Proof Nets for Multimodal Lambek Calculus

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

March 12, 2004

Abstract

This paper contains the full code of an implementation in Haskell [1], in ‘literate programming’ style [2], of proof nets for multimodal Lambek calculus, after [3].

Keywords: Proof nets, linear logic, multimodal Lambek calculus, categorial grammar, Haskell, literate programming.

1 Module Declaration

module MlcNets where

import List
import System

2 Modes, Formulas

Modes:
data ModeJ = ZeroJ | OneJ deriving (Eq,Ord)

instance Show ModeJ where
  show ZeroJ = "0"
  show OneJ = "1"

data ModeI = ZeroI | OneI deriving (Eq,Ord)

instance Show ModeI where
  show ZeroI = "0"
  show OneI = "1"

Formulas:

data Form = Atom String
           | Diamond ModeJ Form
           | Rbox ModeJ Form
           | Bullet ModeI Form Form
           | SlashL ModeI Form Form
           | SlashR ModeI Form Form
    deriving (Eq,Ord)

Finding the atoms:

atomic :: Form -> Bool
atomic (Atom _) = True
atomic _        = False

Useful abbreviations for example formulas:
3 Lexicon

Assign lists of formulas to words.
lexicon :: String -> [Form]
lexicon "Michael" = [np]
lexicon "Jan" = [np]
lexicon "the" = [det]
lexicon "man" = [n]
lexicon "woman" = [n]
lexicon "nice" = [adj]
lexicon "intelligent" = [adj]
lexicon "smiled" = [iv]
lexicon "talked" = [iv]
lexicon "liked" = [tv]
lexicon "admired" = [tv]
lexicon "considered" = [cv]
lexicon "that" = [rel0,rel1]
lexicon "who" = [rel1]
lexicon "whom" = [rel0]
lexicon _ = []

Lexical lookup for a list of input words:

lookupLex :: [String] -> [[Form]]
lookupLex [] = [[]]
lookupLex (word:words) = [ f:fs | f <- lexicon word, fs <- lookupLex words ]

Examples of use:

MlcNets> lookupLex ["that","liked","Michael"]
[[[(N \0 N) /0 (S /0 <0>[0~]NP)),(NP \0 S) /0 NP),NP],
 [[(N \0 N) /0 (NP \0 S)),((NP \0 S) /0 NP),NP]]
MlcNets> lookupLex ["whom","Michael","liked"]
[[[(N \0 N) /0 (S /0 <0>[0~]NP)),NP,((NP \0 S) /0 NP)]]

The scan function scans an input string and replaces punctuation marks by whitespace.

scan :: String -> String
scan [] = []
scan (x:xs) | x == '.' || x == '?' = ':'scan xs
            | otherwise = x :scan xs
The lexer will produce lists of words (strings) from the input string. The \texttt{lexer} function uses the predefined function \texttt{words} to split the input into separate words.

\begin{verbatim}
lexer :: String -> [String]
lexer = words . scan
\end{verbatim}

\section*{4 Graphs and Mgraphs}

Directed graphs consist of nodes and edges. If the nodes are of type \(a\), the edges are type \(b\) connections between two nodes (think of type \(b\) as the label type).

\begin{verbatim}
type Edge a b = (a,a,b)
\end{verbatim}

Useful projection functions for triples, to get at the components of an edge.

\begin{verbatim}
p1 :: (a,b,c) -> a
p1 (x,y,z) = x
p2 :: (a,b,c) -> b
p2 (x,y,z) = y
p3 :: (a,b,c) -> c
p3 (x,y,z) = z
\end{verbatim}

Graphs are pairs consisting of a list of nodes and a list of edges.

\begin{verbatim}
type Graph a b = ([a],[Edge a b])
\end{verbatim}

Since edges are triples, it makes sense to define the following operation on triples:

\begin{verbatim}
mapTriple :: (a -> b, c -> d, e -> f) -> [(a,c,e)] -> [(b,d,f)]
mapTriple (f,g,h) triples = [(f x, g y, h z) | (x,y,z) <- triples ]
\end{verbatim}
In terms of this, we define mappings on graphs:

```haskell
mapGraph :: (a -> b) -> (c -> d) -> Graph a c -> Graph b d
mapGraph f g (nodes,edges) = (map f nodes, mapTriple (f,f,g) edges)
```

For our application, we need a special kind of graph, where links can be from lists of nodes to lists of nodes (to cater for cases where two premisses get linked to a single conclusion and cases where one premiss is linked to two conclusions).

```haskell
type Link a b = ([a],[a],b)
```

Call an object with this kind of link a multigraph or Mgraph.

```haskell
type Mgraph a b = ([a],[Link a b])
```

Mappings on links:

```haskell
mapLink :: (a -> b) -> (c -> d) -> Link a c -> Link b d
mapLink f g (xs,ys,label) = (map f xs, map f ys, g label)
```

Mappings on Mgraphs:

```haskell
mapMgraph :: (a -> b) -> (c -> d) -> Mgraph a c -> Mgraph b d
mapMgraph f g (nodes,links) = (map f nodes, map (mapLink f g) links)
```

An important operation on Mgraphs is fusion of Mgraph nodes.
fuseMgraph :: (Eq a, Ord a) => (a, a) -> Mgraph a b -> Mgraph a b
fuseMgraph (x, y) (nodes, edges) = (nodes \ \ [two], map (mapLink f id) edges)
  where
    one = min x y
    two = max x y
    f = \ z -> if z == two then one else z

5 Proof Structures

Proof structures are a particular kind of Mgraph. For the edge labelling of proof structures
we use Tags. Tags are called Types in [3], but this term already has a specific meaning in the
Haskell context.

data Tag = Ldia ModeJ | Rdia ModeJ | Lrbox ModeJ | Rrbox ModeJ | Brack ModeJ
          | Lbullet ModeI | Rbullet ModeI | Lslashl ModeI | Rslashl ModeI | Lslashr ModeI | Rslashr ModeI | Concat ModeI
deriving Eq

Make tags showable in a pleasant way:
instance Show Tag where
  show (Ldia j) = "L<" ++ show j ++ "">
  show (Rdia j) = "R<" ++ show j ++ "">
  show (Lrbox j) = "L[" ++ show j ++ "]">
  show (Rrbox j) = "R[" ++ show j ++ "]">
  show (Brack j) = "<<" ++ show j ++ ">>
  show (Lbullet i) = "L." ++ show i
  show (Rbullet i) = "R." ++ show i
  show (Lslashl i) = "L\\" ++ show i
  show (Rslashl i) = "R\\" ++ show i
  show (Lslashr i) = "L/" ++ show i
  show (Rslashr i) = "R/" ++ show i
  show (Concat i) = "C." ++ show i

Useful functions for tag decomposition:

jTag :: Tag -> Maybe ModeJ
jTag (Ldia j) = Just j
jTag (Rdia j) = Just j
jTag (Lrbox j) = Just j
jTag (Rrbox j) = Just j
jTag (Brack j) = Just j
jTag _ = Nothing

iTag :: Tag -> Maybe ModeI
iTag (Lbullet i) = Just i
iTag (Rbullet i) = Just i
iTag (Lslashl i) = Just i
iTag (Rslashl i) = Just i
iTag (Lslashr i) = Just i
iTag (Rslashr i) = Just i
iTag (Concat i) = Just i
iTag _ = Nothing

Two of the tags are so-called structural tags:
To facilitate the mapping to hypothesis structures further on, we mark the formulas in our proof structures with an index.

```
type Fi = (Form, Int)
```

Proof structures are Mgraphs with annotations. An annotation on a proof structure is a function on its nodes that allows us to keep track of the formulas that were generated from the lexicon, and of the formula that is the goal of the query.

```
type Annotation = Int -> (String, String)
```

During the generation process, we mark one of the formula indices with the lexical string or the goal string, using the following function.

```
mark :: Int -> String -> String -> Annotation
mark i str1 str2 = \ m -> if i == m then (str1, str2) else ("",""")
```

Conversion from annotations to lists of pairs. First an auxiliary annotation function:

```
annotate :: (String, String) -> String
annotate = \ (x,y) -> if x == ""
    then y
    else if y == ""
        then x
        else x ++ "," ++ y
```

This is used in the conversion function:
A proof structure is a pair consisting of an Mgraph and an annotation.

```haskell
data PS = PS (Mgraph Fi Tag) Annotation
```

Showing a proof structure:

```haskell
instance Show PS where
    show (PS (nodes,links) annotation) =
        show nodes ++ show links ++ show (ann2pairs dom annotation)
        where dom = map snd nodes
```

The hypotheses of a proof structure are the formulas that do not occur as the conclusion of a link:

```haskell
hypotheses :: PS -> [Fi]
hypotheses (PS (nodes,links) _) =
    [ f | f <- nodes, notElem f conclusions ]
    where conclusions = [ c | (xs,ys,tag) <- links, c <- ys ]
```

The conclusions of a proof structure are the formulas that do not occur as the premisse of a link:

```haskell
conclusions :: PS -> [Fi]
conclusions (PS (nodes,links) _) =
    [ f | f <- nodes, notElem f premisses ]
    where premisses = [ c | (xs,ys,tag) <- links, c <- xs ]
```
6 From Formulas to Proof Structures

Expanding a proof structure by means of formula decomposition. We distinguish two kinds of expansion: goal expansion (or: conclusion expansion) and premisse expansion (or: hypothesis expansion).

The five arguments of `expand` are, in order:

1. a pointer to the next available index,
2. all formulas encountered so far,
3. the formulas still to be expanded as premisses,
4. the formulas still to be expanded as goals,
5. the links established so far.

The expansion process terminates when no more formulas are to be expanded (i.e., when the third and fourth arguments are both empty lists).

```haskell
expand :: Int -> [Fi] -> [Fi] -> [Fi] -> [Link Fi Tag] -> Mgraph Fi Tag
expand n all [] [] links = (all,links)
```

Expansion of atoms, diamonds and rboxes as premisses.

```haskell
expand n all ((Atom s,m):xs) ys links =
  expand n ((Atom s,m):all) xs ys links
expand n all ((Diamond j f,m):xs) ys links =
  expand (n+1) ((Diamond j f,m):all) ((f,n):xs) ys (link:links)
  where link = ([[(Diamond j f,m)],[(f,n)]]),Ldia j
expand n all ((Rbox j f,m):xs) ys links =
  expand (n+1) ((Rbox j f,m):all) ((f,n):xs) ys (link:links)
  where link = ([[(Rbox j f,m)],[(f,n)]]),Lrbox j
```

Expansion of bullet and slash formulas as premisses.
Expansion of atoms, diamonds and rboxes as goals.

Expansion of bullet and slash formulas as goals.
Mapping a premisse formula to a proof structure by means of expansion, marking the premisse formula with a lexical string.

```
expandPremisse :: String -> Form -> PS
expandPremisse str form = PS (expand 1 [] [(form,0)] [] []) annotation
  where annotation = mark 0 str ""
```

This gives:

```
MlcNets> expandPremisse "smiled" iv
[(NP,1),(S,2),((NP \0 S),0)][[(NP,1),((NP \0 S),0)],[(S,2)],L\0)](0,"smiled")
MlcNets> hypotheses (expandPremisse "smiled" iv)
[(NP,1),((NP \0 S),0)]
MlcNets> conclusions (expandPremisse "smiled" iv)
[(S,2)]
```

This proof net for the formula `iv`, used as a premisse, indicates that `NP`, `NP \0 S` are the hypotheses of the structure, and `S` is the conclusion.

Mapping a goal formula to a proof structure by means of expansion, marking the goal formula with a goal string.

```
expandGoal :: Form -> PS
expandGoal form = PS (expand 1 [] [] [(form,0)] []) annotation
  where annotation = mark 0 "" "GOAL"
```

Here is the goal expansion of `iv`:

```
MlcNets> expandGoal iv
[(S,2),(NP,1),((NP \0 S),0)][[(S,2)],[(NP,1),((NP \0 S),0)],R\0)](0,"GOAL")
MlcNets> hypotheses (expandGoal iv)
[(S,2)]
MlcNets> conclusions (expandGoal iv)
[(NP,1),((NP \0 S),0)]
```

7 Correction Structures

Correction structures are Mgraphs with integers as nodes and links labelled by tags as edges:
type CS = Mgraph Int Tag

A useful display function, for when the structures get large:

```haskell
display :: Show a => [a] -> IO()
display [] = return ()
display (x:xs) = do print x
display xs
```

Displaying a CS:

```haskell
displayCS :: CS -> IO()
displayCS (nodes, links) = do print nodes
display links
```

8 Hypothesis Structures

Hypothesis structures are pairs \((M, \Lambda)\) consisting of a correction structure \(M\) and a node labelling \(\Lambda\) mapping each node to a pair \((X, Y)\) with \(X\) and \(Y\) lists consisting of formula/string pairs. \(X\) is the list containing the upper label and the lexical annotation. An upper label indicates that the node is a hypothesis node, a lexical annotation indicates that the hypothesis derives from the lexicon. \(Z\) is the list containing the lower label and the goal annotation, indicating whether the node is a conclusion node, and whether this conclusion is the goal of the query. Thus, if, e.g., node \(n\) has upper label \(A\) and no lower label, and is not linked to a lexical item, then the node is a hypothesis \(A\), and we have \(\Lambda(n) = \left([A,""], []\right)\).

```haskell
type Labelling = Int -> ([(Form,String)],[(Form,String)])
```

Instead of working directly with labellings, it is often more convenient to only look at the part where the upper or the lower label is non-empty:

```haskell
type Labels = [(Int,String,[Form],[Form])]
```
A function for converting labellings to labels:

```haskell
lab2labels :: [Int] -> Labelling -> Labels
lab2labels dom f =
[ (n,
   ann ((map snd (hyps n)) ++ (map snd (concs n))),
   map fst (hyps n),
   map fst (concs n)) | n <- dom, (hyps n /= [] || concs n /= []) ]
where
  hyps i   = fst (f i)
  concs i  = snd (f i)
  ann []   = ""
  ann [x]  = x
  ann [x,y] = annotate (x,y)
```

When a goal node $g$ and a premisse node $p$ are fused during linking, in the new structure, the new node need not be a conclusion anymore: it is only a conclusion if $p$ was a conclusion in the old structure. Similarly, the new node need not be a premisse anymore: it is only a premisse if $g$ was a premisse in the old structure. So the premisse label (upper label) of the new node is the upper label of $g$, and the conclusion label (lower label) of the new node is the lower label of $p$. This motivates the following labelling adjustment function.

```haskell
adjustLabelling :: (Int,Int) -> Labelling -> Labelling
adjustLabelling (goalNode,premisseNode) lab = \ z ->
  if z == goalNode || z == premisseNode
  then  (fst (lab goalNode),
          snd (lab premisseNode))
  else lab z
```

When a goal node $g$ and a premisse node $p$ are fused during contraction, the upper label of the new node is the upper label of $p$ and the lower label of the new node is the lower label of $g$.

```haskell
adjustLabC :: (Int,Int) -> Labelling -> Labelling
adjustLabC = adjustLabelling . (\ (x,y) -> (y,x))
```

Hypothesis structures:
data HS = HS CS Labelling

A show function for hypothesis structures, showing only the non-trivial part of the labelling.

```haskell
instance Show HS where
  show (HS (nodes,edges) f) = show nodes ++ show edges ++ show (lab2labels nodes f)
```

Displaying a HS (useful when the structure gets large):

```haskell
displayHS :: HS -> IO()
displayHS (HS (nodes,links) f) = do displayCS (nodes,links)
                           display (lab2labels nodes f)
```

We can use the labelling to find out whether a node is a lexical node or a goal node in a HS.

```haskell
isLexical :: Int -> HS -> Bool
isLexical i (HS _ lab) =
  fst (lab i) /= [] && (snd . head . fst . lab) i /= ""

isGoal :: Int -> HS -> Bool
isGoal i (HS _ lab) =
  snd (lab i) /= [] && (snd . head . snd . lab) i /= ""
```

The lexical annotation of a node in a HS:

```haskell
lexAnn :: HS -> Int -> String
lexAnn hs@(HS _ lab) i |
  isLexical i hs = (snd . head . fst . lab) i |
  otherwise = ""
```

The hypothesis nodes of a HS are the formulas with an upper label:
hypsHS :: HS -> [(Int, Form, String)]
hypsHS (HS (nodes, links) f) =
  [ (i, form, string) | i <- nodes, (form, string) <- fst (f i) ]

The hypothesis nodes that are to be linked are the atomic premisse nodes without lexical annotation:

linkinghypsHS :: HS -> [(Int, Form)]
linkinghypsHS hs@(HS (nodes, links) f) =
  [ (i, form) | (i, form, str) <- hypsHS hs, atomic form, str == "" ]

The conclusion nodes of a HS are the formulas with a lower label.

concsHS :: HS -> [(Int, Form, String)]
concsHS (HS (nodes, links) f) =
  [ (i, form, str) | i <- nodes, (form, str) <- snd (f i) ]

The conclusion nodes that are to be linked are the atomic conclusion nodes without goal annotation.

linkingconcsHS :: HS -> [(Int, Form)]
linkingconcsHS hs@(HS (nodes, links) f) =
  [ (i, form) | (i, form, str) <- concsHS hs, atomic form, str == "" ]

If a hypothesis or conclusion is linked by means of a binary modality then we can check its direction:
data Direction = Le | Ri deriving (Eq,Show)

direction :: Int -> HS -> [Direction]
direction i (HS (nodes,edges) lab) = 
  [ Le | elem i leftnodes ]
  ++
  [ Ri | elem i rightnodes ]

  where 
  leftnodes = map head binaryLinks
  rightnodes = map (head.tail) binaryLinks
  binaryLinks = filter (\ x -> length x == 2) edgePositions
  edgePositions = 
    map (\ x -> p1 x) edges ++ map (\ x -> p2 x) edges

Fusion of HS nodes: the first node of the fusion pair \((n, m)\) is the goal node, the second the hypothesis node. The order is important for the correct re-annotation of the new node.

fuseHS :: HS -> (Int,Int) -> HS
fuseHS (HS ps f) (goalNode,hypothesisNode) =
  HS (fuseMgraph (goalNode,hypothesisNode) ps) 
     (adjustLabelling (goalNode,hypothesisNode) f)

9 Disjoint Union of Hypothesis Structures

The disjoint union of two hypothesis structures is again a hypothesis structure. If we assume that the domains are listed in increasing order, we can proceed as follows:

unionHS :: HS -> HS -> HS
unionHS (HS ps1@(nodes1,links1) lab1) (HS ps2 lab2) = HS ps3 lab3

  where 
  k = if null nodes1 then 0 else (last nodes1) + 1
  (nodes2,links2) = mapMgraph (\ n -> n + k) id ps2
  ps3 = (nodes1 ++ nodes2, links1 ++ links2)
  lab3 = \ m -> if m < k then lab1 m else lab2 (m - k)

The trivial HS:
trivHS :: HS
trivHS = HS ([],[]) (\ x -> undefined)

Extend unions to a list of hypothesis structures:

unionHSs :: [HS] -> HS
unionHSs = foldl unionHS trivHS

10 From Proof Structures to Hypothesis Structures

In order to transform proof structures into hypothesis structures, we have to transform their tensor links. We assume that we start out with a correct formula link.

transformTensorLink :: Link a Tag -> Link a Tag
transformTensorLink ([p],[c], Rdia j) = ([p],[c], Brack j)
transformTensorLink ([p],[c], Lrbox j) = ([p],[c], Brack j)
transformTensorLink ([p1,p2],[c], Lslashl i) = ([p1,p2],[c], Concat i)
transformTensorLink ([p1,p2],[c], Lslashr i) = ([p1,p2],[c], Concat i)
transformTensorLink parlink = parlink

From proof structures to hypothesis structures (the function $S \mapsto \hat{S}$ from [3]):

ps2hs :: PS -> HS
ps2hs ps@(PS (nodes,links) annotation) = HS (ns,ilinks) lab
where
  f = \ (_,i) -> i
  ns = sort (map f nodes)
  ilinks = map ((mapLink f id) . transformTensorLink) links
  lab = \ i ->
    ( [ (h, fst (annotation i)) | (h,j) <- hypotheses ps, i == j ],
      [ (g, snd (annotation i)) | (g,k) <- conclusions ps, i == k ]
    )

This gives, e.g.:
11 Building Hypothesis Structures for Parsing Problems

Building a HS for a list of string/formula pairs and a given goal formula is done by expanding each of the formulas into a proof structure, converting these proof structures to hypothesis structures, and taking the disjoint union of these HSs.

```haskell
buildHS :: [(String, Form)] -> Form -> HS
buildHS premisses goal =
    unionHSs (premissesHSs ++ [goalHS])
    where premissesHSs = [ ps2hs (expandPremisse w f) | (w,f) <- premisses ]
    goalHS = ps2hs (expandGoal goal)
```

Use this to build a list of HSs for a given list of input words, with different HSs for all the different ways of linking the input words to formulas from the lexicon.

```haskell
problemHSs :: [String] -> Form -> [HS]
problemHSs words goal =
    [ buildHS (zip words premisses) goal | premisses <- lookupLex words ]
```

This gives, e.g.:

```
MlcNets> mapM displayHS (problemHSs ["liked", "Michael"] iv)
[0,1,2,3,4,5,6,7,8]
([(3,1),(4)],C.0)
([(0,2),(1)],C.0)
([(8),(7,6)],R\0)
```
12 Linking up Hypothesis Structures

Checking whether a HS is linkable: the numbers of linkable atomic premisses and conclusions of each atomic kind should be the same.

```haskell
linkable :: HS -> Bool
linkable hs@(HS (nodes,links) lab) = hyps == concs
  where
    hyps = sort (map snd (linkinghypsHS hs))
    concs = sort (map snd (linkingconcsHS hs))
```

A linking is a pairing of linkable atomic premisses against linkable atomic conclusions in a HS.

```haskell
type Linking = [(Int,Int)]
```

We are interested in the list of all possible linkings for a HS. We start out from of a function that gives all the permutations of a finite list:

```haskell
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map (insrt x) (perms xs))
  where
    insrt :: a -> [a] -> [[a]]
    insrt x [] = [[x]]
    insrt x (y:ys) = (x:y:ys) : map (y:) (insrt x ys)
```

This gives:
Linking up two lists of index/formula pairs is done by trying to match the first list against all permutations of the second list. The lists agree if they have the same lengths and at each position the formula components are the same.

```haskell
agree :: [(Int,Form)] -> [(Int,Form)] -> Bool
agree [] [] = True
agree (_:_) [] = False
agree [] (_:_) = False
agree ((_,f):xs) ((_,f'):ys) = f == f' && agree xs ys
```

Finding the list of all matching linkings:

```haskell
linkings:: HS -> [Linking]
linkings hs =
  [ zipWith (\ x y -> (fst x, fst y)) concs hyps’ |
    hyps’ <- perms hyps, agree concs hyps’ |
  ]
where
  concs = linkingconcsHS hs
  hyps = linkinghypsHS hs
```

Using a linking to link up a HS (it is assumed that the linking is correct for the HS):

```haskell
linkup :: HS -> Linking -> HS
linkup = foldl fuseHS
```

Use the linkings to construct all possible ways of linking up a HS:

```haskell
linkedHSs :: HS -> [HS]
linkedHSs hs = if not (linkable hs)
  then []
  else [ linkup hs link | link <- linkings hs ]
```
Construct the list of linked HSs for a given list of input words and a given goal formula:

\[
\text{makeHSs :: [String] -> Form -> [HS]}
\]

\[
\text{makeHSs words form = concat (map linkedHSs (problemHSs words form))}
\]

## 13 Contraction of Hypothesis Structures

Call a function of the form \( \text{HS -> \[(Int,Int,[Int],[Link Int Tag])\]} \) a C rule:

\[
\text{type Crule = HS -> \[(Int,Int,[Int],[Link Int Tag])\]}\]

The result of the following functions for contraction of HSs is a list of triples \((g, p, N, E)\) consisting of a goal node \(g\) and a premisse node \(p\) to be fused, and a list of nodes \(N\) and a list of links \(E\) to be deleted. These contractions we call the Crules.

The \(L \diamond_j\) rule:

\[
\text{ldiaM :: Crule}
\]

\[
\text{ldiaM (HS (nodes,links) lab) =}
\]

\[
[(g,p,[n], [link1,link2]) |
\quad \text{link1@([p],[n], Ldia j) <- links,}
\quad \text{link2@([n'],[g],Brack j') <- links,}
\quad n == n',
\quad j == j' ]
\]

The \(L \bullet_i\) rule:

\[
\text{lbulletM :: Crule}
\]

\[
\text{lbulletM (HS (nodes,links) lab) =}
\]

\[
[(g,p,[n,k], [link1,link2]) |
\quad \text{link1@([p],[n,k], Lbullet i) <- links,}
\quad \text{link2@([n',k'],[g], Concat i') <- links,}
\quad n == n',
\quad k == k',
\quad i == i' ]
\]
The $R_{\square j}$ rule:

\[
\text{rrboxM :: Crule}
\]

\[
\text{rrboxM (HS (nodes, links) lab) = }
\]

\[
[(g, p, [n], [link1, link2]) |
\]

\[
\text{link10([p],[n], Brack } j) \gets \text{ links,}
\]

\[
\text{link20([n’],[g],Rrbox } j’) \gets \text{ links,}
\]

\[
n = n’,
\]

\[
j = j’
\]

The $R\setminus_i$ rule:

\[
\text{rslashlM :: Crule}
\]

\[
rslashlM (HS (nodes, links) lab) =
\]

\[
[(g, p, [n, k], [link1, link2]) |
\]

\[
\text{link10([n,p],[k], Concat } i) \gets \text{ links,}
\]

\[
\text{link20([k’],[n’,g], Rslashl } i’) \gets \text{ links,}
\]

\[
n = n’,
\]

\[
k = k’,
\]

\[
i = i’
\]

The $R/i$ rule:

\[
\text{rslashrM :: Crule}
\]

\[
rslashrM (HS (nodes, links) lab) =
\]

\[
[(g, p, [n, k], [link1, link2]) |
\]

\[
\text{link10([p,k],[n], Concat } i) \gets \text{ links,}
\]

\[
\text{link20([n’],[g,k’], Rslashr } i’) \gets \text{ links,}
\]

\[
n = n’,
\]

\[
k = k’,
\]

\[
i = i’
\]

Contraction using a Crule. This uses an auxiliary function contract’ that takes a Crule as its first argument. In a Crule $[(\text{Int, Int, [Int], [Link Int Tag]})]$, the first two integers are the goal node and the premise node that are fused. The integer list and the link list give the nodes and links to be deleted.

24
Here is the list of all contraction rules:

```haskell
cRules :: [Crule]
cRules = [ldiaM, lbulletM, rrboxM, rslashlM, rslashrM]
```

Trying out all contraction rules on a HS:

```haskell
tryCrules :: HS -> [HS]
tryCrules hs = concat [ contract rule hs | rule <- cRules ]
```

### 14 Correction Patterns

The structural rules check for certain patterns in the linking of a hypothesis structure, and replace these patterns by new patterns. This structural adjustment does not affect the input-output structure of the replaced pattern: the adjustment affects $N$ premises by reordering them, and the pattern has a single conclusion that is unaffected.

This means that we can specify each structural conversion as a pair of correction structures (a correction pattern).

The pattern for $\rightarrow \text{Ass}_{0,0} \rightarrow$: 

```haskell
contract :: Crule -> HS -> [HS]
contract rule hs = contract' (rule hs) hs where
contract' :: [([Int],[Int],[Link Int Tag])] -> HS -> [HS]
contract' [] hs = []
contract' ((g,p,ns,ls):rest) hs@(HS (nodes,links) lab) =
  (HS (nodes',links'') lab'): contract' rest hs
  where
    one   = min g p
    two   = max g p
    nodes' = nodes \ (two:ns)
    links' = links \ ls
    links'' = map (mapLink (\x -> if x == two then one else x) id) links'
    lab'  = adjustLabC (g,p) lab
```
ass00 :: (CS,CS)
ass00 = (([0,1,2,3,4,5],
    [
        ([0,4],[3],Concat ZeroI),
        ([1,5],[4],Concat ZeroI),
        ([2],[5],Brack ZeroJ)
    ]),
    ([0,1,2,3,4,5],
    [
        ([0,1],[4],Concat ZeroI),
        ([4,5],[3],Concat ZeroI),
        ([2],[5],Brack ZeroJ)
    ]))

The pattern for \( \text{→ Ass}_{0,1} \rightarrow \):

ass01 :: (CS,CS)
ass01 = (([0,1,2,3,4,5],
    [
        ([0,4],[3],Concat ZeroI),
        ([1,5],[4],Concat ZeroI),
        ([2],[5],Brack OneJ)
    ]),
    ([0,1,2,3,4,5],
    [
        ([0,1],[4],Concat ZeroI),
        ([4,5],[3],Concat ZeroI),
        ([2],[5],Brack OneJ)
    ]))

The pattern for \( \text{→ MxCom}_{0,0} \rightarrow \):

26
mxCom00 :: (CS, CS)
mxCom00 = ([0, 1, 2, 3, 4, 5],
           
           ([0, 5], [4], Concat ZeroI),
           ([4, 2], [3], Concat ZeroI),
           ([1], [5], Brack ZeroJ)
        ),
        ([0, 1, 2, 3, 4, 5],
        
        ([0, 2], [4], Concat ZeroI),
        ([4, 5], [3], Concat ZeroI),
        ([1], [5], Brack ZeroJ)
    )
)

The pattern for \( \rightarrow \text{MxCom}_{0,1} \rightarrow \):

mxCom01 :: (CS, CS)
xCom01 = (([0, 1, 2, 3, 4, 5],
           
           ([0, 5], [4], Concat ZeroI),
           ([4, 2], [3], Concat ZeroI),
           ([1], [5], Brack OneJ)
        ),
        ([0, 1, 2, 3, 4, 5],
        
        ([0, 2], [4], Concat ZeroI),
        ([4, 5], [3], Concat ZeroI),
        ([1], [5], Brack OneJ)
    )
)

The pattern for \( \rightarrow \text{Com}_{0,0} \rightarrow \):\(^1\)

\(^1\)From Puice and Moot [3], but ‘real’ linguists (Moortgat, Steedman) disagree.
Displaying a correction pattern:

displayCP :: (CS,CS) -> IO()
displayCP (before,after) = do displayCS before
                                putStrLn "===>"
                                displayCS after

Here is the list of all correction patterns (with com00 left out):

correctionPatterns :: [(CS,CS)]
correctionPatterns = [ass00, ass01, mxCom00, mxCom01]

15 Structural Conversion of Hypothesis Structures

For matching a HS against a CS, we first define the match of a pair of links:

matchLink :: Link Int Tag -> Link Int Tag -> [Linking]
matchLink (xs1,ys1,tag1) (xs2,ys2,tag2)
  | tag1 == tag2 = [ zip (xs1 ++ ys1) (xs2 ++ ys2) ]
  | otherwise    = []

This function is used for matching a link against a list of links:
match :: Link Int Tag -> [Link Int Tag] -> [(Link Int Tag, Linking)]
match link links = [(link1, linking) | link1 <- links, linking <- matchLink link link1]

To combine the results of several link matches, we define fusion of linkings:

fuseLinkings :: Linking -> Linking -> [Linking]
fuseLinkings [] ls = [ls]
fuseLinkings ls [] = [ls]
fuseLinkings ((x,y):ls) ls' =
  if lookup x ls' == Nothing
    then [(x,y): rest | rest <- fuseLinkings ls ls']
    else
      if lookup x ls' == Just y
        then [(x,y): rest | rest <- fuseLinkings ls (ls' \ [(x,y)])]
        else []

The results of matching a list of links against a list of links.

matchLinks :: [Link Int Tag] -> [Link Int Tag] -> [Linking]
matchLinks [] ls = [[]]
matchLinks ls [] = [[]]
matchLinks (link:links) links1 =
  concat
  [ fuseLinkings linking1 linking2 |
    (link1,linking1) <- match link links1,
    linking2 <- matchLinks links (links1 \ [link1]) ]

Converting a linking to a function, using renaming of new elements to avoid clashes with old elements.
linking2fct :: [Int] -> [Int] -> Linking -> Int -> Int
linking2fct new old linking =
  \ x -> if elem x (map fst linking) then head [ z | (y,z) <- linking, x == y ]
  else if elem x new then x + k else x
  where k = if null old then 0 else (last old) + 1

Matching a HS against a pattern:

matchPattern :: (CS,CS) -> HS -> [(CS,CS)]
matchPattern (before@(nodes,links),after@(nodes1,links1)) (HS (nodes2,links2) lab) =
  [ (mapMgraph f id before, mapMgraph f id after) |
    f <- map (linking2fct (nodes1 \ nodes) nodes2) (matchLinks links links2) ]

Applying a correction pattern. Note that the new nodes that appear in the structure are internal nodes. Since these nodes are neither hypotheses nor conclusions of the new structure, we label them with ([],[]). This is done by means of the auxiliary function adjustLab.

applyPattern :: (CS,CS) -> HS -> [HS]
applyPattern pattern hs@(HS (nodes,links) lab) =
  [ HS (sort ((nodes \ ns1) ++ ns2), union (links \ ls1) ls2) (adjustLab (ns2 \ ns1) lab) |
    ((ns1,ls1),(ns2,ls2)) <- matchPattern pattern hs ]
  where adjustLab dom f = \ x -> if elem x dom then ([],[]) else f x

Trying out all correction patterns from the list:

tryCorrections :: HS -> [HS]
tryCorrections hs =
  concat [ applyPattern pattern hs | pattern <- correctionPatterns ]
16 Hypothesis Trees

A HS is a hypothesis tree for a list of words if

1. it has only structural links,

2. the premisses are exactly the nodes with lexical markings, the only conclusion is the node with a ‘GOAL’ marking, and the lexical premisses appear in the right order in the tree.

To check the first of these conditions, we can use `structuralTag`. For the second, we have to be able to trace the premisses from a node in a HS.

```haskell
tracePrems :: HS -> Int -> [Int]
tracePrems hs@(HS (nodes,edges) lab) i
    | isLexical i hs = [i]
    | otherwise = concat (map (tracePrems hs) is)
        where is = [ j | (x,z,t) <- edges, j <- x, elem i z ]
```

Tracing the premisses from the goal node, and checking the lexical labelling yields a list of words to be checked against the input word list.

```haskell
traceP :: HS -> [String]
traceP hs@(HS (nodes,edges) lab) =
    map (lexAnn hs) (concat (map (tracePrems hs) is))
        where is = [ i | i <- nodes, isGoal i hs ]
```

Use this in the check for valid hypothesis trees:

```haskell
hypothesisTree :: [String] -> HS -> Bool
hypothesisTree words hs@(HS (nodes,links) lab) =
    all (\ (x,y,z) -> structuralTag z) links
    &&
    traceP hs == words
```
17 Parsing

Processing for a list of words (the lexical input) and a HS (the linked HS built for that input):

```
processHS :: [String] -> HS -> [HS]
processHS words hs =
    if hypothesisTree words hs
    then [hs]
    else
        concat (map (processHS words) ((tryCrules hs) ++ (tryCorrections hs)))
```

Processing for a string (the lexical input) and a formula (the goal formula for that input):

```
process :: String -> Form -> [HS]
process string form = concat (map (processHS words) (makeHSs words form))
    where words = lexer string
```

Parsing:

```
parse :: String -> Form -> IO()
parse string form =
    let results = process string form
    in
        if null results then putStrLn "no parse"
        else mapM_ displayHS results
```

The same in verbose style:
parseVerbose :: String -> Form -> IO()
parseVerbose string form =
  let results = process string form
  in
    if null results
    then
      do (putStr . show) string
      putStrLn "no parse"
    else
      do (putStr . show) string
      mapM_ displayHS results

18 Test Suite

examples =
  [ ("Jan liked Michael.", s),
    ("Jan liked the man that liked Michael.", s),
    ("the man that liked Michael smiled.", s),
    ("the man that Michael liked smiled.", s),
    ("the man whom Michael liked smiled.", s),
    ("the man whom liked Michael smiled.", s),
    ("man who considered Michael intelligent.", n),
    ("man who Michael considered intelligent.", n) ]

Run the examples through a test sequence:

test :: IO()
test = sequence_ [ parseVerbose string form | (string, form) <- examples ]
19 Module for Standalone Use

Module declaration:

```haskell
module Main
where
  import MlcNets
  import System
```

Definition of main function:

```haskell
main :: IO()
main = do args <- getArgs
         parse (concat args) s
```

References

