Computational Semantics, Type Theory, and Functional Programming

III — Context Semantics

Jan van Eijck

CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

LOLA7 Tutorial, Pecs

August 2002

Summary

- Incremental Dynamics
- Context and Context Extension
- DRT, Incremental Dynamics and Type Theory
- Incremental Montague Grammar
- Updating Salience Relations in a Context
- Pronoun Reference Resolution

Point of Departure: Incremental Dynamics

Destructive assignment is the main weakness of Dynamic Predicate Logic (DPL, [GS91], but see also [Bar87]) 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 lecture we replace DPL by an incremental logic for NL semantics — call it ID for Incremental Dynamics [Eij01] — and build a type theoretic version of a compositional incremental semantics for NL. This context semantics for NL is without the destructive assignment flaw.

ID can be viewed as the one-variable version of sequence semantics for dynamic predicate logic proposed in [Ver93].

Contexts and Context Extension

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’.

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.

Semantics of ID

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).

Quantification and Atomic Test

$$[\![\exists]\!](c) \;:=\; \{c\hat{\ }d \mid d \in D\}$$

$$[\![Pi_1 \cdots i_n]\!](c) \;:=\; \begin{cases} \uparrow & \text{if } \exists j(1 \leq j \leq n \text{ and } [\![i_j]\!]_c = \uparrow) \\ \{c\} & \text{if } M \models_c Pi_1 \cdots i_n \\ \emptyset & \text{if } M =\!\!\mid_c Pi_1 \cdots i_n \end{cases}$$

$$[\![i_1 \doteq i_2]\!](c) \;:=\; \begin{cases} \uparrow & \text{if } [\![i_1]\!]_c = \uparrow \text{ or } [\![i_1]\!]_c = \uparrow \\ \{c\} & \text{if } M \models_c i_1 \doteq i_2 \\ \emptyset & \text{if } M =\!\!\mid_c i_1 \doteq i_2 \end{cases}$$

Negation

$$[\![\neg\varphi]\!](c) \;:=\; \begin{cases} \uparrow & \text{if } [\![\varphi]\!](c) = \uparrow \\ \{c\} & \text{if } [\![\varphi]\!](c) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Sequential Composition

$$[\![\varphi;\psi]\!](c) \;:=\; \begin{cases} \uparrow & \text{if } [\![\varphi]\!](c) = \uparrow \\ & \text{or } \exists c' \in [\![\varphi]\!](c) \text{ with } [\![\psi]\!](c') = \uparrow \\ \bigcup\{[\![\psi]\!](c') \mid c' \in [\![\varphi]\!](c)\} & \text{otherwise.} \end{cases}$$

This definition hinges on the fact that all contexts in $[\![\varphi]\!](c)$ have the same length.

Implication and Universal Quantification

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 $\exists \Rightarrow \varphi$.

NPs Without Indices

One advantage of the use of contexts is that indefinite NPs do not have to carry index information anymore.

Some man loved some woman.

The ID rendering: $\exists; Mi; \exists; W(i+1); Li(i+1)$.

Here 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

He$_0$ *kissed her*$_1$

can now use this contextual discourse information to pick up the references.

Extension to Typed Logic

The polymorphic type of a context is $[e]$.

An index into a context $[c_0, \ldots, c_{n-1}]$ is a natural number

$$j \in \{0, \ldots, n-1\}.$$

Using the Von Neumann definition of natural numbers

$$n = \{0, \ldots, n-1\},$$

we can say that an index for a context c is a number j with $j \in |c|$.

We will use ι for the type of an index into a context.

This relies on meta-context to make clear what the current constraints on context and indexing into context are.

In types such as $\iota \longrightarrow [e]$, we will tacitly assume that the index fits the size of the context.

Thus, $\iota \longrightarrow [e]$ is really a type scheme rather than a type, although the type polymorphism remains hidden from view.

Since $\iota \longrightarrow [e]$ generalizes over the size of the context, it is shorthand for the types $0 \longrightarrow [e]_0$, $1 \longrightarrow [e]_1$, $2 \longrightarrow [e]_2$, and so on.

Indefinite Noun Phrases

The translation of an indefinite noun phrase *a man*:

$\lambda P \lambda c \lambda c'.\exists x(\text{man } x \wedge Pi(c\hat{\ }x)c')$ where $i = |c|$.

Here P is a variable of type $\iota \rightarrow [e] \rightarrow [e] \rightarrow t$, while c, c' are variables of type $[e]$ (variables ranging over contexts).

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.

The translation has type $(\iota \rightarrow [e] \rightarrow [e]) \rightarrow [e] \rightarrow [e] \rightarrow t$.

Note that if $i = |c|$ then $(c\hat{\ }x)[i] = x$.

The translation of 'a man' does not introduce an anaphoric index, as in DPL based dynamic semantics for NL (see previous lecture).

An anaphoric index i is picked up from the input context. Also, the context is not reset but incremented: context update is not destructive.

Module Declaration

First we declare a module, and import the standard List module, plus the modules with domain and model information:

```
module LOLA3 where

import List
import Domain
import Model
```

Contexts, Propositions, Transitions

For our Haskell 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.

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

Index Lookup and Context Extension

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

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

`extend` is the implementation of $c\hat{}x$. `extend` replaces the destructive update from the previous lecture.

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

To give an incremental version of the fragments from the previous lectures, we define the appropriate dynamic operations in typed logic.

Assume φ and ψ have the type of context transitions, i.e., type $[e] \rightarrow [e] \rightarrow t$, and that c, c', c'' have type $[e]$.

Note that $\hat{}$ is an operation of type $[e] \rightarrow e \rightarrow [e]$.

Incremental quantification

$$\boldsymbol{\exists} \;:=\; \lambda c c'.\exists x(c\hat{\ }x = c')$$

```
exists :: Trans
exists = \ c -> [ extend c x | x <- entities ]
```

Incremental negation

$$\neg\varphi \;:=\; \lambda cc'.(c = c' \wedge \neg\exists c''\varphi cc'')$$

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

Incremental conjunction

$$\varphi \mathbin{;} \psi \;:=\; \lambda c c'.\exists c''(\varphi c c'' \wedge \psi c'' c')$$

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

Incremental implication

$$\varphi \Rightarrow \psi \;:=\; \neg(\varphi\,;\,\neg\psi)$$

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

Universal Quantification

$$\forall\varphi \;:=\; \neg(\exists \,;\, \neg\varphi)$$

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

Predicate Lifting

We have to assume that the lexical meanings of CNs, VPs are given as one place predicates (type $e \longrightarrow t$) and those of TVs as two place predicates (type $e \longrightarrow e \longrightarrow t$). 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 $e \longrightarrow t$, and B an expression of type $e \longrightarrow e \longrightarrow t$; 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:

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

Discourse blow-up of one-placed predicates:

```
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 i2, lookupIdx c i1)
   then [c] else []
```

Anchors for Proper Names

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

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

Datatypes for Syntax

No index information on NPs, except for pronouns. Otherwise, virtually the same as the datatype declaration of the previous lecture.

```
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)
```

Arity Reduction

Interpretation of VPs consisting of a TV with a reflexive pronoun uses the relation reducer `self`.

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

Note the polymorphism of this definition. We will use the arity reducer on relations in type

`Idx -> Idx -> Trans`

rather than

`Entity -> Entity -> Bool`.

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.

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

unique :: Trans -> Trans
unique phi c | singleton xs = [c]
             | otherwise    = []
    where xs = [ x | x <- entities,
                     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 the previous lecture 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 (phi i)) `conj`
     exists `conj` (phi i) `conj` (psi i)) c
```

Interpretation of relativised common nouns 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))
```

Trying It Out

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

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

Evaluation takes place by interpreting a sentence or piece of text in a context:

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

```
LOLA3> eval (S Johnny Smiled)
[]

LOLA3> eval (S Bill Laughed)
[[A,M,B,J]]

LOLA3> eval (S (NP1 The Boy) Laughed)
[[A,M,B,J,J]]

LOLA3> eval (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
[[A,M,B,J,B,A],[A,M,B,J,B,M]]

LOLA3> eval (S (NP1 Some Woman) (VP1 Loved (NP1 Some Man)))
[[A,M,B,J,A,B],[A,M,B,J,A,J],[A,M,B,J,C,B],
 [A,M,B,J,C,J],[A,M,B,J,M,B],[A,M,B,J,M,J]]
```

```
ex1 =
 (S (NP1 Some Woman) (VP1 Loved (NP1 Some Man)))
 ‘Txt‘ (S (PRO 6) (VP1 Loved (PRO 5)))
ex2 =
 (S (NP1 Some Woman) (VP1 Loved (NP1 Some Man)))
 ‘Txt‘ (S (PRO 5) (VP1 Loved (PRO 4)))
```

```
LOLA3> eval ex1

Program error: undefined ctxt element

LOLA3> eval ex2
[[A,M,B,J,A,B],[A,M,B,J,M,B]]
```

```
ex3 =
  S Johnny (VP1 Respected
    (NP2 Some (CN1 Man (VP1 Loved (NP1 Some Woman)))))
```

```
LOLA3> eval ex3
[[A,M,B,J,B,A],[A,M,B,J,B,M]]
```

```
ex4 =
  Txt (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
      (S (PRO 4) (VP1 Respected (PRO 5)))
```

```
LOLA3> eval ex4
[[A,M,B,J,B,A],[A,M,B,J,B,M]]
```

```
ex5 = S (NP1 Every Man) (VP2 Respected Self)
ex6 = S (NP1 Some Man) (VP2 Respected Self)
```

```
LOLA3> eval ex5
[[A,M,B,J]]

LOLA3> eval ex6
[[A,M,B,J,B],[A,M,B,J,D],[A,M,B,J,J]]
```

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.

Details of this are worked out in my invited lecture . . .

References

[Bar87] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifiers: linguistic and logical approaches*, pages 1–30. Reidel, Dordrecht, 1987.

[Eij01] J. van Eijck. Incremental dynamics. *Journal of Logic, Language and Information*, 10:319–351, 2001.

[GS91] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[Rei83] T. Reinhart. *Anaphora and Semantic Interpretation*. Croom Helm, London, 1983.

[Ver93] C.F.M. Vermeulen. Sequence semantics for dynamic predicate logic. *Journal of Logic, Language, and Information*, 2:217–254, 1993.