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 ago 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 Introduction

The representation of context in mathematical discourse is a subject that has received considerable attention in attempts to formalize mathematics. The topic belongs to a sub discipline of mathematical logic called *type theory*, a much richer field than transpires from the theory of simple types adopted by Richard Montague in his program of formal semantics for natural language of the 1970s (Montague, 1973).

In flexible Montague grammar (Hendriks, 1993) simple type theory was replaced by more flexible typing schemes. In a somewhat different direction, there have been proposals to use Martin-Löf style type theory (Martin-Löf, 1984; Martin-Löf, 1982) as a basis for NL processing systems. A meta-mathematical investigation of Martin-Löf type theory is carried out in (Aczel, 1980).

Unfortunately, the perspectives of model-theoretic semantics and of Martin-Löf type theory are at odds. Martin-Löf type theory is inspired by proof theory. Type theorists in the Martin-Löf tradition replace explanations of meaning in terms of truth by an attempt to explain meaning in terms of proofs, viewed as formal objects in their own right. Martin-Löf types are just sets of proofs. Proofs that depend on assumptions are collected in so-called *dependent types*. To prove a universal statement 'every A satisfies B', assume that you have a proof that some x is an A. If you can then always prove *on that assumption about* x that x is also a B, you are done. The set of all dependent proofs satisfying these requirements is called $(\Pi x \in A)B[x]$; this is a dependent type, and it constitutes the meaning of the universal statement. Similarly, $(\Sigma x \in A)B[x]$ is a dependent type constituting the meaning of an existential statement 'some A satisfies B'.

Sundholm (1986) points out how this approach accounts for the meaning of 'donkey sentences' such as (1).

If a man owns a donkey he beats it. (1)

In a Martin-Löf style account, *a man who owns a donkey* is a proof, more specifically an element of the following set:

$$\{x \in \text{MAN} : (\Sigma y \in \text{DONKEY})\text{OWN}[x, y]\}.$$

A proof in this set can be reduced to a standard form. In this form, it looks like a pair (m, b), with m a man (or rather, a proof that something is a man), and b a member of $(\Sigma y \in \text{DONKEY})\text{OWN}[m, y]$. This b can again be reduced to a standard form. In that form, it is a pair (d, c), with d a (proof that something is a) donkey and c a proof in $\text{OWN}[m, d]$, i.e., a proof that the man owns the donkey. The account is rather vague on what the most elementary proofs should look like, but that is another matter.

Ahn and Kolb (1990) and Ranta (1991) (also (Ranta, 1994)) work out this hint in the form of a rational reconstruction of the dynamic semantics paradigm of Discourse Representation Theory (Kamp, 1981) and File Change Semantics (Heim, 1982), in terms of the theory of dependent types.

Attempts to incorporate dynamic semantics in mainstream Montague style natural language semantics were also made. These attempts preserve much more of the original model-theoretic flavour of Montague grammar. Examples can be found in (Groenendijk and Stokhof, 1990; Muskens, 1994; van Eijck, 1997; van Eijck and Kamp, 1997; Kohlhase et al., 1996; Kuschert, 2000). What these attempts at incorporation have in common is the representation of context as a list of variables, and the use of dynamic quantification over these variables as context update. A serious disadvantage of dynamic quantification is that update of a reference marker u_i makes the previous value of u_i inaccessible: dynamic quantification over named variables brings the problem of destructive assignment in its wake.

In current dynamic versions of Montague grammar, an analysis of *a man entered* would introduce a noun phrase *a man*^{i} with an antecedent index i, and with a translation as in (2).

$$\lambda P \lambda s_0 \lambda s_1 \exists s_2 (s_0[u_i]s_2 \wedge \text{man } u_i \wedge P u_i s_2 s_1). \quad (2)$$

Here u_i is a reference marker. Reference markers in the typed logic can be considered as a special kind of *constants* of type m. s_0, s_1, s_2 are state variables; states are functions from reference markers to entities; they have type $m \to e$, which we abbreviate as s. $s_0[u_i]s_2$ means that the two states differ at most in the value of u_i (this is where destructive assignment takes place). P is a variable of type $m \to s \to s \to t$. It maps a reference marker to a state transition. The magic appearance of the antecedent index i (the index of the reference marker u_i) is a problem with this account.

This paper shows how the destructive assignment problem can be avoided by dispensing with reference markers in favour of argument binding by means of indexing into a stack, thus making it possible to replace magical index introduction by computation of antecedent indices from the input context. The redesign of the dynamic logic paradigm for handling context and context change looks as follows. In incremental dynamics (van Eijck, 2000), a context gets represented as a stack of n objects $c_0, \dots, c_{n-1}$, with repetition of objects allowed. Context extension is achieved by adding a new object c_n. Reference resolution for pronouns is viewed as a process of linking a pronoun to a contextually given object c_i.

While staying within the confines of model theoretic semantics of natural language, we show that it is profitable to adopt a sophisticated type theory to give a principled account of *context composition.* Our account uses a mechanism of *indexing into contexts* that was developed to model context in type theory: the so-called De Bruijn indices from lambda calculus (de Bruijn, 1980). Formalized mathematics hardly uses pronouns, and pronoun resolution is not really an issue there. Still, the context mechanism employed in type theory provides an excellent starting point for setting up context handling for pronoun resolution.

Consider (3), a sentence that needs a context for its interpretation.

Hilary forgave him. (3)

To interpret (3), we have to resolve the reference of *him.* Reference resolution for the pronoun *him* is done with respect to a set of suitable candidate antecedents for the pronoun, and it is reasonable to assume that such candidate sets are finite, and that they have a certain additional structure. We may assume, e.g., that contextually available individuals have sortal information on them, and that the context comes with a salience ordering. To keep matters manageable, however, we will consider contexts of the simplest possible kind. Our contexts are just finite lists of individuals. Once we have a formal account of how such contexts are combined, it is our hope that further enrichment with salience orderings, sorts, and so on, will be relatively straightforward.

Contexts can be set up or extended by sentences. Sentence (4) sets up a context for later use.

Bill made a fool of himself. (4)

Sentence (4) itself can be interpreted irrespective of an anaphoric context. It sets up a context with an individual Bill in it that is available for further reference. The reference resolution involved in the processing of (3) after (4) can make use of the context created by the processing of the first sentence.

2 The Key Issue

Let us look at context processing a bit more systematically. 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).

- $\boxed{\texttt{txt txt txt} \uparrow}$: 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).

Our question in this paper is: how can we set up context dynamics in the proper Montagovian style, without running into the destructive assignment problem? We view text interpretation as a process that adds a number of referents (say m) to an existing 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 (5) mean?* has as its answer: they are just glosses to indicate how we suppose the anaphoric reference resolution mechanism links pronouns to antecedents. In example (5), the indices link the pronoun from the second sentence to the noun phrase introduced in the first.

$$\text{A man}^i \text{ walked in. He}_i \text{ smiled.} \tag{5}$$

If we represent an anaphoric context as a stack of n objects available as antecedents in future discourse, then introducing a new topic of conversation extends the anaphoric context by putting a new object on top of the context stack. Thus, the value of i gets determined by the input context.

In our set-up, 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.

3 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 value) to T (state transition) constitutes the dynamic turn in natural language semantics. The state transitions can themselves be viewed as relations between states, and the states in turn as functions from reference markers to entities. So in a more fine-grained set-up, with basic types for t for truth values, e for entities, and m for reference markers, state transitions are of type $(m \to e) \to (m \to e) \to t$.

In this set-up, the interpretation of *a man walked in* is an object of type $s \to s \to t$, i.e., a relation between states. The interpretation will relate states where some reference marker u_i has a certain value (or does not have a value) to states where u_i gets a (new) value d with d satisfying the predicates *man* and *walked-in*. A problem with this is that the account does not specify how to select u_i.

Incremental dynamics avoids this problem. 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 $(5, 3)$.

In a context of length 3, the sentence *a man walked in* will get interpreted as an expression of type $(3, 1)$: the interpretation will act on the context by extending it with one element, namely with a referent for *a man*. A next sentence *he smiled* can then pick up this reference. In the context of length 4 produced by the previous text, the sentence *he smiled* will have an interpretation of type $(4, 0)$: the context of length 4 is not extended.

Here is the definition of the type system that we need for this:

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

N are the natural numbers, $[N]$ are the contexts of sizes given by the natural numbers N. Abbreviation: use (N_1, N_2) for

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

4 Index Variables and Type Schemes

So far, so good. But how do we specify the type of a sentence *a man walked in* if we do not know the size of the initial context of interpretation? To represent contexts of arbitrary size, we introduce *index variables.*

Using an index variable i, we can say that the interpretation of *a man walked in* extends an arbitrary context of size i by one position, so it is of type $(i, 1)$. We call the variable i the context index. We call types that depend on an index type schemes. $(i, 3)$ is the type scheme of a stack transition that extends a stack of length i by 3 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. Here are some further 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. E.g., $\triangleright(i+1, 2)$ is a function that takes an expression of type $i+1$ and produces an extension of a non-empty context by 2 positions, i.e., an expression of type $(i+1, 2)$.

5 Pattern Variables and Type Scheme Patterns

We will also want to express that an arbitrary context is extended by an indeterminate number of elements, in order to use this for encoding the context transformations involved in the meanings of common nouns and verb phrases. A common noun may contain an arbitrary number of indefinite noun phrases and proper names inside relative clauses, so we don't know in advance by how much it will extend the context stack: *man* does not extend the context stack, *man who owns a donkey* extends the context stack by one position, *man who has a neighbour that owns a donkey* extends the context stack by two positions, and so on. The same thing holds for verb phrases: *walked in* does not extend the context stack, *walked in with a dog* extends the context stack by one position, and so on.

To represent extension of context by an indeterminate number of elements, we introduce numerical variables. Thus, $(i+1, J)$ is the type scheme pattern of a stack transition that extends a non-empty context by J positions. $(i, J+1)$ is the type scheme pattern of a stack transition that extends an arbitrary context by $J+1$ positions (i.e., at least one). A syntactic pattern

[[a CN] VP]

will be of type $(i, J+1)$, for we know that the indefinite constituting the subject extends the stack by one position, but we do not know yet what the context contributions of CN and VP will turn out to be. So the context is extended with at least one position, hence the pattern $J+1$.

We distinguish between index variables and pattern variables, and use i, j, k for index variables, I, J, K for pattern variables. This distinction is important, for fleshing out a syntactic pattern like *a CN VP* will instantiate the pattern variables while the context index remains unaffected.

We call a type containing pattern variables a *type pattern*, and a type scheme containing pattern variables a *type scheme pattern*. An example of the latter:

> $i+1 \to (i+1, J)$ 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). Examples:

- $K := 3$ 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 $i+1 \to (i+1, 2)$.

The following sums up the distinction between pattern variables and 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.

Here is the formal definition of Type Scheme Patterns:

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

Next, we specify what it means to *instantiate* a type scheme pattern.

- 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.

Our definition of functional application will have to involve type scheme pattern matching. If we want to apply a function f with type scheme $i \to (i, J)$ to an argument a with type $k + 1$, then we get an appropriate match by unifying i and $k + 1$, with result that $f(a)$ is of type $(k + 1, J)$.

The most general way of matching function and argument type scheme patterns is by means of a unification algorithm phrased in terms of most general unifiers. 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 $T\theta$.

A word of caution concerning unification of numerical terms is in order. 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. It follows 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*. Note that in-between cases exist: the pair $(N + M) \approx (K + L)$ is an example.

6 Unification of Terms and of Type Scheme Patterns

As a preliminary for the unification algorithm for numerical terms, we write numerical terms in canonical form. If Nexp_1 and Nexp_2 are numerical terms, then the pair $\mathrm{Nexp}_1 \approx \mathrm{Nexp}_2$ can be written in a canonical form as follows:

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

With this recipe, a pair $\mathrm{Nexp}_1 \approx \mathrm{Nexp}_2$ 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 left hand side and right hand side 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 $(n > 0, m > 0, k > 0)$

Now define **Failing Pairs**, as follows:

- 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.

Next, define **Unifying pairs**, with their substitutions, using ϵ for the empty substitution (the substitution that maps every term to itself).

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

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

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

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

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

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

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

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

Finally, here is a sketch of the unification algorithm for type scheme patterns, in terms of unification of numerical expressions.

- 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 for term unification above.
- $[\mathrm{Nexp}_1]$ unifies with $[\mathrm{Nexp}_2]$ if Nexp_1 unifies with Nexp_2 and gives mgu according to the rules for term unification above.
- $\mathrm{Tsp}_1 \to \mathrm{Tsp}_2$ unifies with $\mathrm{Tsp}_3 \to \mathrm{Tsp}_4$ if Tsp_1 unifies with Tsp_3 to give mgu θ, and $\mathrm{Tsp}_2\theta$ unifies with $\mathrm{Tsp}_4\theta$ to give mgu σ, and gives mgu $\theta\sigma$.
- No other pairs of Tsp's unify.

We list some facts about type scheme pattern 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.)

To end this section, we give examples of expressions and their type scheme patterns.

Dynamic Exists Using $\exists$ as abbreviation for the dynamic context extension quantifier mentioned above, and writing :: for the copula in a typing judgment, we can express the type scheme pattern of the dynamic quantifier as follows:

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

Here is the explanation. $\exists$ 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. For example, if P is a stack transformer of type $(1, 0)$, then $\exists P$ will be a stack transformer of type $(0, 1)$, for its truth content will be the assertion that P is non-empty, and $\exists P$ will create a context containing an object satisfying P. More generally, if P is a stack transformer of type $(i+1, 0)$, i.e., a test on arbitrary non-empty contexts, then $\exists P$ will be the stack transformer that extends an arbitrary context with one element. The truth content of $\exists P$ will now be the assertion that there is a way to extend an arbitrary context c with a new individual to give a new context c' that satisfies P. The output of the transition $\exists P$ will be c'. Since c' equals c plus one extra element, $\exists P$ extends a context of size i with one element, i.e., it is of type $(i, 1)$. Still more general, finally, is the case where P is not a test but a transition that extends context, with a number of elements J. 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, namely $(i+1, 0) \to (i, 1)$, $(i+1, 1) \to (i, 2)$, and so on. In all these type schemes for $\exists$ there occurs just one type variable, namely i.

Dynamic Negation Using $\neg$ as an abbreviation for dynamic negation we get the following typing judgment.

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

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

Context Composition Using **;** as abbreviation for the context composition operation, we get:

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

; 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.

In the next section we will give the definitions of $\exists$, $\neg$ and **;** that satisfy these judgments.

7 The Logical Language TID

In this section, we introduce the language of Typed Incremental Dynamics (TID) by means of examples.

First the language has constants of arbitrary types. Here, e.g., is the type judgment for a constant for the property of being a man:

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

The translation of a common noun that does not extend the stack is an index into tests on non-empty stacks, i.e., an index into $[i+1] \to [i+1] \to t$. The translation of the common noun '*man*' uses the constant 'man' to construct this expression. A variable j is an appropriate index into a stack $[i+1]$ if its type is $i+1$, i.e., if j ranges over the set of natural numbers $\{0, \ldots, i\}$. Here is the translation for the CN *man* that we are looking for:

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

This can be viewed as a recipe for selecting a man from a non-empty context. The application $(c\ j)$, for $c\ ::\ [n]$, and $j\ ::n$, is written as c_j. As agreed, $i+1 \rightarrow (i+1,0)$ is abbreviated as $\triangleright(i+1,0)$.

It should be noted that subscripts do double duty: in the annotation of abstracted variables $\lambda c_{[i+1]}$, the subscript $[i+1]$ means that c is of type $[i+1]$. In the application c_j, the subscript j means that c is applied to j. This should not create confusion: context of use always makes clear what is meant.

For the definition of $\mathbf{\exists}$, we extend the usual logic for extensional type theory with a constant $[]\ ::\ [0]$ and an operation $(\hat{})\ ::\ [i] \rightarrow e \rightarrow [i+1]$. 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 $\mathbf{\exists}$, as follows. $\mathbf{\exists}$ 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.

$\neg\!\!\neg$ is an abbreviation of:

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

$\mathbf{;}$ 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 $\mathbf{;}$ as an infix operator with association to the left, and we will omit superfluous brackets.

Note how index variables 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 the final position of the stack. Thus, a sentence with an indefinite subject has the general form [[a CN] VP]. Its semantics is a context transformation that takes an arbitrary 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*. The recipe for translating the indefinite determiner becomes:

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

We need patterns $(i+1,N)$, ... 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 $\mathbf{;}$ to $(i+1,N) \rightarrow (i+N+1,M) \rightarrow (i+1,N+M)$. Thus we get that $(Pi\ \mathbf{;}\ Qi)\ ::\ (i+1,N+M)$.

Since $\mathbf{\exists}\ ::\ (i+1,J) \rightarrow (i,J+1)$ 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 $\mathbf{\exists}$ to $(i+1,N+M) \rightarrow (i,N+M+1)$, and we get that $\mathbf{\exists}\ (Pi\ \mathbf{;}\ Qi)\ ::\ (i,N+M+1)$.

This illustrates a general procedure of function application with unification. Function application may involve unification of type schemes, as follows:

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

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

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

applied to

$$\lambda j \lambda c \lambda c' M c_j \wedge c = c' :: T_3$$

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

$$\begin{aligned}&(\lambda P_{\triangleright(k+1,0)} \lambda c \lambda c' \exists x. \\ &\ Pkc\hat{}x\ c)(\lambda j \lambda c \lambda c' M c_j \wedge c = c') \\ &\ :: (k,1).\end{aligned}$$

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

8 Building a Toy Fragment

In this section the Montagovian programme of building a toy fragment is taken up. To begin with, here is how to deal with determiners, nouns and intransitive verbs.

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

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

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

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

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

If A is a list of i elements $A_0, \ldots, A_{i-1}$, and we append a new element B to the list, then in the resulting list $A_0, \ldots, A_{i-1}, B$, the element B occupies position i, so we can retrieve B from the list by lookup at index i. 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:

$$\lambda x.((c_{[4]} \hat{} x)\ 4) \Rightarrow_\iota \lambda x.x$$

Or with variables:

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

Reduction to normal form is achieved by combining beta reduction with iota reduction. Beta reduction is defined in the standard way. We state here without proof that beta-iota reduction is confluent, i.e, if $E \overset{\beta\iota}{\twoheadrightarrow} F$ and $B \overset{\beta\iota}{\twoheadrightarrow} F'$ then there is a G with $F \overset{\beta\iota}{\twoheadrightarrow} G$ and $F' \overset{\beta\iota}{\twoheadrightarrow} G$. Also, beta-iota reduction is strongly normalizing, i.e., every reduction sequence $E \xrightarrow{\beta\iota} F \xrightarrow{\beta\iota} G \ldots$ terminates. It follows that beta-iota reduction yields unique normal forms. As an example, look at the reduction of the translation for 'a man smiled':

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

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

Construction of an anaphora resolution engine is outside our scope, but the present framework makes it easy to specify exactly where anaphora resolution fits in and what it works on. 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 amount of further information to pick an index for that context. The slot $\_$ indicates the place where extra input is needed to perform the resolution step.

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

Here is how to extend the toy fragment with pronouns and transitive verbs.

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

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

$\text{him} \rightsquigarrow \ldots$

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

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

Suppose we are in a context of size ≥ 2. Then we can handle the example sentence 'he$_1$ loves her$_2$', with the pronouns resolved in context as indicated by the indices. The reduction proceeds as follows:

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

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

In as similar way we get a translation of *loves her*, with the pronoun unresolved:

$$\lambda s \lambda c \lambda c'.(((\text{love } c_{(\text{res } c \_)}) \ c_s) \wedge c = c'.$$

In a system with flexible typing, the type $(i+1 \rightarrow (i+1,J)) \rightarrow (i+1,J)$ for unresolved pronouns can be lowered to $[i+1] \rightarrow i+1$, for the meaning of an unresolved pronoun can now simply be given as an invitation to pick a suitable index from a non-empty context. The meaning of an anaphorically resolved pronoun is even simpler to specify; it can be given as an index into the appropriate context. Using the type scheme variables to transfer information about the size of the context, we get by with the following:

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

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

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

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

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

$$\begin{array}{l} \text{himself} \rightsquigarrow \\ \lambda \mathbb{P}_{i+1 \rightarrow i+1 \rightarrow (i+1,0)} \lambda s.((\mathbb{P}s)s) \\ :: (i+1 \rightarrow i+1 \rightarrow (i+1,0)) \\ \quad \rightarrow (i+1 \rightarrow (i+1,0)) \end{array}$$

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

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

Relative clauses are now dealt with as follows:

$$\begin{array}{l} \text{that} \rightsquigarrow \\ \lambda P_{\triangleright(i+J,K)} \lambda Q_{\triangleright(i,J)} \lambda j.((Q \ j); (P \ j)) \\ :: \triangleright(i+J,K) \rightarrow \triangleright(i,J) \rightarrow \triangleright(i, J+K) \end{array}$$

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

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

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

Finally, here are instructions for text connectives.

$$\begin{array}{l} ; \rightsquigarrow \ \mathbf{;} \\ :: \quad (i, J) \rightarrow (i+J, K) \rightarrow (i, J+K) \end{array}$$

$$\begin{array}{l} . \rightsquigarrow \ \mathbf{;} \\ :: \quad (i, J) \rightarrow (i+J, K) \rightarrow (i, J+K) \end{array}$$

$$\begin{array}{l} \text{if} \rightsquigarrow \lambda P_{(i,J)} \lambda Q_{(i+J,K)}.\neg(P; \neg Q) \\ :: \quad (i, J) \rightarrow (i+J, K) \rightarrow (i, 0) \end{array}$$

$$\begin{array}{l} \text{suppose} \rightsquigarrow \\ \lambda P_{(i,J)} \lambda Q_{(i+J,K)}.\neg(P; \neg Q) \\ :: \quad (i, J) \rightarrow (i+J, K) \rightarrow (i, 0) \end{array}$$

$$\begin{array}{l} \text{then} \rightsquigarrow \\ \lambda \mathbb{P}_{(i+J,K) \rightarrow (i,0)} \lambda Q_{(i+J,K)}.(\mathbb{P} \ Q) \\ :: \quad ((i+J, K) \rightarrow (i, 0)) \rightarrow (i+J, K) \rightarrow (i, 0) \end{array}$$

(6) is an example of a text in the fragment:

Suppose a man owns a donkey. Then he beats it. (6)

The key element of the treatment of *suppose* is that we keep track of the size of the context extension in its scope, by means of the pattern variable J. In (6), the first sentence is interpreted as a function of type $(i+2, K) \rightarrow (i, 0)$, where i is the index for the original context. This original context is extended by 2 elements, to wit a man and a donkey owned by that man. The second sentence is of type $(k+2, 0)$, for it is a test on a context containing at least two elements. Function application with unification gives $(i, 0)$ for the type of the translation of (6); this is the type of a test on the original context.

9 Related Work

For connections to Martin-Löf style type theory, and to earlier attempts at a Montagovian treatment of dynamic semantics, see Section 1 above. A somewhat different proposal for a variable free logic for natural language is made in (Purdy, 1991). This work starts out from relation algebra, and is not presented in a Montagovian framework. The issue of incremental updating of contexts is also taken up in a proposal for a variation on predicate logic, in (Dekker, 1994). Van Eijck (2000) gives a sound and complete calculus for ID, essentially a variable free version of dynamic predicate logic, while (van Eijck, 1999) provides a perhaps unexpected spin-off of the ID perspective: elegant axiomatizations of DPL and DRT.

An early warning against the proliferation of indices in the syntax of Montague grammar can be found in (Landman and Moerdijk, 1983). This theme is taken up again in (Jacobson, 1992; Jacobson, 1999), where an attempt is made to account for the meaning of expressions containing pronouns without using unbound variables. To achieve this, type shift operations are proposed that implicitly bind the variables corresponding to the pronouns. We propose a different solution, one that does not involve type shifting for expressions with 'unbound' pronouns. In the present proposal, the translation of *loves her* has the same type as that of *loves Mary*. The difference is that the translation of *loves her* depends on the input context in a way that the translation of *loves Mary* does not. Thus, the proper treatment of pronouns is a treatment that makes their context dependence explicit. An additional advantage of the present approach is that it generalizes to dynamic phenomena in a straightforward way, something which cannot be said about Jacobson's.

For a thorough study of context dependence in dynamic semantics we refer to (Visser, 1994). In the spirit of this work, Vermeulen (1994) introduces *referent systems* to separate the 'storage facility' of a variable from the 'control facility' provided by the variable name. Referent systems are employed in (Groenendijk et al., 1996) to define a system for combining DPL style dynamics with dynamic modality (Veltman, 1996). Incremental dynamics can be viewed as a proposal to dispense with variable names (referents) altogether: the variable name in dynamic semantics is a red herring, and contexts as employed in ID are what is left of referent systems once these names are thrown out. Since ID is eliminative by its very nature, the task of integrating dynamic modality in ID is straightforward. Carrying this out would show that referent systems are only needed in the rudimentary form provided for in ID.

The possibility to connect up with theories of anaphora resolution was already mentioned above. A centering algorithm along the lines of (Grosz et al., 1995; Kameyama, 1998) can be plugged in to specify what happens at the open slot in

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

Hints on how this could be done can be found in (Beaver, 1999; Beaver, 2000).

10 Conclusions

Current reformulations of DRT within a type-theoretic framework are either given in a Martin-Löf style type theory where the problem of finding appropriate representations for pronominal elements is not addressed, or they are based on dynamic logic with destructive assignment. The latter holds for the dynamic Montague grammar of Groenendijk and Stokhof (1990), for Muskens' logic of change (1994), (1996), for Van Eijck's typed logic with states (1997), for Saarbrücken style lambda DRT (Kohlhase et al., 1996; Kuschert, 2000), and so on. In short, any framework that in some way takes the DPL (Groenendijk and Stokhof, 1991) way of treating dynamic variables as its point of departure suffers from the same ailment: the problem of destructive assignment will at some level spoil a correct treatment of anaphor-antecedent linking.

The introduction of referent systems (Vermeulen, 1994; Vermeulen, 1995) solves the destructive assignment problem by holding on to the variable names and fitting out the dynamic variables with additional superstructure. Incremental dynamics shows how to get rid of destructive assignment by getting rid of dynamic variable names altogether while preserving the essence of the context update mechanism.

Incremental dynamics takes context updating seriously; it 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.

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, and to an anonymous referee for urging me to clarify the connection with Martin Löf style type theory.

References

P. Aczel. 1980. Frege structures and the notions of proposition, truth and set. In J. Barwise, H.J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 31–59. North-Holland, Amsterdam.

R. Ahn and H.-P. Kolb. 1990. Discourse representation meets constructive mathematics. In L. Kalman and L. Polos, editors, *Papers from the Second Symposium on Logic and Language*, pages 105–124. Akademiai Kiadoo, Budapest.

D. Beaver. 1999. The logic of anaphora resolution. In P. Dekker, editor, *Proceedings of the Twelfth Amsterdam Colloquium*, pages 61–66, Amsterdam. ILLC.

D. Beaver. 2000. Centering in OT. Handout, Utrecht, September.

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.

P. Dekker. 1994. Predicate logic with anaphora. In L. Santelmann and M. Harvey, editors, *Proceedings of the Fourth Semantics and Linguistic The-*

ory Conference, page 17 vv, Cornell University. DMML Publications.

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. 1991. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100.

J. Groenendijk, M. Stokhof, and F. Veltman. 1996. Coreference and modality. In S. Lappin, editor, *The Handbook of Contemporary Semantic Theory*, pages 179–213. Blackwell, Oxford.

B. Grosz, A. Joshi, and S. Weinstein. 1995. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21:203–226.

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

H. Hendriks. 1993. *Studied Flexibility; Categories and Types in Syntax and Semantics*. Ph.D. thesis, ILLC, Amsterdam.

P. Jacobson. 1992. Antecedent contained deletion in a variable-free semantics. In C. Barker and D. Dowty, editors, *Proceedings of the Second Conference on Semantics and Linguistic Theory*, pages 193–213. Ohio State University.

P. Jacobson. 1999. Towards a variable-free semantics. *Linguistics and Philosophy*, 22(2):117–184, April.

M. Kameyama. 1998. Intrasentential centering: a case study. In M. Walker, A. Joshi, and E. Prince, editors, *Centering Theory in Discourse*, pages 89–112. Clarendon Press.

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.

S. Kuschert. 2000. *Dynamic Meaning and Accommodation*. Ph.D. thesis, Universität des Saarlandes. Thesis defended in 1999.

F. Landman and I. Moerdijk. 1983. Compositionality and the analysis of anaphora. *Linguistics and Philosophy*, 6:89–114.

P. Martin-Löf. 1982. Constructive mathematics and computer programming. In Cohen, Los, Pfeiffer, and Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, pages 153–179. North Holland.

P. Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis.

R. Montague. 1973. The proper treatment of quantification in ordinary English. In J. Hintikka e.a., editor, *Approaches to Natural Language*, pages 221–242. Reidel.

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

R. Muskens. 1996. Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, 19:143–186.

W.C. Purdy. 1991. A logic for natural language. *Notre Dame Journal of Formal Logic*, 32:409–425.

Aarne Ranta. 1991. Intuitionistic categorial grammar. *Linguistics and Philosophy*, 14:203–239.

A. Ranta. 1994. *Type-theoretical Grammar*. Oxford University Press.

G. Sundholm. 1986. Proof theory and meaning. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic III*, pages 471–506. Kluwer, Dordrecht.

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*.

F. Veltman. 1996. Defaults in update semantics. *Journal of Philosophical Logic*, pages 221–261.

C.F.M. Vermeulen. 1994. *Explorations of the Dynamic Environment*. Ph.D. thesis, OTS, Utrecht.

C.F.M. Vermeulen. 1995. Merging without mystery. *Journal of Philosophical Logic*, 24:405–450.

A. Visser. 1994. The design of dynamic discourse denotations. Lecture notes, Utrecht University.