Context Semantics for NL

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

Second Draft — January 2001

Abstract

This paper sketches a dynamic incremental semantics for NL in polymorphic type theory, with a couple of fragments implemented in the functional programming language Haskell. We start out with a barebones framework, and next extend this to a set-up that can handle salience and pronoun resolution in context. The barebones framework can be viewed as the first reconstruction of Discourse Representation Theory in type theory that does justice to the incrementality and the finite domain semantics of the original. The second framework demonstrates how a slight extension of the context notion is enough for an implementation of a simple but powerful mechanism for pronoun reference resolution. The paper contains the complete Haskell code of the implementation.

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

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

1 Point of Departure: Incremental Dynamics

Destructive assignment is the main weakness of Dynamic Predicate Logic (DPL, [16], but see also [1]) as a basis for a compositional semantics of natural language: in DPL, the semantic effect of a quantifier action $\exists x$ is that the previous value of x gets lost forever. In this paper we replace DPL by an incremental logic for NL semantics — call it ID for Incremental Dynamics [9] — and build a type theoretic version of a compositional incremental semantics for NL that is without the destructive assignment flaw. ID can be viewed as the one-variable version of sequence semantics for dynamic predicate logic proposed in [36].

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 index information anymore.

1 *Some man loved some woman.*

The ID rendering of (1) is $\exists; Mi; \exists; Wi+1; Li(i+1)$, where i denotes the length of the input context. On the empty input context, this gets 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*$_1$.

Further on, we will specify a procedure for reference resolution of pronouns in a given context, but for now we will assume that pronouns carry index information.

2 Extension to Typed Logic

The Proper Treatment of Context (PTC) for NL developed in [10] in terms of polymorphic type theory (see, e.g., [21, 30]) 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 $\lambda P \lambda c \lambda c'.\exists x(man\ x \wedge P|c|(c\hat{}x)c')$.

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 [13]). 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 A First Toy Fragment

First we declare a module, and import the standard List module.

```
module CS where

import List
```

To give an incremental version of the toy fragment from [14] and [13], we define the appropriate dynamic operations in typed logic. Assume φ and ψ have the type of context transitions, i.e., type $[e] \to [e] \to t$, and that c, c', c'' have type $[e]$. Note that $\hat{}$ is an operation of type $[e] \to e \to [e]$.

$$\begin{aligned} \exists &:= \lambda cc'.\exists x(c\hat{}x = c') \\ \neg\neg\varphi &:= \lambda cc'.(c = c' \wedge \neg\exists c''\varphi cc'') \\ \varphi\ ;\ \psi &:= \lambda cc'.\exists c''(\varphi cc'' \wedge \psi c''c') \end{aligned}$$

These operations encode the semantics for incremental quantification, dynamic incremental negation and dynamic incremental conjunction in typed logic.

Again, we also define an operation $\Downarrow$: $([e] \to t) \to t$ to indicate that the context set $[e] \to t$ is not empty. Thus, $\Downarrow$ serves as an indication of success. Assume p to be an expression of type $[e] \to t$, the definition of $\Downarrow$ is:

$$\Downarrow p \quad := \quad \exists c.(pc).$$

We have to assume that the lexical meanings of CNs, VPs are given as one place predicates (type $e \to t$) and those of TVs as two place predicates (type $e \to e \to t$). We therefore define blow-up operations for lifting one-placed and two-placed predicates to the dynamic level. Assume A to be an expression of type $e \to t$, and B an expression of type $e \to e \to t$; we use c, c' as variables of type $[e]$, and j, j' as variables of type ι, and we employ postfix notation for the lifting operations:

$$\begin{aligned} A^{\circ} &:= \lambda jcc'.(c = c' \wedge Ac[j]) \\ B^{\bullet} &:= \lambda jj'cc'.(c = c' \wedge Bc[j]c[j']) \end{aligned}$$

The encodings of the ID operations in typed logic and the blow-up operations for one- and two-placed predicates are employed in the semantic specification of the fragment. The semantic specifications employ variables P, Q of type $\iota \to [e] \to [e] \to t$, variables j, j' of type ι, and variables c, c' of type $[e]$.

We will assume that pronouns are the only NPs that carry indices. Appropriate indices for proper names are now extracted from the current context. The topic of pronoun reference resolution in context will be taken up below, in our second fragment.

S	::=	**NP VP**	X	::=	$(X_1 X_2)$
S	::=	*if* **S S**	X	::=	$X_2 \Rightarrow X_3$
S	::=	**S . S**	X	::=	$X_1 \,;\, X_3$
NP	::=	*Mary*	X	::=	$\lambda P c c'.\exists j(c[j] = m \wedge P j c c')$
NP	::=	PRO^j	X	::=	$\lambda P c c'.(P j c c')$
NP	::=	**DET CN**	X	::=	$(X_1 X_2)$
NP	::=	**DET RCN**	X	::=	$(X_1 X_2)$
DET	::=	*every*	X	::=	$\lambda P Q c.(\neg\!\!\neg(\exists \,;\, P\|c\| \,;\, \neg\!\!\neg Q\|c\|))c$
DET	::=	*some*	X	::=	$\lambda P Q c.(\exists \,;\, P\|c\| \,;\, Q\|c\|)c$
DET	::=	*no*	X	::=	$\lambda P Q c.(\neg\!\!\neg(\exists \,;\, P\|c\| \,;\, Q\|c\|))c$
DET	::=	*the*	X	::=	$\lambda P Q c.$ $((\lambda c'.c = c' \wedge \exists x \forall y(\Downarrow(\exists \,;\, P\|c\| \,;\, (\|c\|\|y)c) \leftrightarrow x = y))$ $;\exists \,;\, P\|c\| \,;\, Q\|c\|)c$
CN	::=	*man*	X	::=	M°
CN	::=	*woman*	X	::=	W°
CN	::=	*boy*	X	::=	B°
RCN	::=	**CN** *that* **VP**	X	::=	$\lambda j.((X_1\ j) \,;\, (X_3\ j))$
RCN	::=	**CN** *that* **NP TV**	X	::=	$\lambda j.((X_1\ j) \,;\, (X_3(\lambda j'.((X_4\ j')j))))$
VP	::=	*laughed*	X	::=	L°
VP	::=	*smiled*	X	::=	S°
VP	::=	**TV NP**	X	::=	$\lambda j.(X_2 \,;\, \lambda j'.((X_1\ j')j))$
TV	::=	*loved*	X	::=	$L'^\bullet$
TV	::=	*respected*	X	::=	$R^\bullet$

The main difference with the fragments in [14] and [13] is that determiners do not carry indices anymore. The appropriate index is provided by the length of the input context. As in [13], it is assumed that all proper names are linked to anchored elements in context. In fact, the anchoring mechanism has been greatly improved by the switch from DPL-style to ID-style dynamics, for the incrementality of the context update mechanism ensures that no anchored elements can ever be overwritten.

In the implementation below we also throw in reflexive pronouns; the formulation of the abstract interpretation rule for those is left as an exercise for the reader.

4 Implementation — Basic Types

For our Haskell [23] implementation, 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,Enum,Show)

type Context = [Entity]
type Prop = [Context]
type Trans = Context -> Prop
type Idx = Int
```

5 Index Lookup and Context Extension

`lookupIdx` is the implementation of $c[i]$. `extend` is the implementation of $c\hat{}x$. `extend` replaces the destructive `update` from [13].

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

extend :: Context -> Entity -> Context
extend = \ c e -> c ++ [e]
```

6 Dynamic Negation, Conjunction, Implication, Quantification

```
neg :: Trans -> Trans
neg = \ phi c -> if phi c == [] then [c] else []

conj :: Trans -> Trans -> Trans
conj = \ phi psi c -> concat [ psi c' | c' <- (phi c) ]

impl :: Trans -> Trans -> Trans
impl = \ phi psi ->  neg (phi `conj` (neg psi))

exists :: Trans
exists = \ c -> [ (extend c x) | x <- [minBound..maxBound]]

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

7 Anchors for Proper Names

The anchors for proper names are extracted from an initial context.

```
context :: Context
context = [A,M,B,J]

anchor :: Entity -> Context -> Idx
anchor = \ e c -> anchor' e c 0 where
  anchor' e [] i = error (show e ++ " not anchored in context")
  anchor' e (x:xs) i | e == x    = i
                     | otherwise = anchor' e xs (i+1)
```

8 Syntax of the First Fragment

No index information on NPs, except for pronouns. Otherwise, virtually the same as the datatype declaration in [13]. The NPs `He`, `She`, `It` will be used in our second fragment.

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

data NP = Ann | Mary | Bill | Johnny
        | PRO Idx | He | She | It
        | NP1 DET CN | NP2 DET RCN
     deriving (Eq,Show)

data DET = Every | Some | No | The
     deriving (Eq,Show)

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

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

data VP = Laughed | Smiled | VP1 TV NP | VP2 TV REFL
     deriving (Eq,Show)

data REFL = Self deriving (Eq,Show)

data TV = Loved | Respected | Hated | Owned
     deriving (Eq,Show)
```

9 Model Information

Our model has named entities, one-placed predicates and two-placed predicates. Names:

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

One-placed predicates:

```
man,woman,boy,person,thing,house,cat,mouse,laugh,smile :: 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
smile x = x == M || x == B || x == I || x == K
```

Two-placed predicates:

```
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)
```

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

```
blowupPred :: (Entity -> Bool) -> Idx -> Trans
blowupPred = \ pred i c ->
               if pred (lookupIdx c i) then [c] else []
```

Discourse blow-up for two-placed predicates.

```
blowupPred2 :: (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans
blowupPred2 = \ pred i1 i2 c ->
      if pred (lookupIdx c i1) (lookupIdx c i2) then [c] else []
```

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 -> Trans` rather than `Entity -> Entity -> Bool`.

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

11 Dynamic Interpretation

The interpretation of sentences, in type `S -> Trans`:

```
intS :: S -> Trans
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 and pronouns.

```
intNP :: NP -> (Idx -> Trans) -> Trans
intNP Mary = \ p c -> p (anchor mary c) c
intNP Ann = \ p c -> p (anchor ann c) c
intNP Bill = \ p c -> p (anchor bill c) c
intNP Johnny = \ p c -> p (anchor johnny c) c
intNP (PRO i) = \ p -> p i
```

Interpretation of complex NPs as expected:

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

Interpretation of `(VP1 TV NP)` as expected. Interpretation of `(VP2 TV REFL)` uses the relation reducer `self`. Interpretation of lexical VPs uses discourse blow-up from the lexical meanings.

```
intVP :: VP -> Idx -> Trans
intVP (VP1 tv np) = \ subj -> intNP np (\ obj -> intTV tv obj subj)
intVP (VP2 tv _) = self (intTV tv)
intVP Laughed = blowupPred laugh
intVP Smiled = blowupPred smile
```

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

```
intTV :: TV -> Idx -> Idx -> Trans
intTV Loved = blowupPred2 love
intTV Respected = blowupPred2 respect
intTV Hated = blowupPred2 hate
intTV Owned = blowupPred2 own
```

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

```
intCN :: CN -> Idx -> Trans
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
```

Code for checking that a discourse predicate is unique.

```
singleton :: [a] -> Bool
singleton [x] = True
singleton  _  = False

unique :: Idx -> Trans -> Trans
unique i phi c | singleton xs = [c]
               | otherwise    = []
    where xs = [ x | x <- [minBound..maxBound], phi (extend c x) /= []]
```

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

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

Interpretation of determiners in terms of dynamic quantification `exists`, dynamic negation `neg`, dynamic conjunction `conj`, and dynamic uniqueness check `unique`. The difference with the treatment in [13] is that the indices are now derived from the input context.

```
intDET Some = \ phi psi c -> let i = length c in
                (exists `conj` (phi i) `conj` (psi i)) c
intDET Every = \ phi psi c -> let i = length c in
                neg (exists `conj` (phi i) `conj` (neg (psi i))) c
intDET No = \ phi psi c -> let i = length c in
                neg (exists `conj` (phi i) `conj` (psi i)) c
intDET The = \ phi psi c -> let i = length c in
                ((unique i (phi i)) `conj`
                   exists `conj` (phi i) `conj` (psi i)) c
```

The interpretation of relativised common nouns is as expected:

```
intRCN :: RCN -> Idx -> Trans
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))
```

12 Testing It Out

The initial context from which evaluation can start is given by `context`.

```
eval :: S -> Prop
eval = \ s -> intS s context
```

Here is set of example sentences, for use as a test suite:

```
ex1 = S Johnny Smiled
ex2 = S Bill Laughed
ex3 = If (S Bill Laughed) (S Johnny Smiled)
ex4 = (S Bill Laughed) `Txt` (S Johnny Smiled)
ex5 = (S Bill Smiled) `Txt` (S (PRO 1) (VP1 Loved (NP1 Some Woman)))
ex6 = S (NP1 The Boy) (VP1 Loved (NP1 Some Woman))
ex7 = S (NP1 Some Man) (VP1 Loved (NP1 Some Woman))
ex8 = S (NP1 Some Man) (VP1 Respected (NP1 Some Woman))
ex9 = S (NP1 The Man) (VP1 Loved (NP1 Some Woman))
ex10 = S (NP1 Every Man) (VP1 Loved (NP1 Some Woman))
ex11 = S (NP1 Every Man) (VP1 Loved Johnny)
ex12 = S (NP1 Some Woman) (VP1 Loved Johnny)
ex13 = S Johnny (VP1 Loved (NP1 Some Woman))
ex14 = S Johnny (VP1 Respected (NP2 Some (CN1 Man (VP1 Loved Mary))))
ex15 = S (NP1 No Woman) (VP1 Loved Bill)
ex16 = S (NP2 No (CN1 Woman (VP1 Hated Johnny))) (VP1 Loved Bill)
ex17 = S (NP2 Some(CN1 Woman (VP1 Respected Johnny))) (VP1 Loved Bill)
ex18 = S (NP1 The Boy) (VP1 Loved Johnny)
ex19 = S (PRO 2) (VP1 Loved (PRO 1))
ex20 = S (PRO 2) (VP1 Respected (PRO 1))
ex21 = If (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
                  (S (PRO 4) (VP1 Respected (PRO 5)))
ex22 = Txt (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
                   (S (PRO 4) (VP1 Respected (PRO 5)))
ex23 = S (NP1 Some Woman) (VP1 Owned (NP1 Some Thing))
ex24 = S (NP1 Some Woman) (VP1 Owned (NP1 The House))
ex25 = Txt (S (NP1 Some Woman) (VP1 Owned (NP1 The House)))
           (S Johnny (VP1 Hated (PRO 4)))
ex26 = S (NP1 Every Man) (VP2 Respected Self)
ex27 = S (NP1 Some Man) (VP2 Respected Self)
```

13 Updating Salience Relations in a Context

Pronoun resolution should resolve pronouns to the most salient referent in context, modulo additional constraints such as gender agreement. To handle salience, we need contexts with slightly more structure, so that context elements can be permuted without danger of losing track of them. Contexts as lists of elements under a permutation are conveniently represented as lists of index/element pairs. To talk about the permutation **bcad** of the context **abcd** we represent **abcd** as $[(0,\mathbf{a}),(1,\mathbf{b}),(2,\mathbf{c}),(3,\mathbf{d})]$ and reshuffle this to $[(2,\mathbf{c}),(1,\mathbf{d}),(0,\mathbf{a}),(3,\mathbf{d})]$. Then we can continue to use index 2 to pick up the reference to **c**, no matter where **c** ends up in the reshuffled context, while the list ordering encodes salience of the items.

In an indexed context c, the entity with index i is given by $c[*i]$. E.g.,

$$[(2,\mathbf{c}),(1,\mathbf{d}),(0,\mathbf{a}),(3,\mathbf{d})][*0] = \mathbf{a}.$$

Before, a test on a context left the context unaffected when it succeeded. We now have to modify this to allow for salience reshuffles. For this, we reformulate the semantics for 'contexts under permutation' as follows.

If c is a context under permutation, let $(i)c$ be the result of placing the item $(i, c[*i])$ upfront. E.g., if $c = [(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a}),(3,\mathbf{d})]$, then $(3)c = [(3,\mathbf{d}),(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a})]$, i.e., it is the result of moving $(3,\mathbf{d})$ to the head position in the list. It is not hard to see that successive applications of this operation can generate all permutations of a context.

If $d \in D$, $d : c$ is the result of putting item $(|c|, d)$ at the head position of the context list. Thus, if $c = [(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a}),(3,\mathbf{d})]$, $\mathbf{e} : c = [(4,\mathbf{e}),(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a}),(3,\mathbf{d})]$. This operation is used for adding a new element to the context, in most salient position.

Finally, we need a way of cutting context down do size: $\lceil i \rceil\ c$ specifies the result of removing all items with an index $\geq i$ from the context c. E.g.,

$$\lceil 3 \rceil\ [(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a}),(3,\mathbf{d})] = [(2,\mathbf{c}),(1,\mathbf{b}),(0,\mathbf{a})].$$

Note that $\lceil i \rceil\ c$ produces a context of length i. Lift this operation to sets of contexts C by means of $\lceil i \rceil\ C := \{\lceil i \rceil\ c \mid c \in C\}$.

Assume i, j to be indices with $i, j < |c|$. Then $M \models_c Pi$, $M \Dashv_c Pi$, $M \models_c Pij$ and $M \Dashv_c Pij$ are given by:

$$M \models_c Pi \;\Leftrightarrow\; c[*i] \in I(P), \qquad M \Dashv_c Pi \;\Leftrightarrow\; c[*i] \notin I(P).$$

$$M \models_c Pij \;\Leftrightarrow\; \langle c[*i], c[*j]\rangle \in I(P), \qquad M \Dashv_c Pij \;\Leftrightarrow\; \langle c[*i], c[*j]\rangle \notin I(P).$$

We specify the positive and negative relational meaning of the basic context logic (in a similar style to [5]) as follows:

$$
\begin{array}{rcl}
[\![\exists]\!]^+(c) & := & \{d : c \mid d \in D\} \\
[\![\exists]\!]^-(c) & := & \emptyset \\
[\![Pi]\!]^+(c) & := & \begin{cases} \{(i)c\} & \text{if } M \models_c Pi \\ \emptyset & \text{if } M \not\models_c Pi \end{cases} \\
[\![Pi]\!]^-(c) & := & \begin{cases} \{(i)c\} & \text{if } M \not\models_c Pi \\ \emptyset & \text{if } M \models_c Pi \end{cases} \\
[\![Pij]\!]^+(c) & := & \begin{cases} \{(i)(j)c\} & \text{if } M \models_c Pij \\ \emptyset & \text{if } M \not\models_c Pij \end{cases} \\
[\![Pij]\!]^-(c) & := & \begin{cases} \{(i)(j)c\} & \text{if } M \not\models_c Pij \\ \emptyset & \text{if } M \models_c Pij \end{cases} \\
[\![\neg\varphi]\!]^+(c) & := & [\![\varphi]\!]^-(c) \\
[\![\neg\varphi]\!]^-(c) & := & \lceil |c| \rceil\ [\![\varphi]\!]^+(c) \\
[\![\varphi;\psi]\!]^+(c) & := & \bigcup\{[\![\psi]\!]^+(c') \mid c' \in [\![\varphi]\!]^+(c)\} \\
[\![\varphi;\psi]\!]^-(c) & := & \begin{cases} \emptyset & \text{if } \exists c' \in [\![\varphi]\!]^+(c) \text{ with } [\![\psi]\!]^+(c') \neq \emptyset \\ [\![\varphi]\!]^-(c) & \text{if } [\![\varphi]\!]^+(c) = \emptyset \\ \lceil |c| \rceil \bigcup\{[\![\psi]\!]^-(c') \mid c' \in [\![\varphi]\!]^+(c)\} & \text{otherwise} \end{cases}
\end{array}
$$

The notion of a test on context now gets refined, as follows. Let $c \sim c' :\Leftrightarrow$ c is a permutation of c'. Then a test-new-style on input context c is a formula φ with the property that for any model it holds that all $c' \in [\![\varphi]\!]^+(c)$ satisfy $c \sim c'$, and all $c' \in [\![\varphi]\!]^-(c)$ satisfy $c \sim c'$.

The following proposition asserts that negations and predications are tests-new-style. The proof is immediate from the definitions.

Proposition 1 *For all φ in the language, for all contexts c:*

- *if $c' \in [\![Pi]\!]^+(c)$ then $c \sim c'$, if $c' \in [\![Pi]\!]^-(c)$ then $c \sim c'$,*
- *if $c' \in [\![Pij]\!]^+(c)$ then $c \sim c'$, if $c' \in [\![Pij]\!]^-(c)$ then $c \sim c'$,*
- *if $c' \in [\![\neg\varphi]\!]^+(c)$ then $c \sim c'$, if $c' \in [\![\neg\varphi]\!]^-(c)$ then $c \sim c'$.*

Note that double negation does change the interpretation of φ into a test modulo permutation, for we have:

$$[\![\neg\neg\varphi]\!]^+(c) = [\![\neg\varphi]\!]^-(c) = \lceil |c| \rceil [\![\varphi]\!]^+(c).$$

The next proposition reassures us that the associativity property of sequential composition still holds.

Proposition 2 *For all φ, ψ, χ in the language, for all contexts c:*

$$[\![\varphi;(\psi;\chi)]\!]^+(c) = [\![(\varphi;\psi);\chi]\!]^+(c),$$

and

$$[\![\varphi;(\psi;\chi)]\!]^-(c) = [\![(\varphi;\psi);\chi]\!]^-(c).$$

Proof. The case of $[\![\varphi;(\psi;\chi)]\!]^+(c) = [\![(\varphi;\psi);\chi]\!]^+(c)$ is straightforward. The reasoning for $[\![\varphi;(\psi;\chi)]\!]^-(c) = [\![(\varphi;\psi);\chi]\!]^-(c)$ is as follows.

First assume $\exists c' \in [\![\varphi]\!]^+(c)$ with $[\![\psi;\chi]\!]^+(c') \neq \emptyset$. This is equivalent to $\exists c' \in [\![\varphi;\psi]\!]^+(c)$ with $[\![\chi]\!]^+(c') \neq \emptyset$. Therefore, in this case $[\![\varphi;(\psi;\chi)]\!]^-(c) = [\![(\varphi;\psi);\chi]\!]^-(c) = \emptyset$.

Next assume $[\![\varphi]\!]^+(c) = \emptyset$. Then by the definition of $[\![\cdot]\!]^-$ we have that $[\![\varphi;(\psi;\chi)]\!]^-(c) = [\![\varphi]\!]^-(c)$. From $[\![\varphi]\!]^+(c) = \emptyset$ we have that $[\![\varphi;\psi]\!]^+(c) = \emptyset$. Therefore $[\![(\varphi;\psi);\chi]\!]^-(c) = [\![\varphi;\psi]\!]^-(c)$. Again by $[\![\varphi]\!]^+(c) = \emptyset$ we get that $[\![\varphi;\psi]\!]^-(c) = [\![\varphi]\!]^-(c)$. Therefore, in this case, $[\![\varphi;(\psi;\chi)]\!]^-(c) = [\![(\varphi;\psi);\chi]\!]^-(c) = [\![\varphi]\!]^-(c)$.

Finally assume $[\![\varphi]\!]^+(c) \neq \emptyset$, and for all $c' \in [\![\varphi]\!]^+(c)$ it holds that $[\![\psi;\chi]\!]^+(c') = \emptyset$. Two cases: (i) $[\![\varphi;\psi]\!]^+(c) = \emptyset$ or (ii) $[\![\varphi;\psi]\!]^+(c) \neq \emptyset$.

In case (i),

$$\begin{aligned}
[\![\varphi;(\psi;\chi)]\!]^-(c) &= \lceil |c| \rceil \bigcup \{[\![\psi;\chi]\!]^-(c') \mid c' \in [\![\varphi]\!]^+(c)\} \\
&= \lceil |c| \rceil \bigcup \{[\![\psi]\!]^-(c') \mid c' \in [\![\varphi]\!]^+(c)\} \\
&= [\![\varphi;\psi]\!]^-(c) \\
&= [\![(\varphi;\psi);\chi]\!]^-(c).
\end{aligned}$$

In case (ii), we get:

$$\begin{aligned}
[\![\varphi;(\psi;\chi)]\!]^-(c) &= \lceil |c| \rceil \bigcup \{[\![\psi;\chi]\!]^-(c') \mid c' \in [\![\varphi]\!]^+(c)\} \\
&= \lceil |c| \rceil \bigcup \{[\![\chi]\!]^-(c'') \mid c'' \in \bigcup \{[\![\psi]\!]^+(c') \mid c' \in [\![\varphi]\!]^+(c)\}\} \\
&= \lceil |c| \rceil \bigcup \{[\![\chi]\!]^-(c') \mid c' \in [\![\varphi;\psi]\!]^+(c)\} \\
&= [\![(\varphi;\psi);\chi]\!]^-(c).
\end{aligned}$$

□

Here is an example to further illustrate the definitions of $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$. Suppose we have a context with a reference to Mary, say, an item $(3, \mathbf{m})$ in a context of size 5. Let us assume the context, with its salience ordering, looks like this:

$$[(4, \mathbf{b}), (2, \mathbf{a}), (1, \mathbf{c}), (3, \mathbf{m}), (0, \mathbf{d})]$$

Then *No woman loves Mary* will have translation $\neg(\exists; W5; L5\ 3)$ in this context. Suppose we are interpreting this translation in a model where the sentence is true. Then $\exists; W5; L5\ 3$ should turn out false. For that, every update of the context with an item $(5, w)$, where w is some woman in the model should render $L5\ 3$ false. The processing of this falsity check will have as a result that the context gets reshuffled to

$$[(5, w), (3, \mathbf{m}), (4, \mathbf{b}), (2, \mathbf{a}), (1, \mathbf{c}), (0, \mathbf{d})]$$

Finally, when this is cut down to the original size 5 of the input context, we get:

$$\lceil 5 \rceil\ [(5, w), (3, \mathbf{m}), (4, \mathbf{b}), (2, \mathbf{a}), (1, \mathbf{c}), (0, \mathbf{d})] = [(3, \mathbf{m}), (4, \mathbf{b}), (2, \mathbf{a}), (1, \mathbf{c}), (0, \mathbf{d})].$$

The result is a salience update of the input context, with $(3, \mathbf{m})$ moved to the salient position.

14 Modifications of Basic Type Declarations

We turn to the implementation. As we are going to do pronoun resolution, we need more informative contexts, for the context should reveal which pronoun got resolved to which context element. For this, we define a language of constraints, as follows:

```
data Constraint = C1 VP Idx | C2 TV Idx Idx
                deriving Eq

instance Show Constraint where
  show (C1 vp i) = show vp ++ (' ':show i)
  show (C2 tv i j) = show tv ++ (' ':show i) ++ (' ':show j)
```

Examples of constraints are `C1 Laughed 3` and `C2 Hated 4 5`. These are displayed on the screen as `Laughed 3` and `Hated 4 5`, respectively.

To keep track of the indices in a constraint we define a function that retrieves its largest index.

```
maxIndex  :: Constraint -> Idx
maxIndex (C1 vp i)   = i
maxIndex (C2 tv i j) = max i j
```

To keep track of the modifications, we use `Context'` for the type of contexts-new-style, and similarly for `Prop'` and `Trans'`. Contexts-new-style have constraint lists built into them.

```
type Context' = ([(Idx,Entity)],[Constraint])
type Prop' = [Context']
type Trans' = Context' -> Bool -> Prop'
```

The new datatype for transitions allows for the specification of positive and negative transitions: if `phi :: Trans'` and `c :: Context'`, then `phi c True` specifies the positive transition for c, and `phi c False` specifies the negative transition for c.

For the new kind of context we need a new function to retrieve its size:

```
size :: Context' -> Int
size (c,co) = length c
```

15 New Utility Functions

`lookupIdx'` looks up at an index to retrieve an entity; `lookupIdx' c i` is the implementation of $c[*i]$.

```
lookupIdx' :: Context' -> Idx -> Entity
lookupIdx' ([],co)   j = error "undefined context element"
lookupIdx' ((i,x):xs,co) j | i == j    = x
                           | otherwise = lookupIdx' (xs,co) j
```

`adjust` adjusts the context by putting a discourse item up front, thus permuting the salience ordering.

```
adjust :: (Idx,Entity) -> Context' -> Context'
adjust (i,x) (c,co) | elem (i,x) c = (((i,x):(filter (/=(i,x)) c)),co)
                    | otherwise    = error "item not found in context"
```

New version of `extend`:

```
extend' :: Context' -> Entity -> Context'
extend' = \ (c,co) e -> let i = length c in (((i,e):c),co)
```

`success` checks if transition is possible from context.

```
success :: Context' -> Trans' -> Bool
success = \ c phi -> phi c True /= []
```

`cutoff cs i` cuts off all context elements in the list of contexts `cs` with index $\geq i$. This is the implementation of $\lceil i \rceil\ C$, the operation used to cut all contexts in C down to size i. The constraints that employ indices $\geq i$ are also removed, of course.

```
cutoff :: [Context'] -> Idx -> [Context']
cutoff [] i = []
cutoff ((c,co):cs) i = (cutoffc c i, cutoffco co i):(cutoff cs i)
  where
  cutoffc [] i = []
  cutoffc ((j,x):xs) i | j >= i    = cutoffc xs i
                       | otherwise = (j,x):(cutoffc xs i)
  cutoffco [] i = []
  cutoffco (co:cos) i | maxIndex co >= i = cutoffco cos i
                      | otherwise        = co:(cutoffco cos i)
```

The result of a cutoff may be that we end up with multiple copies of the same context. To remedy this, we need a function for removing superfluous copies: `nub`, from the standard `List` module.

16 New Versions of the Dynamic Operations

The `neg'` of a transition-new-style is got by swapping the truth value, and doing a cutoff for the negative transition relation.

```
neg' :: Trans' -> Trans'
neg' = \ phi c b -> if b then phi c False
                         else cutoff (phi c True) (size c)
```

If `conj'` succeeds then the result is as before, if `conj'` fails the salience order of the input gets adjusted by cutting a 'counterexample' context back to its original size.

```
conj' :: Trans' -> Trans' -> Trans'
conj' = \ phi psi c b ->
   if b then concat [ psi c' True | c' <- phi c True ]
        else if any (\c' -> psi c' True /= []) (phi c True)
          then []
        else if (phi c True) == []
          then (phi c False)
        else
          nub (cutoff
             (concat [ psi c' False  | c' <- phi c True ]) (size c))
```

`impl'` can now simply be defined in terms of `neg'` and `conj'`.

```
impl' ::  Trans' -> Trans' -> Trans'
impl' = \ phi psi ->  neg' (phi `conj'` (neg' psi))
```

`exists'` takes into account that the quantifier action always succeeds.

```
exists' :: Trans'
exists' = \ c b -> if b then [ (extend' c e) | e <- [minBound..maxBound]]
                        else []
```

17 Syntax, Lexical Semantics

Nothing new, except for the fact that pronouns do not carry indices anymore.

The procedure for predicate blowup is modified, for this is the spot where salience gets reset. Note that the salience relations of the context are adjusted even if the predicate does not hold.

```
blowupPred' :: (Entity -> Bool) -> Idx -> Trans'
blowupPred' = \ pred i c  b ->
   let
       e = lookupIdx' c i
       c' = adjust (i,e) c
   in
   if b then if pred e then [c'] else []
        else if pred e then [] else [c']
```

For blowup of VPs we add an extra touch to this, by including the relevant constraint in the context if the predicate holds.

```
blowupVP :: VP -> (Entity -> Bool) -> Idx -> Trans'
blowupVP = \ vp pred i c b ->
   let
       e = lookupIdx' c i
       (c',cos) = adjust (i,e) c
       co = C1 vp i
   in
   if b then if pred e then [(c',co:cos)] else []
        else if pred e then [] else [(c',cos)]
```

The new definition of predicate blowup for two-placed predicates ensures that subject is more salient than object.

```
blowupPred2' :: (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans'
blowupPred2'= \ pred object subject c b ->
   let
       e1 = lookupIdx' c object
       e2 = lookupIdx' c subject
       c' = adjust (subject,e2) (adjust (object,e1) c)
   in
   if b then if pred e1 e2 then [c'] else []
        else if pred e1 e2 then [] else [c']
```

To use this for blowup of TVs, we also need to add the relevant constraint to the context if the predicate holds.

```
blowupTV :: TV -> (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans'
blowupTV = \ tv pred object subject c b ->
   let
       e1 = lookupIdx' c object
       e2 = lookupIdx' c subject
       (c',cos) = adjust (subject,e2) (adjust (object,e1) c)
       co = C2 tv subject object
   in
   if b then if pred e1 e2 then [(c',co:cos)] else []
        else if pred e1 e2 then [] else [(c',cos)]
```

18 Reference Resolution

The following code implements a simple but powerful reference resolution mechanism. It just picks the indices of the entities satisfying the gender constraint from the current context, in order of salience.

```
resolveMASC :: Context' -> [Idx]
resolveMASC (c,co)  = resolveMASC' c where
  resolveMASC' []                     = []
  resolveMASC' ((i,x):xs) | man x     = i : resolveMASC' xs
                          | otherwise = resolveMASC' xs
resolveFEM :: Context' -> [Idx]
resolveFEM (c,co)  = resolveFEM' c where
  resolveFEM' []                      = []
  resolveFEM' ((i,x):xs) | woman x    = i : resolveFEM' xs
                         | otherwise = resolveFEM' xs
resolveNEUTR :: Context' -> [Idx]
resolveNEUTR (c,co)  = resolveNEUTR' c where
  resolveNEUTR' []                     = []
  resolveNEUTR' ((i,x):xs) | thing x   = i : resolveNEUTR' xs
                           | otherwise = resolveNEUTR' xs
```

Names are resolved by picking the most salient index to the named entity from the current context. If the name has no index in context, the context is extended with an index for the named object.

```
resolveNAME :: Entity -> Context' -> (Idx,Context')
resolveNAME x c | i /= -1   = (i,c)
                | otherwise = (j,extend' c x)
  where i = index x c
        j = size c
        index x ([],co) = -1
        index x ((i,y):xs,co) | x == y    = i
                              | otherwise = index x (xs,co)
```

As matters stand now, we will still get wrong results for examples like *he respected him*, where the well-known constraint should be imposed that *he* and *him* do not co-refer (see, e.g., Reinhart [35]). This constraint on coreference can be imposed with the following 'irreflexivizer':

```
nonCoref :: (Idx -> Idx -> Trans') -> Idx -> Idx -> Trans'
nonCoref = \ p i j c b -> if i /= j then (p i j c b) else []
```

When imposed on $[_{\mathrm{VP}}$TV NP] where NP is not a reflexive pronoun, this has the desired effect of blocking coreference.[1]

19 Dynamic Interpretation — New Version

Nothing really new, except for the fact that now we can do pronoun resolution in context. To ensure that the old test suite still runs, we also provide a rule for PRO^i.

```
intS' :: S -> Trans'
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)
```

[1] In [35], it is argued that the constraint on coreference is a pragmatic constraint, and that there are exceptions to this rule. Our implementation takes this into account, for the non-coreference constraint is implemented as a constraint on context, not on the underlying reality, where the relation that interprets the TV need not be irreflexive.

```
intNP' :: NP -> (Idx -> Trans') -> Trans'
intNP' Ann = \ p c -> let (i,c') = resolveNAME ann c in p i c'
intNP' Mary = \ p c -> let (i,c') = resolveNAME mary c in p i c'
intNP' Bill = \ p c -> let (i,c') = resolveNAME bill c in p i c'
intNP' Johnny = \ p c -> let (i,c') = resolveNAME johnny c in p i c'
intNP' He = \ p c b -> concat [p i c b | i <- resolveMASC c]
intNP' She = \ p c b -> concat [p i c b | i <- resolveFEM c]
intNP' It = \ p c b -> concat [p i c b | i <- resolveNEUTR c]
intNP' (PRO i) = \ p c ->  p i c
intNP' (NP1 det cn) = (intDET' det) (intCN' cn)
intNP' (NP2 det rcn) = (intDET' det) (intRCN' rcn)
```

In the interpretation rule for `(VP1 tv np)` we now impose the non-coreference constraint. For the interpretation of `(VP1 tv refl)`, we use the polymorphic `self`, this time as an operation of type `(Idx -> Idx -> Trans') -> Idx -> Trans'`.

```
intVP' :: VP -> Idx -> Trans'
intVP' (VP1 tv np) = \ subj ->
         intNP' np (\ obj -> nonCoref (intTV' tv) obj subj)
intVP' (VP2 tv refl) = self (intTV' tv)
intVP' Laughed = blowupVP Laughed laugh
intVP' Smiled = blowupVP Smiled smile
```

```
intTV' :: TV -> Idx -> Idx -> Trans'
intTV' Loved = blowupTV Loved love
intTV' Respected = blowupTV Respected respect
intTV' Hated = blowupTV Hated hate
intTV' Owned = blowupTV Owned own
```

```
intCN' :: CN -> Idx -> Trans'
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
```

Uniqueness check adjusted to new datastructures:

```
unique' :: Idx -> Trans' -> Trans'
unique' = \ i phi c b ->
   let
      xs = [ x | x <- [minBound..maxBound], success (extend' c x) phi ]
   in
   if b then if singleton xs then [c] else []
        else if singleton xs then [] else [c]
```

```
intDET' :: DET -> (Idx -> Trans') -> (Idx -> Trans') -> Trans'
intDET' Some = \ phi psi c -> let i = size c in
                 (exists' `conj'` (phi i) `conj'` (psi i)) c
intDET' Every = \ phi psi c -> let i = size c in
                  ((exists' `conj'` (phi i)) `impl'` (psi i)) c
intDET' No = \ phi psi c -> let i = size c in
                  ((exists' `conj'` (phi i)) `impl'` (neg' (psi i))) c
intDET' The = \ phi psi c -> let i = size c in
                ((unique' i (phi i)) `conj'`
                  exists' `conj'` (phi i) `conj'` (psi i)) c
```

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

20 Initiatisation and Evaluation – New Style

Conversion from contexts-old-style to contexts-new-style.

```
convert :: Context -> Context'
convert c = (convert' c (length c - 1),[]) where
  convert' [] i = []
  convert' (x:xs) i = (i,x):(convert' xs (i-1))
```

Evaluation of sentences, for initial context, checking for truth:

```
eval' :: S -> Prop'
eval' s = intS' s (convert context) True
```

21 Examples

The position at the front of the context list is the most salient one. We give some variations on the earlier test examples that can be used to check pronoun resolution.

```
nex1 = S He (VP1 Loved (NP1 Some Woman))
nex2 = S He (VP1 Hated (NP1 Some Thing))
nex3 = (S Bill Smiled) `Txt` (S He (VP1 Loved (NP1 Some Woman)))
nex4 = (S Bill Smiled) `Txt` (S He (VP1 Hated (NP1 Some Thing)))
nex5 = S (NP1 The Cat) (VP1 Hated (NP1 The Mouse))
nex6 = S (NP1 The Mouse) (VP1 Hated (NP1 The Cat))
nex7 = (S (NP1 The Mouse) (VP1 Hated (NP1 The Cat))) `Txt` (S It Smiled)
nex8 = (S (NP1 The Mouse) (VP1 Respected (NP1 The Cat)))
       `Txt` (S It (VP1 Hated It))
nex9 = S He (VP1 Loved He)
nex10 = S He (VP2 Respected Self)
nex11 = S He (VP1 Respected He)
nex12 = S (NP1 The Mouse) (VP2 Respected Self)
nex13 = (S (NP1 The Mouse) (VP2 Respected Self))
         `Txt` (S It (VP1 Hated (NP1 The Cat)))
nex14 = (S (NP1 Some Man) (VP2 Respected Self))
        `Txt` (S (NP1 Some Woman) (VP1 Loved He))
nex15 = If (S (NP1 Some Man) (VP2 Respected Self))
           (S (NP1 Some Woman) (VP1 Loved He))
nex16 = (S (NP1 No Woman) (VP1 Hated Johnny))
        `Txt` (S (NP1 Some Woman) (VP1 Loved He))
```

4 *He loved some woman.*

In a context where referents for the pronoun are available, *he* can be resolved to any referent that satisfies the property. And this is what we get. Let us first have a look at the inititial context.

```
CS> convert context
([(3,A),(2,M),(1,B),(0,J)],[])
CS>
```

This context contains two women and two men. The referents `(1,B)` and `(0,J)` in the context are referents for men.

```
CS> eval' nex1
[([(1,B),(4,A),(3,A),(2,M),(0,J)],[Loved 1 4]),
([(1,B),(4,M),(3,A),(2,M),(0,J)],[Loved 1 4])]
CS>
```

As it turns out, in the model only `B` is a lover. Therefore *he* gets resolved to `B`, and the new contexts have `B` and `B`-s loved one as the two most salient items, with the subject more salient than the object.

5 *He hated some thing.*

Here we expect that we get all the new contexts where B or J with their objects of hatred are added, again with the subjects more salient than the objects. And this is what we get:

```
CS> eval' nex2
[([(1,B),(4,E),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,F),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,G),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,H),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,I),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,K),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(1,B),(4,L),(3,A),(2,M),(0,J)],[Hated 1 4]),
([(0,J),(4,E),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,F),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,G),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,H),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,I),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,K),(3,A),(2,M),(1,B)],[Hated 0 4]),
([(0,J),(4,L),(3,A),(2,M),(1,B)],[Hated 0 4])]
CS>
```

6 *Bill smiled. He loved some woman.*

In the same initial context as before we get for example (6):

```
CS> eval' nex3
[([(1,B),(4,A),(3,A),(2,M),(0,J)],[Loved 1 4,Smiled 1]),
([(1,B),(4,M),(3,A),(2,M),(0,J)],[Loved 1 4,Smiled 1])]
CS>
```

He gets resolved to Bill, for as it happens, in the model Bill is the only man in love. Note that the example still works out with an empty initial context:

```
CS> intS' nex3 ([],[]) True
[([(0,B),(1,A)],[Loved 0 1,Smiled 0]),
([(0,B),(1,M)],[Loved 0 1,Smiled 0])]
CS>
```

The referent of *Bill* does not occur in the context, so it gets added.

7 *Bill smiled. He hated some thing.*

Evaluation in the empty context gives:

```
CS> intS' nex4 ([],[]) True
[([(0,B),(1,E)],[Hated 0 1,Smiled 0]),
([(0,B),(1,F)],[Hated 0 1,Smiled 0]),
([(0,B),(1,G)],[Hated 0 1,Smiled 0]),
([(0,B),(1,H)],[Hated 0 1,Smiled 0]),
([(0,B),(1,I)],[Hated 0 1,Smiled 0]),
([(0,B),(1,K)],[Hated 0 1,Smiled 0]),
([(0,B),(1,L)],[Hated 0 1,Smiled 0])]
CS>
```

Since Bill is the only referent available for resolution of the pronoun, we get Bill with his objects of hatred as new contexts. In a richer initial context, we may get more, of course:

```
CS> eval' nex4
[([(1,B),(4,E),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,F),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,G),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,H),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,I),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,K),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(1,B),(4,L),(3,A),(2,M),(0,J)],[Hated 1 4,Smiled 1]),
([(0,J),(4,E),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,F),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,G),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,H),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,I),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,K),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1]),
([(0,J),(4,L),(1,B),(3,A),(2,M)],[Hated 0 4,Smiled 1])]
CS>
```

8 *The mouse hated the cat. It smiled.*

We get that *it* can both be resolved to the mouse and to the cat, with a preference for the first resolution, as subject position is more salient.

```
CS> eval' nex7
[([(4,I),(5,F),(3,A),(2,M),(1,B),(0,J)],[Smiled 4,Hated 4 5]),
([(5,F),(4,I),(3,A),(2,M),(1,B),(0,J)],[Smiled 5,Hated 4 5])]
CS>
```

For examples like (8), Kameyama [24] has argued that the other order of resolution is more plausible, for relying on world knowledge (cartoon knowledge?) we can tell that a cat-hating mouse in sight of a cat is less likely to smile than a cat spotting a contemptuous mouse. But note that even that mechanism is taken into account by our little reference resolution engine. We simply take our model of the world as our yardstick for what is likely and what is not. In

the model under consideration the mouse *does* smile, and that is a piece of world knowledge that makes the reading with *it* resolved to the mouse plausible.[2]

9 *The mouse respected the cat. It hated it.*

This time, we only get a reading where the mouse hates the cat. This is because it so happens that in the background model the cat does not hate the mouse.[3]

```
CS> eval' nex8
[([(4,I),(5,K),(3,A),(2,M),(1,B),(0,J)],[Hated 4 5,Respected 4 5])]
CS>
```

10 *He respected him.*

Here the non-coreference constraint comes into play and forces the two pronouns to get resolved to different men in context.

```
CS>  eval' nex11
[([(1,B),(0,J),(3,A),(2,M)],[Respected 1 0]),
([(0,J),(1,B),(3,A),(2,M)],[Respected 0 1])]
CS>
```

11 *The mouse respected itself. It hated the cat.*

We get the right outcome, with *it* resolved to the mouse, and the mouse ending up in most salient position in the output context:

```
CS> eval' nex13
[([(4,I),(5,K),(3,A),(2,M),(1,B),(0,J)],[Hated 4 5,Respected 4 4])]
CS>
```

12 *Some man respected himself. Some woman loved him.*

Here things go slightly wrong:

```
CS> eval' nex14
[([(5,A),(4,B),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
([(5,A),(1,B),(4,B),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,A),(0,J),(4,B),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,C),(4,B),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
([(5,C),(1,B),(4,B),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,C),(0,J),(4,B),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,M),(4,B),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
```

[2] We may presume object I in the model to be Ignatz Mouse.

[3] The reason for this is that K in the model happens to be Krazy Kat.

```
([(5,M),(1,B),(4,B),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,M),(0,J),(4,B),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,A),(1,B),(4,D),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,A),(0,J),(4,D),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,C),(1,B),(4,D),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,C),(0,J),(4,D),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,M),(1,B),(4,D),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,M),(0,J),(4,D),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,A),(4,J),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
([(5,A),(1,B),(4,J),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,A),(0,J),(4,J),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,C),(4,J),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
([(5,C),(1,B),(4,J),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,C),(0,J),(4,J),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4]),
([(5,M),(4,J),(3,A),(2,M),(1,B),(0,J)],[Loved 5 4,Respected 4 4]),
([(5,M),(1,B),(4,J),(3,A),(2,M),(0,J)],[Loved 5 1,Respected 4 4]),
([(5,M),(0,J),(4,J),(3,A),(2,M),(1,B)],[Loved 5 0,Respected 4 4])]
CS>
```

What we see is that for each man who respects himself we get for every woman that the pronoun resolves to its possible referents in the right order of plausibility, with *the man who respects himself* as the most salient referent for the pronoun. This, however, is not the right overall order of plausibility. The problem is that choice of reference for indefinites is independent of salience order.

13 *If some man respected himself, some woman loved him.*

```
CS> eval' nex15
[([(3,A),(2,M),(1,B),(0,J)],[]),
([(1,B),(3,A),(2,M),(0,J)],[]),
([(0,J),(3,A),(2,M),(1,B)],[])]
CS>
```

In this example, we cannot quite see what has happened. The reason is that the conditional acts as a test. It adds an individual to the context that gets removed again at the end of processing.

14 *No woman hated Johnny. Some woman loved him.*

Again we get that the reference resolution is in the right order *per choice of woman*:

```
CS> eval' nex16
[([(4,A),(0,J),(3,A),(2,M),(1,B)],[Loved 4 0]),
([(4,A),(1,B),(0,J),(3,A),(2,M)],[Loved 4 1]),
([(4,C),(0,J),(3,A),(2,M),(1,B)],[Loved 4 0]),
([(4,C),(1,B),(0,J),(3,A),(2,M)],[Loved 4 1]),
```

```
([(4,M),(0,J),(3,A),(2,M),(1,B)],[Loved 4 0]),
([(4,M),(1,B),(0,J),(3,A),(2,M)],[Loved 4 1])]
CS>
```

22 Related Work

Rational Reconstructions of DRT based on DPL When dynamic semantics for NL first was proposed in [25] and [20], the approach invoked strong opposition from the followers of Montague [31]. Rational reconstructions to restore compositionality were announced in [16] and carried out in [15, 7, 22, 33, 34, 32, 8, 12, 28, 29]. All of these reconstructions are based in some way or other on DPL [16], 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 [25] and [26] 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.

New Rational Reconstruction of DRT 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.

Pronoun Resolution in Context Logic and in DRT The proposal for the treatment of salience in the second fragment is an extension of the basic context logic. It should be compared with the treatment of pronoun resolution in DRT proposed in the second volume of [6], as well as with the earlier proposal for pronoun resolution in DRT in [43].

Visser-Style Contexts Visser's context theories [39, 41, 40, 42] also start out as rational reconstructions of DRT. Visser's view of contexts is considerably more abstract than the simple-minded approach taken here.

The Centering Approach to Reference Resolution Central claim of the *centering theory* of local coherence in discourse [19, 18] is that pronouns are used to signal to the hearer that the speaker continues to talk about the same thing. See [44] for extensions and variations that take world knowledge into account, and [3] for a reformulation in terms of optimality theory. The reference resolution mechanism proposed above is meant as a demonstration 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.

Referent Systems Referent systems [37, 36, 38] are a mechanism for indirect reference to objects, via variable names and indices ('pegs'). Referent systems are employed in [17] to

reformulate the logic of DPL in an incremental fashion in order to solve certain puzzles of epistemic modality in dynamic semantics, and in [2] 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.

23 Future Work

Above, we have demonstrated pronoun reference resolution related to a fixed background model. More realistic is a set-up where the 'compatible' models grow incrementally with the discourse. This can be achieved by using a tableau unfolding mechanism for model generation (cf. [4, 27, 11]). We hope to describe and implement tableau based model generation for context logic in a future paper.

Acknowledgement Many thanks to Albert Visser for stimulus and response.

References

[1] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifiers: linguistic and logical approaches*, pages 1–30. Reidel, Dordrecht, 1987.

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

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

[4] J. van Benthem and J. van Eijck. The dynamics of interpretation. *Journal of Semantics*, 1(1):3–20, 1982.

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

[6] P. Blackburn and J. Bos. *Representation and Inference for Natural Language; A First Course in Computational Semantics — Two Volumes.* Internet, 1999. Electronically available from `http://www.coli.uni-sb.de/~bos/comsem/`.

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

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

[9] J. van Eijck. Incremental dynamics. Technical Report INS-R9811 & LP-1998-08, CWI & ILLC, 1998. Accepted for publication in JoLLI.

[10] 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.

[11] J. van Eijck, J. Heguiabehere, and Breanndán Ó Nualláin. Theorem proving and programming with dynamic first order logic. Technical Report INS-R0020, CWI, Amsterdam, October 2000.

[12] 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.

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

[14] Jan van Eijck. Tutorial on the composition of meaning. Lecture Note, Uil-OTS, November 2000.

[15] 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.

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

[17] 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.

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

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

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

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

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

[23] S Peyton Jones and J. Hughes (eds). Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

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

[25] 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.

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

[27] M. Kohlhase. Model generation for Discoure Representation Theory. In *ECAI Proceedings*, 2000. Available from `http://www.ags.uni-sb.de/~kohlhase/`.

[28] 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.

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

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

[31] 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.

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

[33] 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.

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

[35] T. Reinhart. *Anaphora and Semantic Interpretation.* Croom Helm, London, 1983.

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

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

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

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

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

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

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

[43] H. Wada and N. Asher. BUILDRS: An implementation of DR theory and LFG. In *11th International Conference on Computational Linguistics. Proceedings of Coling '86*, University of Bonn, 1986.

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