Quantification and Reference in Incremental Processing

Jan van Eijck[1] and Rick Nouwen[2]
CWI and ILLC, Amsterdam[1], *Uil-OTS, Utrecht*[1,2]

Second Draft — Spring 2002

Abstract

This paper shows how polymorphic type theory can be used for a compositional incremental natural language semantics that can handle singular and plural reference and quantification, in a setting where context gets dynamically extended. We focus on the treatment of quantifiers and anaphoric reference in dynamic semantics, and for ease of exposition we limit ourselves to distributive readings of plurals. The approach is illustrated with a fragment implemented in the functional programming language Haskell. The paper contains the complete Haskell code of our implementation, in 'literate programming style'.

Keywords: Dynamic predicate logic, discourse semantics, discourse representation, anaphora, natural language meaning, salience, pronoun resolution, quantification, reference, distributive plurals.

MSC codes: 03B60, 03B65, 68S05, 68Q65.

1 Point of Departure: Incremental Dynamics

Our starting point is an incremental logic for NL semantics called Incremental Dynamics (henceforth ID). ID (see [10, 8] for background) can be viewed as the one-variable version of sequence semantics for dynamic predicate logic proposed in [33]. We will propose a format for compositional incremental NL semantics based on polymorphic type theory. The format can handle dynamic context extension, singular and plural reference and quantification. For ease of exposition we focus on the distributive readings of plurals.

Assume a first order model $M = (D, I)$. We will use contexts $c \in D^*$, and replace variables by indices into contexts. The set of terms of the language is $\mathbb{N}$. We use $|c|$ for the length of context c. Given a model $M = (D, I)$ and a context $c = c[0] \cdots c[n-1]$, where $n = |c|$ (the length of the context), we interpret terms of the language by means of $[\![i]\!]_c = c[i]$. A snag is that $[\![i]\!]_c$ is undefined for $i \geq |c|$; we will therefore have to ensure that indices are only evaluated in appropriate contexts. $\uparrow$ will be used for 'undefined'. This allows us to define the relations

$$M \models_c Pi_1 \cdots i_n, \quad M \dashv_c Pi_1 \cdots i_n$$

by means of:

$$M \models_c Pi_1 \cdots i_n \;:\Leftrightarrow\; \forall j(1 \leq j \leq n \Rightarrow [\![i_j]\!]_c \neq\, \uparrow) \text{ and } \langle [\![i_1]\!]_c, \ldots, [\![i_n]\!]_c \rangle \in I(P),$$

$$M \dashv_c Pi_1 \cdots i_n \;:\Leftrightarrow\; \forall j(1 \leq j \leq n \Rightarrow [\![i_j]\!]_c \neq\, \uparrow) \text{ and } \langle [\![i_1]\!]_c, \ldots, [\![i_n]\!]_c \rangle \notin I(P),$$

and the relations

$$M \models_c i_1 \doteq i_2, \quad M \dashv_c i_1 \doteq i_2$$

by means of:

$$M \models_c i_1 \doteq i_2 \;:\Leftrightarrow\; [\![i_1]\!]_c \neq\, \uparrow \text{ and } [\![i_2]\!]_c \neq\, \uparrow \text{ and } [\![i_1]\!]_c = [\![i_2]\!]_c.$$

$$M \dashv_c i_1 \doteq i_2 \;:\Leftrightarrow\; [\![i_1]\!]_c \neq\, \uparrow \text{ and } [\![i_2]\!]_c \neq\, \uparrow \text{ and } [\![i_1]\!]_c \neq [\![i_2]\!]_c.$$

If $c \in D^n$ and $d \in D$ we use $c\hat{\ }d$ for the context $c' \in D^{n+1}$ that is the result of appending d at the end of c.

The ID interpretation of formulas can now be given as a map in $D^* \hookrightarrow \mathcal{P}(D^*)$ (a partial function, because of the possibility of undefinedness):

$$\begin{array}{rcl}
[\![\exists]\!](c) & := & \{c\hat{\ }d \mid d \in D\} \\
[\![Pi_1 \cdots i_n]\!](c) & := & \begin{cases} \uparrow & \text{if } \exists j(1 \leq j \leq n \text{ and } [\![i_j]\!]_c = \uparrow) \\ \{c\} & \text{if } M \models_c Pi_1 \cdots i_n \\ \emptyset & \text{if } M \dashv_c Pi_1 \cdots i_n \end{cases} \\
[\![i_1 \doteq i_2]\!](c) & := & \begin{cases} \uparrow & \text{if } [\![i_1]\!]_c = \uparrow \text{ or } [\![i_1]\!]_c = \uparrow \\ \{c\} & \text{if } M \models_c i_1 \doteq i_2 \\ \emptyset & \text{if } M \dashv_c i_1 \doteq i_2 \end{cases} \\
[\![\neg\varphi]\!](c) & := & \begin{cases} \uparrow & \text{if } [\![\varphi]\!](c) = \uparrow \\ \{c\} & \text{if } [\![\varphi]\!](c) = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\varphi;\psi]\!](c) & := & \begin{cases} \uparrow & \text{if } [\![\varphi]\!](c) = \uparrow \\ & \text{or } \exists c' \in [\![\varphi]\!](c) \text{ with } [\![\psi]\!](c') = \uparrow \\ \bigcup\{[\![\psi]\!](c') \mid c' \in [\![\varphi]\!](c)\} & \text{otherwise.} \end{cases}
\end{array}$$

The definition of the semantic clause for $\varphi;\psi$ employs the fact that all contexts in $[\![\varphi]\!](c)$ have the same length. This property follows by an easy induction on formula structure from the definition of the relational semantics. Thus, if one element $c' \in [\![\varphi]\!](c)$ is such that $[\![\psi]\!](c') = \uparrow$, then all $c' \in [\![\varphi]\!](c)$ have this property.

Dynamic implication $\varphi \Rightarrow \psi$ is defined in terms of $\neg$ and ; by means of $\neg(\varphi;\neg\psi)$. Universal quantification $\forall\varphi$ is defined in terms of $\exists, \neg$ and ; as $\neg(\exists;\neg\varphi)$, or alternatively as $\exists \Rightarrow \varphi$.

One advantage of the use of contexts is that indefinite NPs do not have to carry indices, as in Montague grammar or in dynamic versions of Montague grammar based on Dynamic Predicate Logic.

1 *Some man loved some woman.*

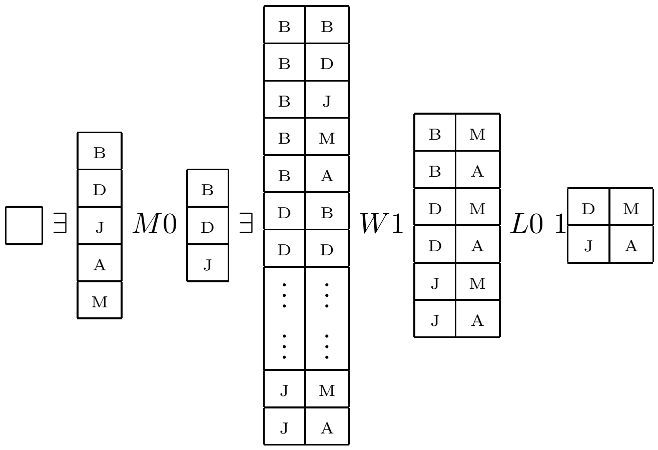
Suppose an utterance like (1) is interpreted in a certain context c, and let i be the length of that input context. Then the ID rendering of (1) is:

$$\exists; Mi; \exists; W(i+1); Li(i+1).$$

For the special case where the input context is empty, we get that (1) is interpreted as the set of all contexts $[e_0, e_1]$ that satisfy the relation 'love' in the model under consideration. The result of this is that the subsequent sentence (2) can now use this contextual discourse information to pick up the references:

2 *He*$_0$ *kissed her*$_1$.

Take for instance a model containing only three men (B, D and J) and two women (M, A). Say, that B loves no-one, D loves M and J loves A. If we now represent contexts as rows of individuals, we can visualize how the semantics updates the set of possible contexts.



In [10] a procedure is specified for reference resolution of pronouns in a given context. For that we need to consider contexts under permutation, where the permutations encode salience. In this paper we will keep matters simple by assuming that pronoun reference gets handled by index information.

2 Encoding in Logic with Polymorphic Types

The Proper Treatment of Context for NL developed in [7] in terms of polymorphic type theory (see, e.g., [18, 26]) uses type specifications of contexts that carry information about the length of the context. E.g., the type of a context is given as $[e]_i$, where i is a type variable. Here, we will cavalierly use $[e]$ for the type of any context, and ι for the type of any index, thus relying on meta-context to make clear what the current constraints on context and indexing into context are. In types such as $\iota \to [e]$, we will tacitly assume that the index fits the size of the context. Thus, $\iota \to [e]$ is really a type scheme rather than a type, although the type polymorphism remains hidden from view. Since $\iota \to [e]$ generalizes over the size of the context, it is shorthand for the types $0 \to [e]_0$, $1 \to [e]_1$, $2 \to [e]_2$, and so on.

The translation of an indefinite noun phrase *a man* becomes something like:

3 $\lambda P \lambda c \lambda c'.\exists x(\mathit{man}\ x \wedge P|c|(c\hat{}x)c')$.

Here P is a variable of type $\iota \to [e] \to [e] \to t$, while c, c' are variables of type $[e]$ (variables ranging over contexts). The translation (3) has type $(\iota \to [e] \to [e]) \to [e] \to [e] \to t$. The P variable marks the slot for the VP interpretation. $|c|$ gives the length of the input context, i.e., the position of the next available slot. Note that $c\hat{}x[|c|] = x$.

Translation (3) does not introduce an anaphoric index, as in DPL based dynamic semantics for NL (see [11]). Instead, an anaphoric index i is picked up from the input context. Also, the context is not reset but incremented: context update is not destructive like in DPL.

3 Extension with Distributive Plurals

We will treat the singular/plural distinction by extending context semantics in the spirit of [4]. States are sets of contexts of the same length. Sentences will be interpreted as state transitions, i.e., as functions from states to states.

For an incremental toy fragment we need to define the appropriate dynamic operations in typed logic. We let a state be (a characteristic function of) a set of contexts of the same length, i.e., the type for states is $[e] \to t$. This is a bit sloppy, as the actual context length information is omitted, but the development below will demonstrate that this causes no harm.

Assume φ and ψ have the type of state transitions, i.e., type $([e] \to t) \to ([e] \to t)$, and that s has type $[e] \to t$ and c, c' have type $[e]$.

Our new dynamic operations now become:

$$\begin{array}{rcl}
\boldsymbol{\exists} & := & \lambda sc.\exists x \exists c' \in s(c = c'\hat{}x) \\
\neg\!\!\neg\varphi & := & \lambda sc.(c \in s \wedge \varphi s = f_\emptyset) \\
\varphi \,\boldsymbol{;}\, \psi & := & \lambda s.\psi(\varphi s) \\
\varphi \Rightarrow \psi & := & \neg\!\!\neg(\varphi \,\boldsymbol{;}\, \neg\!\!\neg\psi) \\
\Downarrow s & := & s \neq f_\emptyset
\end{array}$$

These operations encode the semantics for incremental (distributive) quantification, dynamic incremental negation, dynamic incremental conjunction, dynamic incremental implication, and success, in typed logic. $f_\emptyset$ denotes the empty state. Note that $\Downarrow$ is of type $([e] \to t) \to t$.

For purposes of expressing quantification, it is convenient to have a function for mapping states to the sizes of their contexts. Assume we have a type $\mathbb{N}$ for natural numbers. Then the following is a function of type $([e] \to t) \to \mathbb{N}$.

$$\#(s) \quad := \quad \begin{cases} 0 & \text{if } s = f_\emptyset \\ |c| & \text{if } c \in s \end{cases}$$

Note that the correctness of this definition depends on the fact that all contexts in a state have the same size.

The interpretation process of a plural noun phrase like *the men* will involve extending every context c in the current state with a man. A sentence like *the men VP* is interpreted by checking whether each context $c\hat{}m$ survives the transition associated with *VP*.

In general, we will be interested in the sets we get by collecting the individuals at a given index in a state. In the example of *the men*, the men in the model under consideration are the elements m that occur at some position i in the contexts belonging to a state s.

Extend the indexing notation to states as follows: $s[i] := \{c[i] \mid c \in s\}$. Then, in the example, the collection of men will be $s[i]$.

For the definition of generalized quantifier operations for plurals, we employ a function gq that compares the results of two state transitions for a given index and a given initial state.

$$\text{gq } f \ g \ i \ s \quad := \quad (m, k) \text{ where } m = |f(s)[i]| \text{ and } k = |g(f(s))[i]|$$

This gives a pair of numbers (m, k), where m is the size at i of the result of updating s with f, and k is the size at i of the result of updating s with both f and g. The definition of gq uses the so-called *conservativity* of generalized quantifiers, the property that if Q is a generalized quantifier then $Q(A, B)$ holds iff $Q(A, A \cap B)$ holds. In the present setting, f is the dynamic version of the interpretation of A, and g is the dynamic version of the interpretation of B.

In the case of the interpretation of *all men*, the relevant gq condition is that the numbers in the gq pair (m, k) are equal. For instance, for the interpretation of *all men are mortal* the following test is performed:

$$\text{gq } (\exists; man\#(s)) \ (mortal\#(s)) \ \#(s) \ s \ = \ (m, m) \text{ for some } m$$

Evaluation in the empty state is as follows. $(\exists; man0)f_\emptyset$ denotes the state where each context contains exactly one man. The left value of the generalized quantifier function is thus the cardinality of $s[0]$, the cardinality of the set of all men in the domain of discourse. For the other value, we need to update the state $(\exists; man0)f_\emptyset$ with $mortal0$. The example comes out true if this update does not change the cardinality of $s[0]$, i.e. if there are no immortal men.

4 Modular Implementation

We will give a Haskell [20] implementation of a fragment. In fact, this paper can be viewed as the documentation of that implementation: the implementation code is what appears in typescript in rectangular boxes. The paper contains the full code of the implementation, in Literate Programming style [23].

We have relegated the module for the syntactic datastructures to the first appendix (page 24). The definition of the syntactic datastructures is in a module called `Cat`. The syntax may look rather involved, but in the second appendix (page 30) we specify a simple parser that produces syntactic data-structures in this format. The parser is in a module called `Parser`. The declaration of the main module of the program starts in Section 5.

5 Specification of Basic Types for Context Modeling

Declare a module `Qar` that imports the standard `List` module plus the modules `Cat` and `Parser`.

```
module Qar where

import List
import Cat
import Parser
```

The semantic specifications below employ variables P, Q of type

$$\iota \to ([e] \to t) \to ([e] \to t),$$

variables j, j' of type ι, and variables s, s' of type $[e] \to t$.

We will assume that pronouns are the only NPs that carry indices. Appropriate indices for proper names are extracted from the current state.

The main difference with fragments based on dynamic versions of Montague grammar is that determiners do not carry indices anymore. The appropriate index is provided by the size of the input state, i.e., the length of contexts in the input state.

For convenience, we will assume that all proper names are linked to anchored elements in context. The incrementality of the context update mechanism ensures that no anchored elements can ever be overwritten.

We start out from basic types for booleans and entities. Contexts get represented as lists of entities. Propositions are lists of contexts. Transitions are maps from contexts to propositions. Indices are integers:

```
data Entity = A | B | C | D | E | F | G | H | I | J | K | L | M
     deriving (Eq,Bounded,Ord,Enum,Show)

type Context    = [Entity]
type State      = [Context]
type Transition = State -> State
```

6 Index Lookup and Context Extension

`lookupIdx` is the implementation of $c[i]$.

```
lookupIdx :: Context -> Idx -> Entity
lookupIdx []     i = error "undefined context element"
lookupIdx (x:xs) 0 = x
lookupIdx (x:xs) i = lookupIdx xs (i-1)
```

`extend` is the implementation of $s\hat{}x$.

```
extend :: State -> Entity -> State
extend = \ s e -> [ c ++ [e] | c <- s ]
```

`size` gives the size of the contexts in a state. `size s` implements $\#s$.

```
size :: State -> Int
size []     = 0
size (c:cs) = length c
```

`test` lifts booleans to transitions:

```
test :: Bool -> Transition
test = \ b s -> if b then s else []
```

7 Tools for Plurality and Quantification

A type for collections of entities:

```
type Coll = [Entity]
```

Collections are formed from a state and an index by collecting the individuals at the index in the state. `(sort . nub)` removes the duplicates and sorts the list. `extension s i` is the implementation of $s[i]$.

```
extension :: State -> Idx -> Coll
extension s i = (sort . nub) [ lookupIdx c i | c <- s ]
```

Counting the sizes of collections: `count s i` is the implementation of $|s[i]|$.

```
count :: State -> Idx -> Int
count s i = length (extension s i)
```

Testing whether a transition is uniquely satisfied at an index:

```
unique :: Idx -> Transition -> Transition
unique i f s = if count (f s) i == 1 then (f s) else []
```

Our main tool for dealing with quantification will be the following function for computing generalized quantifier number pairs, the first one for the (lifted) CN denotations, the second one for the (lifted) CN $\cap$ VP denotations. We assume that we compare the extension at an index after a transition f (f corresponding to the CN denotation) with the extension at the same index after a further transition g (g corresponding to the VP denotation). Assume that s is the initial state. `gquant` implements the function gq.

```
gquant :: Transition -> Transition -> Idx -> State -> (Int,Int)
gquant f g i s = (m,k)
   where
   m = count (f s) i
   k = count (g (f s)) i
```

8 Negation, Conjunction, Implication, Quantification

Here are the implementations of $\neg$, **;** , $\Rightarrow$, $\exists$.

```
neg :: Transition -> Transition
neg = \ phi s -> if phi s  == [] then s else []

conj :: Transition -> Transition -> Transition
conj = flip (.)

impl :: Transition -> Transition -> Transition
impl = \ phi psi ->  neg (phi ‘conj‘ (neg psi))

exists :: Transition
exists = \ s -> nub (concat [ (extend s x) | x <- [minBound..maxBound]])
```

Universal quantification can be expressed in terms of negation, conjunction and existential quantification:

```
forall :: Transition -> Transition
forall = \ phi -> neg (exists ‘conj‘ (neg phi))
```

9 Anchors for Proper Names

The anchors for proper names are extracted from the following initial state.

```
start :: State
start = [[A,M,B,J]]
```

Anchoring a set X is just a matter of finding an index with X as its extension. Anchoring a singular proper name n is just a matter of anchoring the set $\{n'\}$, where n' is the interpretation of n.

```
anchor :: Coll -> State -> Idx
anchor = \ xs s -> anchor’ xs s 0 where
   anchor’ xs s i | i >= size s           = error (show xs ++
                                              " not anchored in state")
                  | extension s i == xs   = i
                  | otherwise             = anchor’ xs s (i+1)
```

10 Model Information

Our model has named entities, one-placed predicates and two-placed predicates. Names are interpreted as entities in the model.

```
ann, mary, bill, johnny :: Entity
ann    = A
mary   = M
bill   = B
johnny = J
```

One-placed predicates are interpreted as functions of type `Entity -> Bool`.

```
man,woman,boy,person,thing,house,cat,mouse,laugh,cry,curse,smile,old,young
    :: Entity -> Bool
man x    = x == B || x == D || x == J
woman x  = x == A || x == C || x == M
boy x    = x == J
person x = man x || woman x
thing x  = not (person x)
house x  = x == H
cat x    = x == K
mouse x  = x == I
laugh x  = x == M || x == C || x == B || x == D || x == J
cry   x  = x == B || x == D
curse x  = x == J || x == A
smile x  = x == M || x == B || x == I || x == K
young x  = x == J
old x    = x == B || x == D
```

Two-placed predicates are interpreted as functions of type `Entity -> Entity -> Bool`.

```
love,respect,hate,own :: Entity -> Entity -> Bool
love x y    =  ((x == M || x == A) && y == B)
            || ((x == J || x == B) && woman y)
respect x y = person x && person y || y == F || y == I
hate x y    = (thing x && (y == B || y == J)) || (x == K && y == I)
own x y     =  (x == E && y == A) || ((x == K || x == H) && y == M)
```

11 Lexical Meaning

The lexical meanings of VPs and CNs are one-placed predicates, those of TVs two-placed predicates. These lexical meanings are blown up to the appropriate discourse types.

Assume A to be an expression of type $e \to t$, let s be a variable of type $[e] \to t$ and let j be a variable of type ι. Then the lifting operation for verb phrase meanings looks like this:

$$A^\circ \quad := \quad \lambda js.f, \text{ where } f \text{ characterizes } \{c \in s \mid Ac[j]\}$$

Mapping one-placed predicates to functions from indices to transitions (or: indexed transitions) is done by:

```
blowupPred :: (Entity -> Bool) -> Idx -> Transition
blowupPred pred i s = [ c | c <- s, pred (lookupIdx c i)]
```

Let B be an expression of type $e \to e \to t$, s a variable of type $[e] \to t$, c a variable of type $[e]$, and j, j' variables of type ι. Then the lifting operation for transitive verb meanings looks like this:

$$B^{\bullet} \quad := \quad \lambda j j' s.g, \text{ where } g \text{ characterizes } \{c \in s \mid Bc[j]c[j']\}$$

Discourse blow-up for two-placed predicates is done as follows:

```
blowupPred2 :: (Entity -> Entity -> Bool) -> Idx -> Idx -> Transition
blowupPred2 pred i j s =
  [ c | c <- s, pred (lookupIdx c i) (lookupIdx c j)]
```

Interpretation of VPs consisting of a TV with a reflexive pronoun uses the relation reducer `self`. Note the polymorphism of this definition. We will use the relation reducer on relations in type `Idx -> Idx -> Transition` rather than `Entity -> Entity -> Bool`.

```
self :: (a -> a -> b) -> a -> b
self = \ p x -> p x x
```

12 Dynamic Interpretation

The interpretation of sentences, in type `S -> Trans`, goes according to the following rules:

$$\begin{array}{lll@{\qquad}lll}
\mathbf{S} & ::= & \mathbf{NP}\ \mathbf{VP} & X & ::= & (X_1 X_2) \\
\mathbf{S} & ::= & \textit{if}\ \mathbf{S}\ \ \mathbf{S} & X & ::= & X_2 \Rightarrow X_3 \\
\mathbf{S} & ::= & \mathbf{S}\ .\ \mathbf{S} & X & ::= & X_1\ \mathbf{;}\ X_3
\end{array}$$

```
intS :: S -> Transition
intS (S np vp) = (intNP np) (intVP vp)
intS (If s1 s2) = (intS s1) `impl` (intS s2)
intS (Txt s1 s2) = (intS s1) `conj` (intS s2)
```

Interpretations of proper names according to the following rule:

$$\mathbf{NP} \quad ::= \quad \textit{Mary} \qquad X \quad ::= \quad \lambda Ps.Pjs \text{ where } s[j] = \{m\}$$

```
intNP :: NP -> (Idx -> Transition) -> Transition
intNP Mary p s = p (anchor [mary] s) s
intNP Ann  p s = p (anchor [ann] s) s
intNP Bill p s = p (anchor [bill] s) s
intNP Johnny p s = p (anchor [johnny] s) s
```

Interpretation of singular pronouns according to the following rule:

$$\mathbf{NP} \quad ::= \quad PRO^{i}_{\text{sing}} \qquad X \quad ::= \quad \lambda Ps.Pis$$

Interpretation of plural pronouns: we will assume that a plural pronoun *they*$_i$ means the same as *all things at index i*. Thus, we bring in the generalized quantifier function for its interpretation:

$$\begin{array}{lllll}\mathbf{NP} & ::= & PRO^{i}_{\text{plur}} & X & ::= \lambda Ps. \text{ if the } i\text{-collection survives the } P \text{ update} \\ & & & & \quad \text{then } P\ i\ s \\ & & & & \quad \text{else } f_{\emptyset}\end{array}$$

```
intNP (PRO agr i) p s = if elem Sg agr then p i s
                        else if m == k then p i s
                        else []
   where
   f      = id
   g      = (p i)
   (m,k)  = gquant f g i s
```

Interpretation of complex NPs as specified by the following rules:

$$\begin{array}{llllll}\mathbf{NP} & ::= & \mathbf{DET\ CN} & X & ::= & (X_1X_2) \\ \mathbf{NP} & ::= & \mathbf{DET\ RCN} & X & ::= & (X_1X_2)\end{array}$$

```
intNP (NP1 _ det cn) p s = (intDET det) (intCN cn) p s
intNP (NP2 _ det rcn) p s = (intDET det) (intRCN rcn) p s
```

Interpretation of `(VP1 TV NP)` as given by the following rule.

$$\mathbf{VP} \quad ::= \quad \mathbf{TV\ NP} \qquad X \quad ::= \quad \lambda j.(X_2 \mathbin{;} \lambda j'.((X_1\ j')j))$$

Interpretation of `(VP2 TV REFL)` uses the relation reducer `self`. Interpretation of lexical VPs uses discourse blow-up from the lexical meanings.

$$\mathbf{VP} \quad ::= \quad \textit{laugh} \qquad X \quad ::= \quad L^{\circ}$$

```
intVP :: VP -> Idx -> Transition
intVP (VP1 tv np)   = \ subj -> intNP np (\ obj -> intTV tv obj subj)
intVP (VP2 _ tv _)  = self (intTV tv)
intVP (Laugh _)     = blowupPred laugh
intVP (Cry _)       = blowupPred cry
intVP (Smile _)     = blowupPred smile
intVP (Curse _)     = blowupPred curse
```

Interpretation of TVs uses discourse blow-up of two-placed predicates.

$$\mathbf{TV} \quad ::= \quad \textit{respect} \qquad X \quad ::= \quad R^{\bullet}$$

```
intTV :: TV -> Idx -> Idx -> Transition
intTV (Love _)    = blowupPred2 love
intTV (Respect _) = blowupPred2 respect
intTV (Hate _)    = blowupPred2 hate
intTV (Own _)     = blowupPred2 own
```

Interpretation of CNs uses discourse blow-up of one-placed predicates.

$$\mathbf{CN} \quad ::= \quad \textit{woman} \qquad X \quad ::= \quad W^{\circ}$$

Singular and plural CNs get the same meanings, so the number argument is disregarded.

```
intCN :: CN -> Idx -> Transition
intCN (Man _)    = blowupPred man
intCN (Boy _)    = blowupPred boy
intCN (Woman _)  = blowupPred woman
intCN (Person _) = blowupPred person
intCN (Thing _)  = blowupPred thing
intCN (House _)  = blowupPred house
intCN (Cat _)    = blowupPred cat
intCN (Mouse _)  = blowupPred mouse
intCN (ACN adj cn) = \ i -> (intADJ adj i) . (intCN cn i)
```

Adjectives:

```
intADJ Old   = blowupPred old
intADJ Young = blowupPred young
```

Discourse type of determiners: combine two context predicates into a transition.

```
intDET :: DET ->
          (Idx -> Transition) -> (Idx -> Transition) -> Transition
```

Interpretation of singular determiners in terms of dynamic quantification `exists`, dynamic negation `neg`, dynamic conjunction `conj`, and dynamic uniqueness check `unique`. Note that the index i is derived from the input state.

DET	::=	*every*	X	::=	$\lambda PQs.(\neg(\exists\ ;\ P\#(s)\ ;\ \neg Q\#(s)))s$
DET	::=	*some*	X	::=	$\lambda PQc.(\exists\ ;\ P\#(s)\ ;\ Q\#(s))s$
DET	::=	*no*	X	::=	$\lambda PQc.(\neg(\exists\ ;\ P\#(s)\ ;\ Q\#(s))s$
DET	::=	*the*	X	::=	$\lambda PQs.$ $((\lambda s'.s = s' \wedge \exists x \forall y (\Downarrow (P\#(s)\ (s\hat{\ }y)) \leftrightarrow x = y))$ $;\ \exists\ ;\ P\#(s)\ ;\ Q\#(s))s$

```
intDET (ALL [Sg]) phi psi s = let i = size s in
       neg (exists `conj` (phi i) `conj` (neg (psi i))) s
intDET (NO [Sg]) phi psi s = let i = size s in
       neg (exists `conj` (phi i) `conj` (psi i)) s
intDET (THE [Sg]) phi psi s = let i = size s in
       ((unique i (exists `conj` (phi i))) `conj` (psi i)) s
intDET (SOME [Sg]) phi psi s = let i = size s in
                 (exists `conj` (phi i) `conj` (psi i)) s
```

Interpretation of plural noun phrases uses the generalized quantifier function.

$$\begin{array}{llllll} \textbf{DET} & ::= & D[+pl] & X & ::= & \lambda PQsc.\ c \in (\exists; P\#(s); Q\#(s))s \wedge \\ & & & & & \qquad Q_D(\text{gq}\ (\exists; P\#(s))\ (Q\#(s))\ \#(s)\ s)) \end{array}$$

```
intDET (THE [Pl]) phi psi s = if m == k then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (m,k)  = gquant f g i s
```

```
intDET (ALL [Pl]) phi psi s = if m == k then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (m,k)  = gquant f g i s
```

```
intDET (SOME [Pl]) phi psi s = if k /= 0 then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (_,k)  = gquant f g i s
```

```
intDET (NO [Pl]) phi psi s = if k == 0 then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = s
   (_,k)  = gquant f g i s
```

```
intDET (LESS n) phi psi s = test (k < n) s
   where
   i = size s
   f = exists `conj` (phi i)
   g = (psi i)
   (_,k) = gquant f g i s
```

```
intDET (MORE n) phi psi s = if k > n then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (_,k)  = gquant f g i s
```

```
intDET (EXACT n) phi psi s = if k == n then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (_,k)  = gquant f g i s
```

```
intDET MOST phi psi s = if 2 * k > m  then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   g      = (psi i)
   result = g (f s)
   (m,k)  = gquant f g i s
```

The interpretation of relativised common nouns is as given by the following rules:

RCN	::=	**CN** *that* **VP**	X	::=	$\lambda j.((X_1\ j)\ ;\ (X_3\ j))$
RCN	::=	**CN** *that* **NP TV**	X	::=	$\lambda j.((X_1\ j)\ ;\ (X_3(\lambda j'.((X_4\ j')j))))$

```
intRCN :: RCN -> Idx -> Transition
intRCN (CN1 cn vp)    = \ i -> conj (intCN cn i) (intVP vp i)
intRCN (CN2 cn np tv) = \ i -> conj (intCN cn i)
                                    (intNP np (intTV tv i))
```

13 Testing It Out

The initial context from which evaluation can start is given by `start`. The function `parse` parses an input string. Its definition is given in the second appendix.

The parse function returns a list of data structures of type `S`. Failure to find a parse is indicated with `[]`.

```
eval :: String -> [State]
eval = \ str -> map (\x -> intS x start) (parse str)
```

The second appendix gives sets of example sentences, for use as a test suite.

14 Related Work

Typed Logics for Incremental Processing Dynamic versions of Montague grammar use some version of typed logic. Most typed logics assume a basic type for discourse referents or indices, and impose suitable axioms to ensure that these behave like the variables of dynamic predicate logic. Examples of this approach are [12, 5, 19, 29, 30, 28, 6, 9, 24, 25, 3].

Closest to our approach is [3], where a typed dynamic logic is proposed that handles the singular/plural distinction and treats anaphoric linking. The main difference is that [3] follows the lead of DPL ([13]), whereas we remain much closer to the spirit of DRT. Because of the reliance on dynamic variable binding, [3] has trouble with reuse of the same dynamically bound variable. In order to solve these difficulties, a predicate *free* is introduced that applies to those reference markers that have not yet been constrained in the present model. Such mix-up of syntax and semantics is avoided in our approach.

Rational Reconstructions of DRT When dynamic semantics for NL first was proposed in [21] and [17], the approach invoked strong opposition from the followers of Montague [27]. Rational reconstructions to restore compositionality were announced in [13] and carried out in the papers mentioned above. All of these reconstructions are based in some way or other on DPL [13], and they all inherit the main flaw of this approach: the destructive assignment problem. Interestingly, DRT itself did not suffer from this problem: the discourse representation construction algorithms of [21] and [22] are stated in terms of functions with finite domains, and carefully talk about 'taking a fresh discourse referent' to extend the domain of a verifying function, for each new NP to be processed.

The present approach, based on ID rather than DPL, makes clear how the instruction to take fresh discourse referents when needed can be made fully precise by using the standard toolset of (polymorphic) type theory. To our knowledge this is the first reconstruction of DRT in type theory that does justice to the incrementality and the finite state semantics of the original.

The Centering Approach to Reference Resolution Central claim of the *centering theory* of local coherence in discourse [16, 15] is that pronouns are used to signal to the hearer that the speaker continues to talk about the same thing. See [40] for extensions and variations that take world knowledge into account, and [2] for a reformulation in terms of optimality theory. [10] demonstrates that reference resolution can be brought within the compass of dynamic semantics in a relatively straightforward way, and that very simple means are enough to implement something quite useful.

Visser-Style Contexts and Referent Systems Visser's context theories [36, 38, 37, 39] also start out as rational reconstructions of DRT. Visser's view of contexts is considerably more abstract than the simple-minded approach taken here. Our approach illustrates how little one has to assume about contexts for a working system.

Referent systems [34, 33, 35] are a mechanism for indirect reference to objects, via variable names and indices ('pegs'). Referent systems are employed in [14] to reformulate the logic of DPL in an incremental fashion in order to solve certain puzzles of epistemic modality in dynamic semantics, and in [1] in a sketch of a logic of anaphora resolution. We hope to have shown that reliance on context leads to a much simpler set-up of incremental dynamic logic, with context indices as 'pegs'. In a sense, contexts or contexts under permutation are what is left of reference systems when one leaves out the variable names.

15 Loose Ends and Future Work

CN anaphora Strictly speaking, quantificational sentences are dynamic in two ways. First of all they output the intersection of restrictor and scope and second they make the restrictor itself available for future anaphoric reference.

The status of this latter kind of reference is unclear. Kamp and Reyle [22] analyse the possibility of this kind of reference as reference to some sort of genus introduced by the quantification, but a more specific interpretation seems to be available as well. In (4), the plural pronoun can both refer to students in general and to the specific set of students quantified over in the first sentence.

4 *Few*$_{1,2}$ *students did their homework. They*$_{1/2}$ *are so lazy!*

For simplicity we have not made the determiners dynamically output a restrictor value, although this is very easy to acchieve. As an example, we give the modified code for *most*.

```
intDET MOST' phi psi s = if 2 * k > m  then result else []
   where
   i      = size s
   f      = exists `conj` (phi i)
   j      = size (f s)
   g      = exists `conj` (phi j) `conj` (psi j)
   result = g (f s)
   (m,k)  = gquant f g j s
```

Functional dependency and distribution

5 *Most*$_1$ *students have a pet*$_2$*. They*$_1$ *adore it*$_2$*.*

Examples like (5) pose a problem. The example comes out true in the current set-up if and only if most students have a pet and the students that own a pet each adore at least one pet owned by some student. In the required reading each student adores her own pet. The problem is caused by our naive treatment of distribution. Instead of distributing over the collection which forms the extension of some index (1 in the example above), there should be distribution over the contexts in the state. So Pij is true in state S if and only if for all contexts c in state S, $M \models Pc(i)c(j)$. This gets the functional dependency right: every pet-owning student now adores her own pet. See [4] for an extensive discussion of various modes of distribution.

Anaphoric definite descriptions Thus far, pronouns have been the only anaphoric items. We can, however, easily extend the semantics to anaphoric uses of definite descriptions. We give here the plural case. Because of the semantic content of descriptions, they can be subsectionally anaphoric. That is, they can access parts of sets assigned distributively to an index. Therefore, anaphoric definite descriptions need to introduce an index of their own. Otherwise, they behave exactly like plural pronouns.

```
intDET (THEA j) phi psi s = if m == k then result else []
   where
   i      = size s
   f      = exists ‘conj‘ (eq i j) ‘conj‘ (phi i)
   g      = (psi i)
   result = g (f s)
   (m,k)  = gquant f g i s
```

Here is an example of use of an anaphoric definite.

```
example = (S (NP1 MOST Man) Laughed) ‘Txt‘
          (S (NP1 (THEA 4) (ACN Old Man)) Cried)
```

Evaluating this example with `eval` will result in a state of contexts of length 5, since all old men who laughed also cried. The members of the set of laughing old men are assigned to index 5.

Adjectival quantifiers and non-maximal anaphora Since we have made the minimal design choice of representing plurality as set-values distributed across the contexts in a state, we will not be able to handle adjectival quantifiers and non-maximal anaphora.

6 *Four*$_1$ *women entered the bar yesterday. They*$_1$...

This example has a reading where there were more than four women entering the bar yesterday. This means the adjectival quantifier is not exhaustive; nor are subsequent plural anaphora.

Van den Berg, using a notion of state similar to the one adopted here, *does* handle these cases ([4]:174). He considers the first sentence in the example above as a non-unique update, with multiple possible output states. The quantifier *four women*, then, behaves like an indefinite. However, to work this out one has to turn to a relational (and thus non-computational) view on states (as does van den Berg) or lift the information states once more. Multiplication of states would cause a considerable increase in complexity, so we have chosen to leave the problem unsolved.

More on the maximal/non-maximal distinction in [31]. A study of the relation between (speaker's) specificity and exhaustivity in dynamic semantics can be found in [32].

Downward entailing quantifiers Another unsolved problem is the treatment of downward entailing quantifiers. There are essentially two issues. First of all, since we have chosen to make quantifiers dynamic by *existentially* introducing a new index, we presuppose the existence of a set of individuals satisfying both restrictor and scope of the quantifier relation. Of course, in case the determiner is downward monotone in its right argument, this relation is satisfied even if no such individual exists. The second issue is more complicated and does not realy affect the semantics presented here since it involves collective quantification, which the system does not deal with anyway. The use of $s[i]$, the set of all values at index i in state s, in the definition of generalized quantifier gave us a natural way of ensuring that the sets exported and compared by the quantifier are *maximal*. With collective quantification more than one maximum is possible for the intersection of restrictor and scope. But while for increasing quantifiers it suffices if one of these satisfies the quantifier relation, for decreasing quantifiers *all* maxima should. Crucially then, the universal quantification needed to derive the proper truth-conditions destroys the dynamics of the quantifier. We will not resolve these issues here, but it should be noted that we share these two problems with many other approaches, among which [4].

References

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

[2] D. Beaver. The optimization of discourse. Manuscript, Stanford University, July 2000.

[3] D. Bekki. Typed dynamic logic for E-type link. In *Proceedings for Third International Conference on Discourse Anaphora and Anaphor Resolution (DAARC2000)*, pages 39–48. Lancaster University, U.K., 2000.

[4] M.H. van den Berg. *The Internal Structure of Discourse; The Dynamics of Nominal Anaphora.* PhD thesis, ILLC, Amsterdam, March 1996.

[5] G. Chierchia. Anaphora and dynamic binding. *Linguistics and Philosophy*, 15(2):111–183, 1992.

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

[7] J. van Eijck. The proper treatment of context in NL. In Paola Monachesi, editor, *Computational Linguistics in the Netherlands 1999; Selected Papers from the Tenth CLIN Meeting*, pages 41–51. Utrecht Institute of Linguistics OTS, 2000.

[8] J. van Eijck. Incremental dynamics. *Journal of Logic, Language and Information*, 10:319–351, 2001.

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

[10] Jan van Eijck. Context semantics for NL. Lecture Note, Uil-OTS, November 2000.

[11] Jan van Eijck. Dynamic semantics for NL. Lecture Note, Uil-OTS, November 2000.

[12] J. Groenendijk and M. Stokhof. 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, 1990.

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

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

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

[16] B.J. Grosz and C.L. Sidner. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12:175–204, 1986.

[17] I. Heim. *The Semantics of Definite and Indefinite Noun Phrases.* PhD thesis, University of Massachusetts, Amherst, 1982.

[18] J. Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1997.

[19] Martin Jansche. Dynamic Montague Grammar lite. Dept of Linguistics, Ohio State University, November 1998.

[20] S. Peyton Jones, J. Hughes, et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

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

[22] H. Kamp and U. Reyle. *From Discourse to Logic.* Kluwer, Dordrecht, 1993.

[23] D.E. Knuth. *Literate Programming.* CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

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

[25] S. Kuschert. *Dynamic Meaning and Accommodation.* PhD thesis, Universität des Saarlandes, 2000. Thesis defended in 1999.

[26] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

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

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

[29] R. Muskens. Tense and the logic of change. In U. Egli et al., editor, *Lexical Knowledge in the Organization of Language*, pages 147–183. W. Benjamins, 1995.

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

[31] Anna Szabolsci. Strategies for scope taking. In Anna Szabolsci, editor, *Ways of scope taking*, pages 109–154. Kluwer Academic Publishers, 1997.

[32] Robert van Rooy. Exhaustivity in dynamic semantics; referential and descriptive pronouns. *Linguistics and philosophy*, 24(5), October 2001.

[33] C.F.M. Vermeulen. Sequence semantics for dynamic predicate logic. *Journal of Logic, Language, and Information*, 2:217–254, 1993.

[34] C.F.M. Vermeulen. *Explorations of the Dynamic Environment.* PhD thesis, OTS, Utrecht, 1994.

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

[36] A. Visser. Prolegomena to the definition of dynamic predicate logic with local assignments. Technical Report 178, Utrecht Research Institute for Philosophy, October 1997.

[37] A. Visser. Freut sich die Lies — context modification in action. Lecture Notes, Department of Philosophy, Utrecht University, November 2000.

[38] A. Visser. Heut kommt der Hans nach Haus — a study of contexts, brackets, arguments & files. Manuscript, Department of Philosophy, Utrecht University, February 2000.

[39] A. Visser and C. Vermeulen. Dynamic bracketing and discourse representation. *Notre Dame Journal of Formal Logic*, 37:321–365, 1996.

[40] M. Walker, A. Joshi, and E. Prince, editors. *Centering Theory in Discourse.* Clarendon Press, 1998.

Appendix A: Syntactic Datastructures for the Fragment

No index information on NPs, except for pronouns. Otherwise, virtually the same as a datatype declaration for a fragment of dynamic Montague grammar. The module `Cat` imports the standard `List` module. Lists will be employed to implement a simple feature agreement mechanism.

```
module Cat

where

import List
```

Define features, feature lists, indices, and numerals.

```
data Feature =   Masc | Fem | Neutr
               | Sg   | Pl
               | Fst  | Snd | Thrd
               | Nom  | Acc
             deriving (Eq,Ord)

instance Show Feature
  where
  show Masc  = "M"
  show Fem   = "F"
  show Neutr = "N"
  show Sg    = "Sg"
  show Pl    = "Pl"
  show Fst   = "1"
  show Snd   = "2"
  show Thrd  = "3"
  show Nom   = "n"
  show Acc   = "a"

type Agreement = [Feature]
type Idx       = Int
type Numeral   = Int
```

Selecting the gender, number, person and case part of a feature list:

```
gen, nr, ps, cs :: Agreement -> Agreement
gen = filter (\x -> x == Masc || x == Fem || x == Neutr)
nr  = filter (\x -> x == Sg || x == Pl)
ps  = filter (\x -> x == Fst || x == Snd || x == Thrd)
cs  = filter (\x -> x == Nom || x == Acc)
```

Declare a class `Cat` for categories that carry number and gender information, with a function `fs` that gives the feature list of a category, functions `gender`, `number`, `sperson` and `scase` for the syntactic gender, number, person and case features of the category, a function `combine` that computes the features of a combined category (with feature clashes reported by `[]`), and a function `agree` indicating whether there is a feature clash or not when two categories are combined.

```
class (Eq a, Show a) => Cat a where
   fs  :: a -> Agreement
   gender, number, sperson, scase  :: a -> Agreement
   gender cat = gen (fs cat)
   number cat = nr (fs cat)
   sperson cat = ps (fs cat)
   scase  cat = cs (fs cat)
   combine :: Cat b => a -> b -> [Agreement]
   combine cat1 cat2 = [ feats | length (gen feats) <= 1,
                                 length (nr feats)  <= 1,
                                 length (ps feats)  <= 1,
                                 length (cs feats)  <= 1   ]
     where
     feats = (nub . sort) (fs cat1 ++ fs cat2)
   agree :: Cat b => a -> b -> Bool
   agree cat1 cat2 = not (null (combine cat1 cat2))
```

Sentences are in class `Cat`. Set the agreement lists of ‘if then’ sentences and texts consisting of several sentences to `[]`.

```
data S = S NP VP | If S S | Txt S S
     deriving (Eq,Show)

instance Cat S
  where
  fs (S np vp) = fs vp
  fs _         = []
```

Pronouns and complex NPs carry explicit feature information. The feature information of proper names depends on the name.

```
data NP = Ann | Mary | Bill | Johnny
        | PERS Agreement
        | PRO  Agreement Idx
        | NP1 Agreement DET CN
        | NP2 Agreement DET RCN
     deriving (Eq,Show)

instance Cat NP
    where
    fs Ann          = [Fem,Sg,Thrd]
    fs Mary         = [Fem,Sg,Thrd]
    fs Bill         = [Masc,Sg,Thrd]
    fs Johnny       = [Masc,Sg,Thrd]
    fs (PERS ftrs)  = ftrs
    fs (PRO ftrs i)  = ftrs
    fs (NP1 ftrs det cn)  = ftrs
    fs (NP2 ftrs det rcn) = ftrs
```

The entries `ALL`, `SOME`, `NO`, `THE` are for both singular and plural determiners, so they carry no number feature information. The entries `LESS` and `MOST` are for plural determiners. The number feature of `MORE` and `EXACT` depends on the numeral.

```
data DET = ALL Agreement | SOME Agreement | NO Agreement | THE Agreement
         | LESS Numeral | MORE Numeral | EXACT Numeral | MOST
     deriving (Eq,Show)
```

We only set the number feature.

```
instance Cat DET
    where
    fs (ALL ftrs)  = ftrs
    fs (SOME ftrs) = ftrs
    fs (NO ftrs)   = ftrs
    fs (THE ftrs)  = ftrs
    fs (LESS i)    = [Pl]
    fs (MORE 1)    = [Sg]
    fs (MORE i)    = [Pl]
    fs (EXACT 1)   = [Sg]
    fs (EXACT i)   = [Pl]
    fs MOST        = [Pl]
```

We make a syntactic distinction between singular and plural versions of CNs and RCNs, although their semantic treatment will be the same.

```
data CN = Man Agreement   | Woman Agreement
        | Boy Agreement   | Person Agreement
        | Thing Agreement | House Agreement
        | Cat Agreement   | Mouse Agreement
        | ACN ADJ CN
     deriving (Eq,Show)

instance Cat CN
  where
  fs (Man ftrs)    = ftrs
  fs (Woman ftrs)  = ftrs
  fs (Boy ftrs)    = ftrs
  fs (Person ftrs) = ftrs
  fs (Thing ftrs)  = ftrs
  fs (House ftrs)  = ftrs
  fs (Cat ftrs)    = ftrs
  fs (Mouse ftrs)  = ftrs
  fs (ACN adj cn)  = fs cn
```

```
data ADJ = Old | Young
     deriving (Eq,Show)

instance Cat ADJ
  where
  fs Old   = []
  fs Young = []

data RCN = CN1 CN VP | CN2 CN NP TV
     deriving (Eq,Show)

instance Cat RCN
  where
  fs (CN1 cn vp)    = fs cn
  fs (CN2 cn np tv) = fs cn
```

We make a syntactic distinction between singular and plural versions of VPs and TVs, although their semantic treatment will be the same.

```
data VP = Laugh Agreement | Cry Agreement | Curse Agreement
        | Smile Agreement
        | VP1 TV NP | VP2 Agreement TV REFL
     deriving (Eq,Show)

instance Cat VP
  where
  fs (Laugh ftrs)       = ftrs
  fs (Cry ftrs)         = ftrs
  fs (Curse ftrs)       = ftrs
  fs (Smile ftrs)       = ftrs
  fs (VP1 tv np)        = fs tv
  fs (VP2 ftrs tv refl) = ftrs

data REFL = Self Agreement deriving (Eq,Show)

instance Cat REFL
  where fs (Self ftrs) = ftrs
```

Transitive verbs carry a number feature, so they are in the class `Cat`.

```
data TV = Love Agreement | Respect Agreement
        | Hate Agreement | Own Agreement
     deriving (Eq,Show)

instance Cat TV
  where
  fs (Love ftrs)    = ftrs
  fs (Respect ftrs) = ftrs
  fs (Hate ftrs)    = ftrs
  fs (Own ftrs)     = ftrs
```

Appendix B: A Simple CF Parser

```
module Parser
where
import Cat
```

```
type Words = [String]
```

NPs

```
lexNP :: Words -> [(NP,Words)]
lexNP ("ann":xs)    = [(Ann,xs)]
lexNP ("mary":xs)   = [(Mary,xs)]
lexNP ("bill":xs)   = [(Bill,xs)]
lexNP ("johnny":xs) = [(Johnny,xs)]
lexNP ("i":xs)      = [(PERS [Sg,Fst,Nom],xs)]
lexNP ("me":xs)     = [(PERS [Sg,Fst,Acc],xs)]
lexNP ("we":xs)     = [(PERS [Pl,Fst,Nom],xs)]
lexNP ("us":xs)     = [(PERS [Pl,Fst,Acc],xs)]
lexNP ("you":xs)    = [(PERS [Snd],xs)]
lexNP ("he":x:xs)   = [((PRO [Masc,Sg,Thrd,Nom] (read x)),xs)]
lexNP ("him":x:xs)  = [((PRO [Masc,Sg,Thrd,Acc] (read x)),xs)]
lexNP ("she":x:xs)  = [((PRO [Fem,Sg,Thrd,Nom] (read x)),xs)]
lexNP ("her":x:xs)  = [((PRO [Fem,Sg,Thrd,Acc] (read x)),xs)]
lexNP ("it":x:xs)   = [((PRO [Neutr,Sg,Thrd] (read x)),xs)]
lexNP ("they":x:xs) = [((PRO [Pl,Thrd,Nom] (read x)),xs)]
lexNP ("them":x:xs) = [((PRO [Pl,Thrd,Acc] (read x)),xs)]
lexNP _             = []
```

```
parseNP :: Words -> [(NP,Words)]
parseNP = \xs ->
    [ (NP1 agr det cn,zs) | (det,ys) <- parseDET xs,
                            (cn, zs) <- parseCN ys,
                             agr     <- combine det cn    ]
    ++
    [ (NP2 agr det rcn,zs) | (det,ys)  <- parseDET xs,
                             (rcn, zs) <- parseRCN ys,
                             agr       <- combine det rcn  ]
    ++
    [ (np,ys) | (np,ys) <- lexNP xs ]
```

Determiners

Note that we need a distinction in the lexicon between singular and plural *some*, *no* and *the*, because of the semantic distinction.

```
lexDET :: Words ->[(DET,Words)]
lexDET ("every":xs)         = [(ALL [Sg], xs)]
lexDET ("all":xs)           = [(ALL [Pl], xs)]
lexDET ("some":xs)          = [(SOME [Sg], xs),(SOME [Pl], xs)]
lexDET ("no":xs)            = [(NO [Sg], xs), (NO [Pl], xs)]
lexDET ("the":xs)           = [(THE [Sg], xs),(THE [Pl], xs)]
lexDET ("less":"than":x:xs) = [((LESS (read x)), xs)]
lexDET ("more":"than":x:xs) = [((MORE (read x)), xs)]
lexDET ("exactly":x:xs)     = [((EXACT (read x)), xs)]
lexDET ("most":xs)          = [(MOST, xs)]
lexDET _                    = []
```

```
parseDET :: Words -> [(DET,Words)]
parseDET = lexDET
```

ADJs

```
lexADJ :: Words -> [(ADJ,Words)]
lexADJ ("old":xs)   = [(Old,xs)]
lexADJ ("young":xs) = [(Young,xs)]
lexADJ ("other":xs) = [(Other,xs)]
lexADJ _            = []
```

```
parseADJ :: Words -> [(ADJ,Words)]
parseADJ = lexADJ
```

CNs

Singular and plural CNs get distinguished by means of an appropriate number feature.

```
lexCN :: Words ->[(CN,Words)]
lexCN ("man":xs)    = [(Man [Masc,Sg,Thrd],xs)]
lexCN ("men":xs)    = [(Man [Masc,Pl,Thrd],xs)]
lexCN ("woman":xs)  = [(Woman [Fem,Sg,Thrd],xs)]
lexCN ("women":xs)  = [(Woman [Fem,Pl,Thrd],xs)]
lexCN ("boy":xs)    = [(Boy [Masc,Sg,Thrd],xs)]
lexCN ("boys":xs)   = [(Boy [Masc,Pl,Thrd],xs)]
lexCN ("person":xs) = [(Person [Sg,Thrd],xs)]
lexCN ("persons":xs)= [(Person [Pl,Thrd],xs)]
lexCN ("thing":xs)  = [(Thing [Neutr,Sg,Thrd],xs)]
lexCN ("things":xs) = [(Thing [Neutr,Pl,Thrd],xs)]
lexCN ("house":xs)  = [(House [Neutr,Sg,Thrd],xs)]
lexCN ("houses":xs) = [(House [Neutr,Pl,Thrd],xs)]
lexCN ("cat":xs)    = [(Cat [Neutr,Sg,Thrd],xs)]
lexCN ("cats":xs)   = [(Cat [Neutr,Pl,Thrd],xs)]
lexCN ("mouse":xs)  = [(Mouse [Neutr,Sg,Thrd],xs)]
lexCN ("mice":xs)   = [(Mouse [Neutr,Pl,Thrd],xs)]
lexCN _             = []
```

```
parseCN :: Words -> [(CN,Words)]
parseCN = \xs ->
    [ (cn,ys)| (cn,ys) <- lexCN xs ]
    ++
    [ (ACN adj cn, zs) | (adj,ys) <- parseADJ xs,
                         (cn, zs) <- parseCN  ys ]
```

RCNs

```
parseTHAT :: Words -> [Words]
parseTHAT ("that":xs) = [xs]
parseTHAT _           = []
```

```
parseRCN :: Words -> [(RCN,Words)]
parseRCN = \xs ->
    [ (CN1 cn vp, us) | (cn,ys) <- parseCN xs,
                         zs     <- parseTHAT ys,
                        (vp,us) <- parseVP zs,
                         agree cn vp              ]
    ++
    [ (CN2 cn np tv, vs) | (cn,ys) <- parseCN xs,
                            zs     <- parseTHAT ys,
                           (np,us) <- parseNP zs,
                           (tv,vs) <- parseTV us,
                            agree np tv,
                            notElem Acc (fs np)    ]
```

REFLs

```
parseREFL :: Words -> [(REFL,Words)]
parseREFL ("myself":xs)     = [(Self [Sg,Fst], xs)]
parseREFL ("ourselves":xs)  = [(Self [Pl,Fst], xs)]
parseREFL ("yourself":xs)   = [(Self [Sg,Snd], xs)]
parseREFL ("yourselves":xs) = [(Self [Pl,Snd], xs)]
parseREFL ("himself":xs)    = [(Self [Masc,Sg,Thrd], xs)]
parseREFL ("herself":xs)    = [(Self [Fem,Sg,Thrd], xs)]
parseREFL ("itself":xs)     = [(Self [Neutr,Sg,Thrd], xs)]
parseREFL ("themselves":xs) = [(Self [Pl,Thrd], xs)]
parseREFL _                 = []
```

VPs

```
lexVP :: Words -> [(VP,Words)]
lexVP ("laughs":xs) = [(Laugh [Sg,Thrd],xs)]
lexVP ("laugh":xs)  =
     [(Laugh [Sg,Fst],xs),(Laugh [Sg,Snd],xs),(Laugh [Pl],xs)]
lexVP ("cries":xs)  = [(Cry [Sg,Thrd],xs)]
lexVP ("cry":xs)    =
     [(Cry [Sg,Fst],xs),(Cry [Sg,Snd],xs),(Cry [Pl],xs)]
lexVP ("curses":xs) = [(Curse [Sg,Thrd],xs)]
lexVP ("curse":xs)  =
     [(Curse [Sg,Fst],xs),(Curse [Sg,Snd],xs),(Curse [Pl],xs)]
lexVP ("smiles":xs) = [(Smile [Sg,Thrd],xs)]
lexVP ("smile":xs)  =
     [(Smile [Sg,Fst],xs),(Smile [Sg,Snd],xs),(Smile [Pl],xs)]
lexVP _             = []
```

```
parseVP :: Words -> [(VP,Words)]
parseVP = \xs ->
    [ (VP1 tv np,zs) | (tv,ys) <- parseTV xs,
                       (np,zs) <- parseNP ys,
                        notElem Nom (fs np)    ]
    ++
    [ (VP2 agr tv refl,zs) | (tv,ys)    <- parseTV xs,
                             (refl,zs)  <- parseREFL ys,
                              agr       <- combine tv refl   ]
    ++
    [ (vp,ys)| (vp,ys) <- lexVP xs ]
```

TVs

```
lexTV :: Words ->[(TV,Words)]
lexTV ("loves":xs)    = [(Love [Sg,Thrd],xs)]
lexTV ("love":xs)     =
   [(Love [Sg,Fst],xs), (Love [Sg,Snd],xs), (Love [Pl],xs)]
lexTV ("respects":xs) = [(Respect [Sg,Thrd],xs)]
lexTV ("respect":xs)  =
   [(Respect [Sg,Fst],xs),(Respect [Sg,Snd],xs),(Respect [Pl],xs)]
lexTV ("hates":xs)    = [(Hate [Sg,Thrd],xs)]
lexTV ("hate":xs)     =
   [(Hate [Sg,Fst],xs), (Hate [Sg,Snd],xs), (Hate [Pl],xs)]
lexTV ("owns":xs)     = [(Own [Sg,Thrd],xs)]
lexTV ("own":xs)      =
   [(Own [Sg,Fst],xs),(Own [Sg,Snd],xs),(Own [Pl],xs)]
lexTV _               = []
```

```
parseTV :: Words -> [(TV,Words)]
parseTV = \xs ->
    [ (tv,ys)| (tv,ys) <- lexTV xs ]
```

IF, THEN, '.', ';'

```
parseIF :: Words -> [Words]
parseIF ("if":xs) = [xs]
parseIF _         = []

parseTHEN :: Words -> [Words]
parseTHEN ("then":xs) = [xs]
parseTHEN _           = []

parseC :: Words -> [Words]
parseC (".":xs) = [xs]
parseC (";":xs) = [xs]
parseC _        = []
```

Ss

```
parseS :: Words -> [(S,Words)]
parseS = \xs ->
    [ (S np vp,zs) | (np,ys) <- parseNP xs,
                     (vp,zs) <- parseVP ys,
                      agree np vp,
                      notElem Acc (fs np)      ]
    ++
    [ (If s1 s2,vs) |  ys     <- parseIF xs,
                      (s1,zs) <- parseS ys,
                       us     <- parseTHEN zs,
                      (s2,vs) <- parseS us     ]
```

Text Since the rule $T ::= S \mid T.S$ is left-recursive, we need an extra function for splitting the input word list: `split` gives all the ways to split a list of at least two elements in two non-empty parts.

```
split :: [a] -> [([a],[a])]
split [x,y] = [([x],[y])]
split (x:y:zs) =
    ([x],(y:zs)):(map ( \ (us,vs) -> ((x:us),vs)) (split (y:zs)))
```

```
parseTxt :: Words -> [(S,Words)]
parseTxt = \xs ->
    parseS xs
    ++
    [ (Txt t s,vs) | (ys,zs) <- split xs,
                      us     <- parseC zs,
                     (t,[])  <- parseTxt ys,
                     (s,vs)  <- parseS us    ]
```

The 'parse' function The next function scans an input string and puts whitespace in front of punctuation marks and numerals. This can be used to convert a string like *"He1 loves her2."* to *"He 1 loves her 2 ."*

```
scan :: String -> String
scan [] = []
scan (x:xs) | x == '.' || x == ';' = ' ':x:scan xs
            | isDigit x            = ' ':x: (digits ++ scan rest)
            | otherwise            =     x:scan xs
                           where (digits,rest) = span isDigit xs
```

The main parse function uses the predefined function `words` to split the input into separate words. Punctuation marks and pronoun indices should come out as separate words; we use `scan` for that. Also, for robustness, we convert everything to lowercase.

```
parse :: String  -> [S]
parse string = [ s | (s,["."]) <- parseTxt
                     (words (map toLower (scan string))) ]
```

Now try it out:

```
Parser> parse "Every man loves some woman."
[S (NP1 [M,Sg,3] (ALL [Sg]) (Man [M,Sg,3])) (VP1 (Love [Sg,3]) (NP1 [F,Sg,3]
   (SOME [Sg]) (Woman [F,Sg,3]))))]
Parser> parse "All men love some woman."
[S (NP1 [M,Pl,3] (ALL [Pl]) (Man [M,Pl,3])) (VP1 (Love [Pl]) (NP1 [F,Sg,3]
   (SOME [Sg]) (Woman [F,Sg,3]))))]
Parser> parse "All men love some women."
[S (NP1 [M,Pl,3] (ALL [Pl]) (Man [M,Pl,3])) (VP1 (Love [Pl]) (NP1 [F,Pl,3]
   (SOME [Pl]) (Woman [F,Pl,3]))))]
Parser> parse "Bill loves more than 1 woman."
[S Bill (VP1 (Love [Sg,3]) (NP1 [F,Sg,3] (MORE 1) (Woman [F,Sg,3])))]
Parser> parse "Bill loves more than 1 women."
[]
Parser> parse "Bill loves more than 1 woman. He0 respects them1."
[Txt (S Bill (VP1 (Love [Sg,3]) (NP1 [F,Sg,3] (MORE 1) (Woman [F,Sg,3]))))
     (S (PRO [M,Sg,3,n] 0) (VP1 (Respect [Sg,3]) (PRO [Pl,3,a] 1)))]
Parser>
```

Examples with personal pronouns:

```
pp1  = "I love you."
pp2  = "We respect ourselves."
pp3  = "We respect every woman that respects herself."
pp4  = "You respect yourself."
pp5  = "You respect yourselves."
```

Examples with singular NPs:

```
ex1  = "Johnny smiles."
ex2  = "Bill laughs."
ex3  = "if Bill laughs then Johnny smiles."
ex4  = "Bill laughs. Johnny smiles."
ex5  = "Bill smiles. He1 loves some woman."
ex6  = "The boy loves some woman."
ex7  = "Some man loves some woman that smiles."
ex8  = "Some man respects some woman."
ex9  = "The man loves some woman."
ex10 = "Every man loves some woman."
ex11 = "Every man loves Johnny."
ex12 = "Some woman loves Johnny."
ex13 = "Johnny loves some woman."
ex14 = "Johnny respects some man that loves Mary."
ex15 = "No woman loves Bill."
ex16 = "No woman that hates Johnny loves Bill."
ex17 = "Some woman that respects Johnny loves Bill."
ex18 = "The boy loves Johnny."
ex19 = "He2 loves her1."
ex20 = "He2 respects her1."
ex21 = "If some man loves some woman then he4 respects her5."
ex22 = "Some man loves some woman. He4 respects her5."
ex23 = "Some woman owns some thing."
ex24 = "Some woman owns the house."
ex25 = "Some woman owns the house that Johnny hates."
ex26 = "No man that cries respects himself."
ex27 = "Some man respects himself."
ex28 = "Exactly 1 boy curses."
```

Examples with plurals:

```
px1  = "More than 1 man laughs."
px2  = "More than 2 men laugh."
px3  = "Most men that love some woman smile."
px4  = "Some women cry. No men cry."
px5  = "No men that cry respect themselves."
px6  = "All men cry."
px7  = "The men curse. The women laugh."
px8  = "Most men curse. No women curse."
px9  = "Most men smile. They4 laugh."
px10 = "Most men cry. They4 laugh."
px11 = "More than 1 man laughs. They4 love Mary."
px12 = "Less than 4 men laugh."
px13 = "Less than 4 men laugh. They4 love Mary."
```

Appendix C: An example evaluation

```
Hugs session for:
/share/hugs/lib/Prelude.hs
/share/hugs/lib/Maybe.hs
/share/hugs/lib/List.hs
Cat.lhs
Parser.lhs
Qar.lhs
Type :? for help
Qar> eval "More than 2 men laugh. They4 respect some young boy."
[[[A,M,B,J,B,J],[A,M,B,J,D,J],[A,M,B,J,J,J]]]
Qar> eval "More than 2 men laugh. They4 respect some woman."
[[[A,M,B,J,B,A],[A,M,B,J,D,A],[A,M,B,J,J,A],[A,M,B,J,B,C],[A,M,B,J,D,C],
[A,M,B,J,J,C],[A,M,B,J,B,M],[A,M,B,J,D,M],[A,M,B,J,J,M]]]
Qar>
```

Remember that the state `start` for evaluation consisted of `A,M,B` and `J`. The example "More than 2 men laugh. They4 respect some young boy." is parsed as follows.

```
Qar> parse "More than 2 men laugh. They4 respect some woman."
[Txt (S (NP1 [M,Pl,3] (MORE 2) (Man [M,Pl,3])) (Laugh [Pl]))
(S (PRO [Pl,3,n] 4) (VP1 (Respect [Pl]) (NP1 [F,Sg,3] (SOME [Sg])
(Woman [F,Sg,3])))))]
Qar>
```

In the example, the determiner in the first sentence checks whether the following holds.

$$|((\exists; man4; laugh4)\texttt{start})[4]| > 2$$

The relevant state is built up as follows.

$$\begin{array}{|c|c|c|c|}\hline A&M&B&J\\\hline\end{array}\ \exists\ \begin{array}{|c|c|c|c|c|}\hline A&M&B&J&A\\\hline A&M&B&J&B\\\hline \multicolumn{5}{c}{\vdots}\\ \multicolumn{5}{c}{\vdots}\\\hline A&M&B&J&M\\\hline\end{array}\ man4\ \begin{array}{|c|c|c|c|c|}\hline A&M&B&J&B\\\hline A&M&B&J&D\\\hline A&M&B&J&J\\\hline\end{array}\ laugh4\ \begin{array}{|c|c|c|c|c|}\hline A&M&B&J&B\\\hline A&M&B&J&D\\\hline A&M&B&J&J\\\hline\end{array}$$

The pronoun in the second sentence now checks whether the individuals assigned to index 4 survive after the predication “respect some woman.” In other words, it checks whether:

$$\left(\begin{array}{|c|c|c|c|c|}\hline A&M&B&J&B\\\hline A&M&B&J&D\\\hline A&M&B&J&J\\\hline\end{array}\ \exists; woman5; respect4\ 5\right)[4] \ = \ \begin{array}{|c|c|c|c|c|}\hline A&M&B&J&B\\\hline A&M&B&J&D\\\hline A&M&B&J&J\\\hline\end{array}\ [4]$$

The update “$\exists; woman5; respect4\ 5$” increments the state as follows.

$$\begin{array}{|c|c|c|c|c|}\hline A&M&B&J&B\\\hline A&M&B&J&D\\\hline A&M&B&J&J\\\hline\end{array}\ \exists\ ; woman5\ \begin{array}{|c|c|c|c|c|c|}\hline A&M&B&J&B&A\\\hline A&M&B&J&B&C\\\hline A&M&B&J&B&M\\\hline A&M&B&J&D&A\\\hline A&M&B&J&D&C\\\hline A&M&B&J&D&M\\\hline A&M&B&J&J&A\\\hline A&M&B&J&J&C\\\hline A&M&B&J&J&M\\\hline\end{array}\ respect4\ 5\ \begin{array}{|c|c|c|c|c|c|}\hline A&M&B&J&B&A\\\hline A&M&B&J&B&C\\\hline A&M&B&J&B&M\\\hline A&M&B&J&D&A\\\hline A&M&B&J&D&C\\\hline A&M&B&J&D&M\\\hline A&M&B&J&J&A\\\hline A&M&B&J&J&C\\\hline A&M&B&J&J&M\\\hline\end{array}$$

All three individuals that were assigned to index four after processing the first sentence are still assigned to index four after processing ‘*they*$_4$ *respect some woman.*” This means the the context is passed on.