \wedge c = c')
\end{array}
$$

$$
\begin{array}{l}
\text{he}_1 \text{ loves her}_2 \rightsquigarrow \\
(\lambda P \lambda c \lambda c'.(((P1)c)c')) \\
(\lambda s \lambda c \lambda c'.(((\text{love } c_2) \ c_s) \wedge c = c')) \\
\quad \Rightarrow_\beta \quad \lambda c \lambda c'.(((\text{love } c_2) \ c_1) \wedge c = c')
\end{array}
$$

27 Flexible Typing: Pronouns and Transitive Verbs Again

In a system with flexible typing, the type $(i+1 \rightarrow (i+1,J)) \rightarrow (i+1,J)$ for pronouns can be lowered to $i+1$. The simplest meaning of pronoun is: an invitation to pick a suitable index from a context. The simplest meaning for an anaphorically resolved pronoun is: an index into the appropriate context.

If we use the type scheme variables to transfer the information about the size of the context, we can get by with the following:

$$\text{res} :: \ [i+1] \rightarrow \_ \rightarrow i+1$$

$$\text{he} \rightsquigarrow \ \lambda c.(\text{res } c \ \_) :: \ [i+1] \rightarrow i+1$$

$$\text{he}_k \rightsquigarrow \ k \ :: \ i+1$$

$$
\begin{array}{l}
\text{loves} \rightsquigarrow \\
\quad \lambda o \lambda s \lambda c \lambda c'.(((\text{love}c_o)c_s) \wedge c = c') \\
\quad :: \ i+1 \rightarrow i+1 \rightarrow (i+1,0)
\end{array}
$$

28 Toy Fragment: Reflexives

In the flexible set-up, where transitive verbs have type $i+1 \to i+1 \to (i+1,0)$, we can treat reflexives as relation reducers:

himself $\rightsquigarrow$
$\lambda \mathbb{P}_{i+1 \to i+1 \to (i+1,0)} \lambda s.((\mathbb{P} s) s)$
$:: (i+1 \to i+1 \to (i+1,0))$
$\quad \to (i+1 \to (i+1,0))$

loves himself $\rightsquigarrow \cdots$
$\lambda s \lambda c \lambda c'.(((\text{love } c_s)\ c_s) \wedge c = c')$

every man loves himself $\rightsquigarrow \cdots$
$\lambda c \lambda c'.(\neg \exists x(\text{man } x \wedge \neg \text{love } x\ x) \wedge c = c')$

29 Toy Fragment: Relative Clauses

that $\rightsquigarrow$
$\lambda P_{\triangleright(i+J,K)} \lambda Q_{\triangleright(i,J)} \lambda j.((Q\ j); (P\ j))$
$:: \triangleright(i+J,K) \to \triangleright(i,J) \to \triangleright(i,J+K)$

loves a woman $\rightsquigarrow$
$\lambda j c c'.\exists x(\text{woman } x \wedge ((\text{love } x)\ c'_j) \wedge c \hat{}\, x = c')$

that loves a woman $\rightsquigarrow$
$\lambda Q j c c'.\exists c''(Q j c c'' \wedge$
$\quad \exists x(\text{woman } x \wedge ((\text{love } x)\ c''_j) \wedge c'' \hat{}\, x = c'))$

man that loves a woman $\rightsquigarrow$
$\lambda j c c'.\text{man } c_j \wedge$
$\quad \exists x(\text{woman } x \wedge ((\text{love } x)\ c_j) \wedge c \hat{}\, x = c')$

30 Toy Fragment: Text Connectives

; $\rightsquigarrow$;
$:: \quad (i,J) \to (i+J,K) \to (i,J+K)$

. $\rightsquigarrow$;
$:: \quad (i,J) \to (i+J,K) \to (i,J+K)$

if $\rightsquigarrow \lambda P_{(i,J)} \lambda Q_{(i+J,K)}.\neg(P; \neg Q)$
$:: \quad (i,J) \to (i+J,K) \to (i,0)$

suppose $\rightsquigarrow$
$\lambda P_{(i,J)} \lambda Q_{(i+J,K)}.\neg(P; \neg Q)$
$:: \quad (i,J) \to (i+J,K) \to (i,0)$

then $\rightsquigarrow$
$\lambda \mathbb{P}_{(i+J,K) \to (i,0)} \lambda Q_{(i+J,K)}.(\mathbb{P}\ Q)$
$:: \quad ((i+J,K) \to (i,0)) \to (i+J,K) \to (i,0)$

An example of a text in the fragment:

2 *Suppose a farmer owns a donkey. Then he beats it.*

31 Conclusions

Current reformulations of DRT within a type-theoretic framework are all without fail based on dynamic logic with destructive assignment. This holds for the dynamic Montague grammar of Groenendijk and Stokhof (1990), for Muskens' logic of change (1994), for Van Eijck's typed logic with states (1997), for Saarbrücken style lambda DRT (Kohlhase et al., 1996), and so on. In short, any framework that in some way takes the Groenendijk and Stokhof DPL (1991a) way of treating dynamic variables as its point of departure will suffer from the same ailment: the problem of destructive assignment will at some level spoil a correct treatment of anaphor-antecedent linking.

Incremental dynamics avoids this problem by taking context updating seriously. Incremental dynamics is both a 'better' rational reconstruction of DRT than DPL and an improvement on DRT itself. It is a better rational reconstruction because it does away with the artificial problems introduced by the DPL treatment of variables. It is an improvement because it makes clear that the DRT departure from the standard type-theoretic paradigm introduced by Montague was unnecessary after all. Indeed, typed incremental dynamics has the same advantages over dynamic Montague grammar and its ilk that ID has over DPL (and to a lesser extent over DRT).

We are now in a position to combine incremental dynamics with dynamic modality in a principled fashion, by integrating epistemic modalities within ID. This task is much easier than the combination of DPL and Update Semantics discussed by Groenendijk and Stokhof in (1991b) because DPL is non-eliminative whereas ID is eliminative by its very nature.

Further information on ID can be found in (van Eijck, 2000), while (van Eijck, 1999) provides a perhaps unexpected spin-off of the ID perspective: elegant axiomatisations of DPL and DRT.

Acknowledgements Thanks to Johan van Benthem, Paul Dekker, Michael Kohlhase, Kees Vermeulen and Albert Visser for their comments on a draft version of this paper.

References

N.G. de Bruijn. 1980. A survey of the project AUTOMATH. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, London.

J. Groenendijk and M. Stokhof. 1990. Dynamic Montague Grammar. In L. Kalman and L. Polos, editors, *Papers from the Second Symposium on Logic and Language*, pages 3–48. Akademiai Kiadoo, Budapest.

J. Groenendijk and M. Stokhof. 1991a. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100.

J. Groenendijk and M. Stokhof. 1991b. Two theories of dynamic semantics. In J. van Eijck, editor, *Logics in AI—European Workshop JELIA '90*, Springer Lecture Notes in Artificial Intelligence, pages 55–64, Berlin. Springer, Berlin.

I. Heim. 1982. *The Semantics of Definite and Indefinite Noun Phrases*. Ph.D. thesis, University of Massachusetts, Amherst.

H. Kamp. 1981. A theory of truth and semantic representation. In J. Groenendijk et al., editors, *Formal Methods in the Study of Language*. Mathematisch Centrum, Amsterdam.

M. Kohlhase, S. Kuschert, and M. Pinkal. 1996. A type-theoretic semantics for λ-DRT. In P. Dekker and M. Stokhof, editors, *Proceedings of the Tenth Amsterdam Colloquium*, Amsterdam. ILLC.

R. Muskens. 1994. A compositional discourse representation theory. In P. Dekker and M. Stokhof, editors, *Proceedings 9th Amsterdam Colloquium*, pages 467–486. ILLC, Amsterdam.

J. van Eijck and H. Kamp. 1997. Representing discourse in context. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 179–237. Elsevier, Amsterdam.

J. van Eijck. 1997. Typed logics with states. *Logic Journal of the IGPL*, 5(5):623–645.

J. van Eijck. 1999. Axiomatising dynamic logics for anaphora. *Journal of Language and Computation*, 1:103–126.

J. van Eijck. 2000. Incremental dynamics. *Journal of Logic, Language and Information*.