

# Natural Logic for Natural Language

Jan van Eijck

March 31, 2005

## Abstract

We implement the extension of the logical consequence relation to a partial order  $\leq$  on arbitrary types built from  $e$  (entities) and  $t$  (Booleans) that was given in [1], and the definition of monotonicity preserving and monotonicity reversing functions in terms of  $\leq$ . Next, we present a new algorithm for polarity marking, and implement this for a particular fragment of syntax. Finally, we list the research agenda that these definitions and this algorithm suggest. The implementations use Haskell [8], and are given in ‘literate programming’ style [9].

## 1 Introduction

Explanations of aspects of the human reasoning faculty must be based on hypotheses about calculating mechanisms. Monotonicity calculi have been proposed time and again in the literature as candidates for such mechanisms, by philosophers [15, 5], logicians [1, 14], computer scientists [13], linguists [3], and most recently by cognitive scientists [6], with less or more explicit suggestions to use them as a basis for generating hypotheses about processing load in human reasoning. The catch phrases for this enterprise used to be ‘natural logic’ or ‘logic for natural language’, for the logic that was meant to provide an account of the way human reasoners actually reason.

Monotonicity calculi can be viewed as the logical mechanics of syllogistic theory [5, 4]. A monotonicity preserving function  $F$  can be represented as a kind of ‘mental model’ [7], as follows:

$$\frac{X \Longrightarrow X' \quad X \xrightarrow{F} Y \quad X' \xrightarrow{F} Y'}{Y \Longrightarrow Y'} F \uparrow$$

The mental model somehow represents the ‘transfer’ by  $F$  of the growth of  $X$  to the growth of  $Y$ , with details largely irrelevant. Indeed, the lack of formal detail of the publications in the mental models school seems to indicate that mental models are meant to provide a suggestive *metaphor* of cognitive processing rather than a formal mechanism. The metaphor suggests that when the mental picture of ‘uniform growth’ is put in reverse, processing load increases:

$$\frac{X \Longrightarrow X' \quad X \xrightarrow{F} Y \quad X' \xrightarrow{F} Y'}{Y \Leftarrow Y'} F \downarrow$$

This links the hypothesis of [6] that reversal of mononotonicity increases human processing load to the mental models metaphor. Interestingly, from a logical point of view the reversed monotonicity pattern is no more complex than the pattern of preserved monotonicity.

Still, a question remains. Monotonicity calculi can be specified in a fully precise manner, by presenting them as proof calculi, consisting of axioms and inference rules. Such calculi are meant to capture standard notions of logical consequence: they are not calculi of logical falsehoods. If they can be used to explain where human reasoners err, it should be in an indirect manner, by making clear what the added complexity of a particular task is in comparison with tasks where human reasoners tend not to err. This suggests that, given a suitably precise version of a monotonicity calculus, it should be possible to flesh out the mental models metaphor as a formally precise extension of the monotonicity calculus, a kind of add-on tool that allows us to classify reasoning tasks with respect to their claims on the human processing faculty.

We will start with the semantics of monotonicity from [1], given in terms of partial orders on arbitrary semantic domains (supposed to correspond to various syntactic categories). This effectively extends the notion of logical entailment from the level of sentences to that of verb phrases, quantifiers, noun phrases, adjectives, and so on. Just as we can say that ‘Gaia is smiling’ logically implies ‘Gaia is smiling or Gaia is crying’, we can say that ‘smiling’ logically involves ‘smiling or crying’, or that ‘dancing’ logically implies ‘moving’, but also that ‘at least three’ logically implies ‘at least two’, and so on.

In this paper we implement the definitions that accomplish this lift of the logical consequence relation from the level of sentences (expressions of type  $t$ ) to arbitrary levels of functions built from the types  $t$  and  $e$  (expressions that denote entities).

Next, we turn to the topic of syntactic polarity marking. We show that this can be done by means of a very simple single pass top-down algorithm that employs nothing but the encodings of the monotonicity behaviour of functional categories.

At the end of the paper we list the agenda for further investigation of syntactic calculi that have been proposed or can be proposed to capture the semantic notion of lifted logical consequence.

## 2 Module Declaration

```
module NLNL where

import List
```

## 3 Representing a Domain of Entities

To illustrate the type  $e$ , we construct a small example domain of entities consisting of individuals  $A, \dots$ , by declaring a datatype `Entity`.

Because `Entity` is a bounded and enumerable type, we can put *all* of its elements in a finite list. For purposes of efficient computation, it makes sense to restrict the number of entities. A convenient way to do this is by commenting out some of the entities (by means of `--`).

```
data Entity = A | B | C
--          | D | E | F | G
--          | H | I | J | K | L | M | N
--          | O | P | Q | R | S | T | U
--          | V | W | X | Y | Z
          deriving (Show, Eq, Bounded, Enum)
```

The stuff about `deriving (Show, Eq, Bounded, Enum)` is there to display entities (`Show`), to enable us to do equality tests on entities (`Eq`), to refer to *A* as the minimum element and *M* as the maximum element (`Bounded`), and to enumerate the elements (`Enum`).

```
entities :: [Entity]
entities = [minBound..maxBound]
```

## 4 The Class of Partial Orderings

Fixity declarations for `==>` and `<==`:

```
infix 1 ==>, <==
```

A partial ordering is a class with functions  $\leq$  (implemented as `==>`) and  $\geq$  (implemented as `<==`) defined on it.

```
class Eq a => Pord a where
  (==>) :: a -> a -> Bool
  (<==) :: a -> a -> Bool

  -- minimal complete definition: (==>)
  (<==) = flip (==>)
```

Partial ordering of entities:

```
instance Pord Entity where
  (==>) = (==)
```

Partial ordering of Booleans:

```
instance Pord Bool where
  x ==> y = x <= y
```

## 5 Extending the Class of Partial Orderings (and Other Classes) to Functional Types

Show functions on bounded and enumerated domains by showing their graphs:

```
instance (Show a, Bounded a, Enum a, Show b) => Show (a -> b) where
  show f = show [(x,f x) | x <- [minBound..maxBound] ]
```

Define the min and max bounds of functions in terms of the bounds of their codomains:

```
instance Bounded b => Bounded (a -> b) where
  minBound = \ x -> minBound
  maxBound = \ x -> maxBound
```

Define equality for functions in point-wise fashion:

```
instance (Bounded a, Enum a, Eq b) => Eq (a -> b) where
  f == g = all (\ (x,y) -> x == y) (zip fvalues gvalues)
  where fvalues = map f [minBound..maxBound]
        gvalues = map g [minBound..maxBound]
```

If we are looking at functions with a domain of size  $n$  and a codomain of size  $m$  then index  $k$  corresponds to the function given by

$$k = c_{n-1} \times m^{n-1} + \dots + c_1 \times m + c_0,$$

with  $0 \leq c_i < m$ . The  $c_i$  are the indices of the values of the function. (Think of the representation of  $k$  in base  $m$ .)

The following function creates the graph of a function from the `maxBound` of its domain, a `dummy` value in its co-domain and an index for the function. The `dummy` is necessary to be able to generate the co-domain (from the type of the `dummy`).

```
graph :: (Eq a, Bounded a, Enum a, Eq b, Bounded b, Enum b) =>
  a -> b -> Int -> [(a,b)]
graph z dummy n =
  if z == minBound then [(z,toEnum n)]
  else ((z,toEnum q)) : graph (pred z) dummy r
  where
    cod = if dummy == maxBound then [minBound ..dummy]
          else [minBound..dummy]++[succ dummy..maxBound]
    b   = length cod
    p   = fromEnum z
    q   = quot n (b^p)
    r   = rem  n (b^p)
```

Now we are ready to define the `toEnum` and `fromEnum` functions for functional types, necessary and sufficient to lift enumeration to functional types.

```

instance (Eq a, Bounded a, Enum a, Eq b, Bounded b, Enum b) => Enum (a -> b)
  where
    toEnum n x = case lookup x (graph maxBound minBound n) of
      Just y  -> y
      Nothing -> error "lookup failure"

    fromEnum f = encBase b ns
      where xs = [ f x | x <- [minBound..maxBound] ]
            ns = map fromEnum xs
            y  = f minBound -- dummy value to compute the codomain
            cod = if y == maxBound then [minBound..y]
                  else [minBound..y]++[succ y..maxBound]
            b  = length cod
            encBase :: Int -> [Int] -> Int
            encBase b [] = 0
            encBase b (x:xs) = (x * b^m) + (encBase b xs)
              where m = length xs

```

Finally, we can extend the partial ordering to functional types:

```

instance (Eq a, Bounded a, Enum a, Pord a, Eq b, Pord b) => Pord (a -> b)
  where
    f ==> g = all (\ (x,y) -> x ==> y) (zip fvalues gvalues)
      where fvalues = map f [minBound..maxBound]
            gvalues = map g [minBound..maxBound]

```

## 6 Monotonicity

```

data Direction = Up | Down | None deriving (Eq,Show)

```

Monotonicity definition for functions. If  $f$  preserves  $\leq$  then it is upward monotone (or: monotonicity preserving), if it maps  $\leq$  to  $\geq$  then it is downward monotone (or: monotonicity reversing), if neither of the above then it is not monotone.

```

mon :: (Enum a, Bounded a, Pord a, Pord b) => (a -> b) -> Direction
mon f | all (uncurry (==>)) fxs = Up
      | all (uncurry (<==)) fxs = Down
      | otherwise                = None
  where pairs = [(u,v) | u <- [minBound..maxBound],
                          v <- [minBound..maxBound] ]
        leqxs = filter (uncurry (==>)) pairs
        fxs    = map (\ (u,v) -> (f u, f v)) leqxs

```

## 7 Connectives and Quantifiers

Generalized conjunction and disjunction:

```

conj, disj :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
conj p q = \ x -> p x && q x
disj p q = \ x -> p x || q x

```

Restricted generalized quantifiers on entities:

```

every :: (Entity -> Bool) -> (Entity -> Bool) -> Bool
every p q = all q (filter p entities)

some :: (Entity -> Bool) -> (Entity -> Bool) -> Bool
some p q = any q (filter p entities)

no :: (Entity -> Bool) -> (Entity -> Bool) -> Bool
no p = not . some p

most :: (Entity -> Bool) -> (Entity -> Bool) -> Bool
most p q = psqs > psnqs
  where psqs = length (filter (conj p q) entities)
        psnqs = length (filter (conj p (not . q)) entities)

atleast :: Int -> (Entity -> Bool) -> (Entity -> Bool) -> Bool
atleast n p q = psqs >= n
  where psqs = length (filter (conj p q) entities)

atmost :: Int -> (Entity -> Bool) -> (Entity -> Bool) -> Bool
atmost n p q = psqs <= n
  where psqs = length (filter (conj p q) entities)

```

A predicate for thinghood:

```

thing, things :: (Entity -> Bool)
thing = \ x -> True
things = \ x -> True

```

Generalized quantifiers on entities:

```

everything :: (Entity -> Bool) -> Bool
everything = every thing

something :: (Entity -> Bool) -> Bool
something = some thing

nothing :: (Entity -> Bool) -> Bool
nothing = no thing

```

Proper names, lifted in Montague fashion [10]:

```
ann, bob :: (Entity -> Bool) -> Bool
ann = \ p -> p A
bob = \ p -> p B
```

Common nouns:

```
cn1, cn2 :: (Entity -> Bool)
cn1 = \ x -> elem x [A,B]
cn2 = \ x -> elem x [B,C]
```

## 8 Test Suite

```
test :: (Pord a, Pord b, Show b, Enum a, Bounded a, Show a) =>
      (a -> b) -> IO ()
test f = do putStrLn (show f)
           putStrLn (show (mon f))
```

```
test0 = test thing
test1 = test not
test2 = test everything
test3 = test something
test4 = test nothing
test5 = test ((atleast 2) things)
test6 = test ((atmost 2) thing)
test7 = test (most things)
test8 = test every
test9 = test some
test10 = test no
test11 = test (atleast 2)
test12 = test (atmost 2)
test13 = test most
```



```

(C,False)],False),([(A,True),(B,True),(C,False)],False),([(A,False),(B,False),
(C,True)],False),([(A,True),(B,False),(C,True)],False),([(A,False),(B,True),
(C,True)],True),([(A,True),(B,True),(C,True)],True)],([(A,True),(B,True),(C,True)],
[[[(A,False),(B,False),(C,False)],False),([(A,True),(B,False),(C,False)],False),
[(A,False),(B,True),(C,False)],False),([(A,True),(B,True),(C,False)],True),
[(A,False),(B,False),(C,True)],False),([(A,True),(B,False),(C,True)],True),
[(A,False),(B,True),(C,True)],True),([(A,True),(B,True),(C,True)],True)])]]
None

```

Further tests:

```

NLNL> atleast 1 ==> atleast 2
False
NLNL> atmost 1 ==> atmost 2
True
NLNL> atleast 2 ==> atleast 1
True
NLNL> atmost 1 ==> atmost 2
True
NLNL> atmost 2 ==> atmost 1
False
NLNL> most ==> some
True
NLNL> some ==> some
True
NLNL> some ==> most
False
NLNL> some ==> every
False
NLNL> every ==> some
False

```

## 9 Monotonicity of Simple Predicates

In principle a function could be monotonicity preserving and reversing at the same time. The following alternative monotonicity test takes this possibility into account:

```

mon2 :: (Enum a, Bounded a, Pord a, Pord b) => (a -> b) -> [Direction]
mon2 f = [ Up   | all (uncurry (==>)) fxs ]
        ++
        [ Down | all (uncurry (<==)) fxs ]
  where pairs = [(u,v) | u <- [minBound..maxBound],
                          v <- [minBound..maxBound] ]
        leqxs = filter (uncurry (==>)) pairs
        fxs   = map (\ (u,v) -> (f u, f v)) leqxs

```

Observe that simple predicates (the denotations of common nouns and intransitive verbs) are both upward and downward monotonic. This is thanks to the definition of  $\leq$  for entities. Here is a demo:

```

NLNL> mon2 thing
[Up,Down]
NLNL> mon2 cn1
[Up,Down]
NLNL> mon2 cn2
[Up,Down]

```

This means that semantically speaking, a simple predicate can occur in positive and in negative position. Whether a position is positive or negative is determined top-down by the monotonicity preserving or reversing behaviour of the functions that it depends on.

## 10 Datastructures for Syntax

Instead of using the semantics to compute the monotonicity behaviour of semantic functions, we can define a syntax for these functions instead, and let the monotonicity markings be computed by a set of construction rules. Such a monotonicity calculus is linked to a specific syntax, but the mononicity rules need not be taken as part of the syntax. They could be in a separate component, on a par with the components for syntax, morphology or logical form. This will be demonstrated below, for the example syntax defined by the following data structures:

```

data Sent = Sent NP VP
    deriving (Eq,Show)

data NP = Ann | Mary | Bill | Johnny | NP1 DET CN | NP2 DET RCN
    deriving (Eq,Show)

data DET = Every | Some | Any | No | The | Most | Atleast Int
    deriving (Eq,Show)

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

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

data REL = That deriving (Eq,Show)

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

data AUX = Did | Didnt
    deriving (Eq,Show)

data INF = Laugh | Smile | INF1 TINF NP
    deriving (Eq,Show)

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

data TINF = Love | Respect | Hate | Own
    deriving (Eq,Show)

```

## 11 Polarity Marking

Instead of defining logical form rules for syntactic structures, we will give polarity marking rules. A data structure for polarity markings:

```

data Marking = Plus | Minus | Npol deriving Eq

instance Show Marking where
  show Plus   = "+"
  show Minus  = "-"
  show Npol   = "0"

```

Existing polarity marking calculi are all based in one way or another on Sanchez' [14] bottom-up algorithm for polarity marking. We propose to replace this by a top-down polarity marking algorithm that boils down to the following:

**Root Marking** The main structure  $C$  to be marked has positive polarity.

**Component Marking** If a structure  $C$  has polarity marking  $m$ , then:

**Leaf Marking** If  $C$  is a leaf, then done.

**Composite Marking** If  $C$  consists of a function  $F$  and argument  $A$ , then  $F$  has polarity marking  $m$ , and  $A$  has polarity marking  $f(m)$  where  $f$  is the polarity marking map that encodes the monotonicity behaviour of  $F$ .

There are three important maps on polarity markings: preservation, reversal, and breaking of polarity. Polarity marking is fully determined by the polarity preserving and reversing properties of the semantic functions involved. Instead of explicitly giving the function, in a monotonicity calculus it is enough to give the mappings on polarity markings: preservation (the mapping `id`), reversal (the mapping `rv`), or breaking of monotonicity (the mapping `br`).

```

rv, br :: Marking -> Marking
rv Plus  = Minus
rv Minus = Plus
rv Npol  = Npol
br _     = Npol

```

Functional categories are the syntactic categories that translate into semantic functions and that take (the objects corresponding to) other categories as their arguments. In the present fragment, these are determiners, NPs, auxiliaries, TVs and TINFs. Every functional category is equipped with a list of marking transformers.

```

monDET :: DET -> [Marking -> Marking]
monDET Every = [rv, id]
monDET Some  = [id, id]
monDET No    = [rv, rv]
monDET Any   = [id, id]
monDET The   = [br, id]
monDET Most  = [br, id]
monDET (Atleast i) = [id, id]

monNP :: NP -> [Marking -> Marking]
monNP Ann   = [id]
monNP Mary  = [id]
monNP Bill  = [id]
monNP Johnny = [id]
monNP (NP1 det cn) = tail (monDET det)
monNP (NP2 det rcn) = tail (monDET det)

monAUX :: AUX -> [Marking -> Marking]
monAUX Did = [id]
monAUX Didnt = [rv]

monTV :: TV -> [Marking -> Marking]
monTV _ = [id]

monTINF :: TINF -> [Marking -> Marking]
monTINF _ = [id]

```

Syntactic markings are labelled trees. A data structure for labelled trees:

```

data Tree a = Leaf a | Node a (Tree a) (Tree a) deriving (Eq,Ord,Show)

```

Trees labelled with a marking are of type `Tree Marking`. For every category we define a mapping from markings to trees labelled with markings. Our approach is different from that of Sanchez [14] in that the markings are computed top-down and without intermediate stages. Sanchez' algorithm works bottom-up and has three stages: (i) marking argument positions in lexical entries, (ii) propagating the markings to other categories, and (iii) polarity determination of nodes  $C$  by counting the number of plusses and minuses on a path from the root to  $C$ . Our approach is different from [3] in that it avoids multiplication of syntactic categories for items that can occur in both positive and negative positions: the approach of [3] is constraint-based and bottom up, whereas ours is top-down and driven by the information that the root node has

positive polarity. Finally, our approach shows that monotonicity marking can do without the rather top-heavy machinery of multimodal categorial grammar employed in [2].

The marking of a sentence  $[_S \text{ NP VP }]$ : the NP is the function and the VP the argument, so NP inherits the marking from S, and the marking of VP is determined by the monotonicity behaviour of NP.

```
mS :: Sent -> Marking -> Tree Marking
mS (Sent np vp) m = Node m (mNP np m) (mVP vp m')
  where m' = head (monNP np) m
```

Marking an NP. If the NP is a proper name, then nothing needs to be done. if the form is  $[_{NP} \text{ DET CN }]$  or  $[_{NP} \text{ DET RCN }]$ , then DET is the function and CN the argument, so DET inherits the marking from NP, and (R)CN has its marking determined by the monotonicity behaviour of DET.

```
mNP :: NP -> Marking -> Tree Marking
mNP Ann m = (Leaf m)
mNP Mary m = (Leaf m)
mNP Bill m = (Leaf m)
mNP Johnny m = (Leaf m)
mNP Bill m = (Leaf m)
mNP (NP1 det cn) m = Node m (mDET det m) (mCN cn m')
  where m' = head (monDET det) m
mNP (NP2 det rcn) m = Node m (mDET det m) (mRCN rcn m')
  where m' = head (monDET det) m
```

Marking of DET or CN: these are leaf nodes, so nothing needs to be done.

```
mDET :: DET -> Marking -> Tree Marking
mDET _ m = Leaf m

mCN :: CN -> Marking -> Tree Marking
mCN _ m = Leaf m
```

In a structure of the form  $[_{RCN} \text{ CN REL VP }]$ , REL is the function and CN and VP are arguments. Since the monotonicity behaviour of REL is monotone increasing in both arguments, both CN and VP inherit the polarity marking from RCN. Similarly for structures of the form  $[_{RCN} \text{ CN REL NP TV }]$ .

```

mRCN :: RCN -> Marking -> Tree Marking
mRCN (CN1 cn rel vp) m = Node m (mCN cn m) (mVP vp m)
mRCN (CN2 cn rel np tv) m = Node m (mCN cn m) (mNP np m)

```

In a structure of the form [VP TV NP ], TV is the function and NP the argument. In a structure of the form [VP AUX INF ], AUX is the function and INF the argument.

```

mVP :: VP -> Marking -> Tree Marking
mVP Laughed m = Leaf m
mVP Smiled m = Leaf m
mVP (VP1 tv np) m = Node m (mTV tv m) (mNP np m')
  where m' = head (monTV tv) m
mVP (VP2 aux inf) m = Node m (mAUX aux m) (mINF inf m')
  where m' = head (monAUX aux) m

```

All TVs are lexical, so nothing needs to be done.

```

mTV :: TV -> Marking -> Tree Marking
mTV _ m = Leaf m

```

All AUXes are lexical, nothing needs to be done.

```

mAUX :: AUX -> Marking -> Tree Marking
mAUX _ m = Leaf m

```

Non-lexical INFs are of the form [INF TINF NP ]. Here TINF is the function and NP the argument.

```

mINF :: INF -> Marking -> Tree Marking
mINF Laugh m = Leaf m
mINF Smile m = Leaf m
mINF (INF1 tinf np) m = Node m (mTINF tinf m) (mNP np m')
  where m' = head (monTINF tinf) m

```

All TINFs are lexical:

```
mTINF _ m = Leaf m
```

## 12 Test Suite, Again

```
mtest1 = mS (Sent Mary (VP2 Didnt Laugh)) Plus
mtest2 = mS (Sent (NP2 Every (CN1 Man That Laughed)) Smiled) Plus
mtest3 = mS (Sent (NP2 No (CN1 Man That Laughed)) Smiled) Plus
mtest4 = mS (Sent (NP2 No (CN1 Man That Laughed)) Smiled) Minus
mtest5 = mS (Sent (NP2 Every (CN1 Man That (VP2 Didnt Laugh))) Smiled) Plus
mtest6 = mS (Sent (NP2 The (CN1 Woman That (VP2 Didnt Laugh))) Smiled) Plus
```

This gives:

```
NLNL> mtest1
Node + (Leaf +) (Node + (Leaf +) (Leaf -))
NLNL> mtest2
Node + (Node + (Leaf +) (Node - (Leaf -) (Leaf -))) (Leaf +)
NLNL> mtest3
Node + (Node + (Leaf +) (Node - (Leaf -) (Leaf -))) (Leaf -)
NLNL> mtest4
Node - (Node - (Leaf -) (Node + (Leaf +) (Leaf +))) (Leaf +)
NLNL> mtest5
Node + (Node + (Leaf +) (Node - (Leaf -) (Node - (Leaf -) (Leaf +)))) (Leaf +)
NLNL> mtest6
Node + (Node + (Leaf +) (Node 0 (Leaf 0) (Node 0 (Leaf 0) (Leaf 0)))) (Leaf +)
```

## 13 Work...

Sections 4 and 5 above define a partial order on any domain built from  $e$  (entities) and  $t$  (booleans), and Section 6 defines upward and downward monotonicity properties of functions in terms of this partial order. The partial order can be viewed as an extension of the logical consequence relation to arbitrary types built from  $e$  and  $t$ . This definition is from [1]. Proposals for syntactic calculi intended to cover this lifted consequence relation were made in [14, 3, 2]. The polarity marking algorithm in Section 11 above gives what can plausibly be viewed as the core of these calculi. Next on the agenda is the further investigation of the calculi:

- formal definition on the basis of suitable natural language fragments (see [11, 12] for inspiration),
- presentation of soundness and completeness proofs (completeness of the rules in [3] was conjectured in the paper, but never proved),
- complexity analysis (see again [12]),
- classification of degree of difficulty of reasoning tasks (on the basis of the mental models metaphor, or in other terms),
- investigation of human reasoning faculty guided by hypotheses deriving from the study of monotonicity calculi (first steps are in [6]).

This research agenda presents the investigation of the human reasoning faculty as a task in need of a proper logical foundation: to get the investigation off the ground, the calculi at the basis of the analysis should meet the usual logical standards of formal precision.

## References

- [1] BENTHEM, J. v. *Language in Action: categories, lambdas and dynamic logic*. Studies in Logic 130. Elsevier, Amsterdam, 1991.
- [2] BERNARDI, R. *Reasoning with Polarity in Categorical Type Logic*. PhD thesis, Uil-OTS, Utrecht University, 2002.
- [3] DOWTY, D. Negative polarity and concord marking in natural language reasoning. In *SALT Proceedings* (1994).
- [4] EIJCK, J. v. Generalized quantifiers and traditional logic. In *Generalized Quantifiers, Theory and Applications*, J. van Benthem and A. ter Meulen, Eds. Foris, Dordrecht, Dordrecht, 1985.
- [5] ENGLEBRETSSEN, G. Notes on the new syllogistic. *Logique et Analyse* 85–86 (1979), 111–120.
- [6] GEURTS, B., AND VAN DER SLIK, F. Monotonicity and processing load. *Journal of Semantics* 22, 97–117 (2005).
- [7] JOHNSON-LAIRD, P. *Mental Models; towards a cognitive science of language, inference and consciousness*. Cambridge University Press, 1983.
- [8] JONES, S. P., HUGHES, J., ET AL. Report on the programming language Haskell 98. Available from the Haskell homepage: <http://www.haskell.org>, 1999.
- [9] KNUTH, D. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.
- [10] MONTAGUE, R. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*, J. Hintikka, Ed. Reidel, 1973, pp. 221–242.

- [11] PRATT-HARTMANN, I. A two-variable fragment of English. *Journal of Logic, Language and Information* 12, 1 (2003), 13–45.
- [12] PRATT-HARTMANN, I. Fragments of language. *Journal of Logic, Language and Information* 13, 2 (2004), 207–223.
- [13] PURDY, W. A logic for natural language. *Notre Dame Journal of Formal Logic* 32 (1991), 409–425.
- [14] SÁNCHEZ, V. *Studies on Natural Logic and Categorical Grammar*. PhD thesis, University of Amsterdam, 1991.
- [15] SOMMERS, F. *The Logic of Natural Language*. Cambridge University Press, 1982.