Formalizing the Graphical Notation of Proof Structures for Lambek-Grishin Calculus

Sjoerd Dost

Abstract

In this paper a method of drawing proof structures for **LG** is presented. This method is fully deterministic and produces readable depiction of the structure. The program at the end of the paper implements this method and can draw a proof structure for any formula in LaTeX. We only consider structures that are a direct result of lexical unfolding.

Introduction

The Lambek-Grishin calculus (**LG**) was first introduced by Grishin in 1983 [2] as an extension of Lambek-calculus. Moortgat in [3] gives a clear overview of this logic and its notation. We will use this notation, of which a brief summary can be found in Figure 1. This paper will focus on proof structures, hypergraphs usable for constructing proofs. Since this paper is about drawing these structures, we mostly define them in graphical terms. Of course correspondence between graphics and logic is pointed out, but the exact logical background will only be relevant in motivating a certain graphical notation.

Figure 1: Formula language for Lambek-Grishin calculus (from [4])

$$\begin{aligned} A, B \ ::= \ & p \mid \\ & A \otimes B \mid B \backslash A \mid A / B \mid \\ & A \oplus B \mid A \oslash B \mid B \oslash A \end{aligned}$$

Proof structures are hard to draw. In the literature some aspects of their graphical representation have been formalized. Still structures can be drawn in several ways and the literature is not consistent in doing so. This paper presents a way of drawing these structures that is formal and rigid, with no room for interpretation. We also present a program capable of automatically creating a structure corresponding to a formula and drawing it in LaTeX in this way. We acknowledge that in some cases the structures drawn here could have been drawn in a more readable fashion, but hope that this (minor) lack of clarity is more than made up for in consistency.

Proof Nets and Proof Structures

Proof nets were introduced for linear logic in 1987 by Girard [1]. Their extension for **LG** has been shown in 2002 by Moot and Puite [6] and expanded upon in [5] and [4]. Proof nets are a graphical equivalent of a sequent proof in **LG**. Using this graphical method of proving hides much of the structural rules in a sequent proof, making the proof more readable. What follows is a graphically oriented description of proof nets.

Proof structures for **LG** are hypergraphs, consisting of hyperedges and vertices. The hyperedges correspond to the logical rules and the vertices to formulas. Proof structures are drawn as a node-link diagram, in which vertices are drawn

as a dot and edges as a circle connected to its vertices by lines. This circle can be filled either white or black, called a tensor or cotensor respectively. Each edge is a set of exactly three vertices, so proof structures are 3-uniform hypergraphs. Vertices are divided in premises and conclusions for each edge. From now on we will use the term *link* whenever we mean an edge. All links have one or two premises.

In proof structures the orientation of vertices relative to the links that contain them has meaning. In a top-down approach we draw a link's premises above it and its conclusions below it. Structures as a whole may be rotated but for clarity the top-down approach is considered standard. If a link has more than one premise its premises are ordered from left to right, and likewise for its conclusions. This is an important distinction for correctly drawing these structures. In this paper we divide links in one- and two-premise links. One-premise links have a *top, bottom-left* and *bottom-right* vertex. Two-premise links have a *top-left, top-right* and *bottom* vertex.

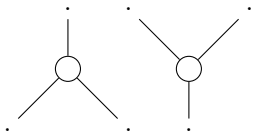


Figure 2: Links with one or two premises

Proof structures are defined such that each formula (vertex) is at most once the premise and at most once the conclusion of a link. Vertices can be internal, meaning that they are both premise and conclusion (thereby linking two links together). A vertex that's not included in the conclusions of any link is a *hypothesis* of the structure. Likewise, a vertex that's not included in the premises of any link is a *conclusion* of the structure. A proof structure with hypotheses $H_1, \ldots H_m$ and conclusions $C_1, \ldots C_n$ is a proof structure of sequent $H_1, \ldots H_m \Rightarrow C_1, \ldots C_n$ [4].

To prove a sequent we first create a proof structure for each formula. These are connected to each other, creating one structure, by means of identifying formulas. We have a proof for the sequent if this resulting structure can be reduced to a proof net, adhering to certain further constraints. Connecting and proving is beyond the scope of this paper. We consider a proof structure for a formula created by means of lexical unfolding.

Lexical Unfolding

We say a proof structure corresponding to the hypothesis unfolding of this formula is a proof structure for this formula. Any formula can be lexically unfolded with either positive or negative polarity. This is respectively called *hypothesis* and *conclusion unfolding*. The accompanying code shows hypothesis unfolding by default, with an option for conclusion unfolding. Lexical unfolding has a simple algorithm to create a structure corresponding to the formula.

The algorithm takes a formula ϕ and a boolean h signifying hypothesis or conclusion. This boolean is True by default for hypothesis unfolding. We start by creating a proof structure with a single vertex n. This vertex is its main vertex and defines it uniquely. Vertex n is decorated with ϕ. We call the algorithm and as arguments pass it n and h. If ϕ is an atomic formula, we stop here. Otherwise, we locate its main connective c and split the formula in two subformulas ϕ_1 and ϕ_2. Here ϕ_1 is the part of the formula left of c and ϕ_2 the part to the right. We look up the link corresponding to the rule eliminating or introducing this operator (depending on h) in a table (see [4] Figure 3). This link is connected to n and added in the proof structure's list of links. Its other vertices are recursively unfolded by calling the algorithm again for each vertex. The boolean flag given to these recursive calls correspond to the vertex appearing in the link's hypotheses or conclusions. Recursion stops when atomic formula are unfolded, as described above. Finally the algorithm returns the unfolded structure.

Graph drawing heuristics

Graphs can be drawn in many ways, each with its own merits. To measure how good a graphical representation is we need heuristics. Many

criteria for good graph drawing have been debated, of which we name a few. These criteria are numbered in order of importance.

1. Above all, the drawing should be readable. That is, the underlying structure must be intelligible.

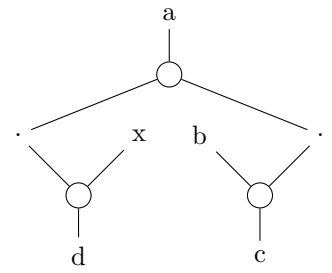
2. The number of crossing edges should be as low as possible.

3. The total area taken should be minimized. This can be measured as the area of its smallest enclosing box.

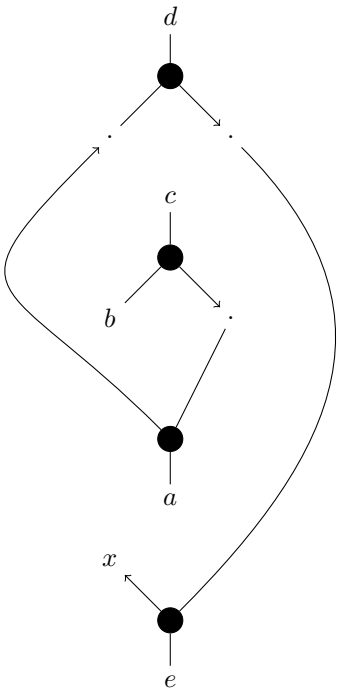
Since our graph must conserve both up-down and left-to-right ordering in the structure, we cannot hope to fulfill all these criteria in equal measure. Both rotating and mirroring a link are out of the question. If we would allow them we would directly introduce ambiguity in our drawing.

We take criterion 1 to be the most important and all others to be subgoals leading towards it. In fact, we will pose a restriction on our drawing making sure to maintain readability: all vertices will be drawn as close as possible to the links containing them (up to a minimum distance). For internal vertices this means as close as possible to one of the two links that contain them. All links will resemble as closely as possible one of the two layouts shown in Figure 2. Note that although only tensors are shown the same goes for cotensors.

The result of lexical unfolding is a connected proof structure. Its main vertex n can be taken as a unique starting point for traversing the graph. A naive way of displaying the structure is by printing this vertex at coordinates (0,0) and traversing the structure, printing each connected link or vertex in position relative to this main vertex. By creating equal spacing between two premises or two conclusions of the same link we maintain symmetry. This will become problematic for structures that are not trees: multiple links are now drawn in the same place. A structure with inherent overlapping using this method is shown in Figure 3.



(a) $x :: (a \oslash (b \backslash c)) \backslash d$



(b) $x :: e \oslash ((a \oslash (b \backslash c)) \backslash d)$

Figure 3: Inherent overlap

The structure in $3a$ can be drawn quite symmetrical. For the structure in $3b$ we are tempted to replace x in $3a$ with another link (abstracting over tensors and cotensors), but then either way symmetry is broken. Note that the overlap would occur between the top two links in $3b$. The solution is to give up this tree-like symmetry and draw links underneath each other. This means that $3a$ is not the way we would formally draw a structure like this. In some cases such as $3a$ symmetry might be used to draw a clear picture, but it is impossible to generalize this way of drawing. We therefore do not use symmetry

as a way of adhering to criterion 1. Our solution will try to make up for this loss in possible clarity.

Note that criterion 2 may be fully met when drawing proof structures that are planar. We do not give a solution striving for zero crossing links since proof structures can be non-planar. We merely wish to avoid lots of crossings.

Displaying the structure

Instead of drawing the structure from the viewpoint of a single vertex or link at a time, we now consider a method that considers the structure as a whole. We print all links in the structure underneath each other and then draw vertices and lines accordingly. Immediately the benefit of our method becomes clear: whereas symmetry in larger structures means more and more distance between links, the distance between two subsequent links is now constant. Especially for larger structures this is in favor of criterion 3 since the smallest bounding box will only increase linearly with the number of links.

The most important influence on the drawing is now the ordering of links in the structure's list of links. The algorithm for lexical unfolding evaluates links recursively, leading to an ordering that is not optimal for this method. This ordering can be thought of as a pre-order traversal. The order we use is more an inorder traversal, where the hypotheses are treated as the left subtree and the conclusions as the right subtree.

The link containing the main vertex is still used as a starting point and we'll call it the main link. When evaluating a link's premises, all new links created will be drawn above it, so they will be ordered before that link. Its conclusions will be ordered after it so to draw them below it. This ordering plays into the relative meaning of up and down in the structure meaning premise and conclusion respectively. To get this specific ordering we construe a method of inserting a link in the list of links. The main link is added as the sole element of the order list at start. Now for each link that is created, we give it its origin, which is the link that evaluated a vertex creating this link, and a boolean signifying hypothesis or conclusion. If the evaluated vertex was a conclusion of the origin link, the current link is created as hypothesis and vice versa. So a link that has True as its hypothesis value is a conclusion of the origin link. The ordering algorithm for all links other than the main link is now relatively simple.

We call this method for every link, giving it a boolean h and link O:

if h is True **then**
    insert self directly after O in order
else
    insert self directly before O in order
end if

Since the absolute index of links in this list can change due to links being inserted before them, we cannot immediately use the order list as the list of links. The list of links is first naively constructed and its contents are permuted according to the order list after unfolding. After ordering we can draw the links in the structure's list one by one below each other. We draw the first link at $(0, 0)$ and decrease y by 3 every time we draw a new link. The next logical step is drawing the vertices. We can draw the premises and conclusions of each link close to it, but in doing so we would draw internal nodes twice (once for each link containing them). We therefore only draw the first occurrence of a vertex, for which we need to remember which vertices we've already drawn.

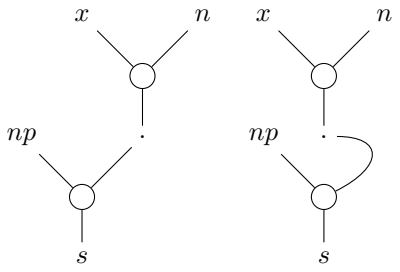


Figure 4: Shifting on the x-axis

All that is left is drawing connections between vertices and links. To minimize crossings we first minimize the length of lines by shifting links on the x-axis relative to the previous link. If it

is connected to this link we can minimize the length of the connecting line by aligning the point of connection, see Figure 4. Here the left structure has applied shifting so the shape of the lower tensor remains intact.

This shifting is accomplished by comparing each link to draw with the link we drew before it. The method, called *adjust_xy*, takes the previous and current link and compares them, returning a tuple (x, y) containing adjustments to the x and y coordinates. We adjust the y-coordinate as well because we can also make the drawing more compact by decreasing link distance in some cases. Adjustments are only made if the previous link is connected to the current link via one internal vertex, according to this table:

	top-left	top	top-right
bottom-left	(0,0)	(-1,1)	(-2,1)
bottom	(1,1)	(0,0)	(-1,1)
bottom-right	(2,1)	(1,1)	(0,0)

We now draw lines, straight ones between local links and vertices and curved ones in case link and vertex are separated by more than a link's length. For cotensors we draw an arrow pointing to their main formula. This can be a curved arrow if it's main formula is drawn higher up the structure.

A justification for using this many curved lines can be seen in Figure 3. Here curved lines make sure longer lines do not cross lines between a link and simple hypotheses or conclusions, making these immediately identifiable. The only crossing in fact will occur between curved lines.

We can now also justify the decision to draw all links below each other. When two structures are connected by identifying their formulas, the resulting structure needs no transformation to be drawn. Connecting two structures can result in new symmetries, which a method based on symmetry will try to show by adjusting the whole structure. Using our method connecting two structures is a simple operation of first placing the two structures beneath each other and then drawing a line between the two identified formulas. Of course, in this process one of the two vertices is destroyed. How to identify formulas and connect structures in a useful way is beyond the scope of this paper.

Summarizing, our method is described as:

1. Create the proof structure
2. Reorder the links
3. For each link, do:
   (a) Adjust x- and y-coordinates
   (b) Draw self at (x,y)
   (c) Draw all vertices in premises not already drawn
   (d) Draw all vertices in conclusions not already drawn
   (e) Adjust the y-coordinate
4. Draw all links and arrows

Python, LaTeX and TikZ

Appendices A through D give a program that implements the described method of drawing proof structures. This implementation is object-oriented and written in Python. The bulk of the work is done in `classes-linear.py`, containing all relevant objects. In `proofnet.py` the main method is called, unfolding the main vertex. Upon creation links and vertices unfold further so that the structure is built up as a result. Afterwards the structure can be printed to LaTeX using the TikZ package.

A quick look at `argparser.py` reveals all options in this program. Alternatively executing the `python proofnet.py --help` command does the same. We give an overview of all implemented options:

- `-t or --tex`: do not print the structure in LaTeX
- `-a or --abstract`: hide internal node decoration
- `-m or --main`: hide main formula as argument given
- `-r or --rotate`: orientate the structure lengthwise

- `-c or --conclusion`: conclusion unfolding instead of hypothesis unfolding

A sample call and its output in Figure 5.

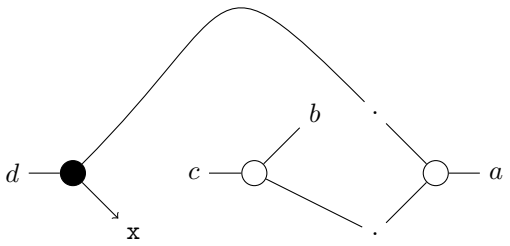


Figure 5

```
python proofnet.py -ram x (a(/)(b\c))(\)d
```

Discussion

The described algorithm can easily be implemented. It is formulated in such terms that expansion to use for more complicated structures or nets is easy. There are, however, some structures that could have been drawn better. Consider the structure of $\phi : a/(b \oslash (d\backslash c))$ in Figure 6.

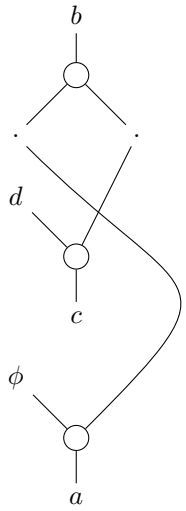


Figure 6

This structure has a crossing link that could have been avoided by shifting the top two links to the right. Generally when drawing a link we could derive its x-coordinate from its predecessors instead of the link immediately preceding it. We chose not to implement this fix, as a simpler structure (though not a direct result of unfolding) cannot use this fix. This structure is shown in Figure 7.

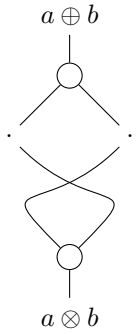


Figure 7

This structure corresponds to the sequent $a \oplus b \Rightarrow a \otimes b$ after identifying formulas. It is not derivable, but that is besides the point. Such a structure should be drawable in such a way that it is clear what it represents. With shifting this structure would only get worse. We think that any formal drawing rule that would make the structure in Figure 6 more readable would only distort Figure 7. Again, a solution might be a check whether the structure is a tree and only resorting to the described method if the structure is not.

Conclusion

The accompanying code fully automates lexical unfolding, both hypothesis and conclusion, and can print the resulting structure in a readable fashion. It might be improved upon by first testing whether the structure is a tree, in which case it may be drawn using a standard tree-drawing algorithm. This seems to be the most common alternative for drawing (simple) structures. Further work might include expanding this code to a prover for **LG**.

References

[1] Jean-Yves Girard. Linear Logic: its syntax and semantics. *Theoretical Computer Science*, 50:1-102, 1987.

[2] V.N. Grishin. On a generalization of the Ajdukiewicz-Lambek system. In A.I. Mikhailov, editor, *Studies in Nonclassical Logics and Formal Systems*, pages 315–334. Nauka, Moscow, 1983. [English translation in Abrusci and Casadio (eds.) Proceedings 5th Roma Workshop, Bulzoni Editore, Roma, 2002].

[3] Michael Moortgat. Symmetries in Natural Language Syntax and Semantics: The Lambek-Grishin Calculus. *Proceedings WoLLIC '07*, pg 264-284, 2007.

[4] Michael Moortgat and Richard Moot. Proof nets and the categorial flow of information. 2012.

[5] Richard Moot. Proof nets for display logic. *Technical report, CNRS and INRIA Futurs*, 2007.

[6] Richard Moot and Quintijn Puite. Proof nets for the multimodal Lambek calculus. *Studia Logica*, 71(3):415–442, 2002.

A: proofnet.py

```
#!/usr/bin/env python

# LIRa refers to:
# http://www.phil.uu.nl/~moortgat/lmnlp/2012/Docs/contributionLIRA.pdf
# Proofs nets and the categorial flow of information
# Michael Moortgat and Richard Moot

#BLEH
#A/(B(/)(D\C))
#STERKER NOG
#E/(A/(B(/)(D\C)))

from helper_functions import *
import classes_linear as classes
import argparser

import os, sys
import platform


# By default the formula appears in hypothesis position.
def unfold_formula(formula, hypothesis=1):
    vertex = classes.Vertex(formula, hypothesis)
    structure = classes.ProofStructure(formula, vertex)#, hypothesis)
    if not simple_formula(formula):
        vertex.unfold(formula, hypothesis, structure)     # Recursively unfold
    # Toggle whole formula
    p = argparser.Parser()
    args = p.get_arguments()
    if args.main:
        vertex.main ='|texttt{{{0}}}'.format(args.main)
    return structure

def main():
    p = argparser.Parser()
    args = p.get_arguments()
    if len(args.formula) != 1:
        p.print_help()
        sys.exit()
    formula = args.formula[0]
    hypothesis = not args.conclusion
    structure = unfold_formula(formula,hypothesis)
    structure.print_debug()  # for debugging
    if not args.tex:
        structure.toTeX()
        os.system('pdflatex formula.tex')
        if platform.system() == 'Windows':
            os.system('start formula.pdf')
        elif platform.system() == 'Linux':
            os.system('pdfopen --file formula.pdf')
        # Mac OS X ?

if __name__ == '__main__':
    main()
```

B: helper_functions.py

```
import re

vertices = 0
texlist = []

# This returns True if the formula contains no connectives.
def simple_formula(formula):
    connectives = re.compile(r"(\*|\\|/|\(\*\)|\(/\)|\(\\\))")
    search = connectives.search(formula)
    return search is None


# Flip the hypothesis boolean
def flip_hypo(hypothesis):
    return (hypothesis + 1) % 2

def type(connective, hypo):
    types = {
        # LIRa figure 3
        # (con,hypo):(premise#,geometry)
        # geometry: (f)ormula,(l)eft,(r)ight,(<)arrow to previous

        # Fusion connectives - hypothesis
        ("/",1):(2,"frl"),
        ("*",1):(1,"f<lr"),
        ("\\",1):(2,"lfr"),
        # Fusion connectives - conclusion
        ("/",0):(1,"lf<r"),
        ("*",0):(2,"lrf"),
        ("\\",0):(1,"rlf<"),
        # Fission connectives - hypothesis
        ("(/)",1):(2,"f<rl"),
        ("(*)",1):(1,"flr"),
        ("(\\)",1):(2,"lf<r"),
        # Fission connectives - conclusion
        ("(/)",0):(1,"lfr"),
        ("(*)",0):(2,"lrf<"),
        ("(\\)",0):(1,"rlf")
    }
    return types[(connective, hypo)]
```

C: argparser.py

```
import argparse
import textwrap


class Parser(object):

    def __init__(self):
        self.p = argparse.ArgumentParser(
            formatter_class=argparse.RawDescriptionHelpFormatter,
            description = textwrap.dedent('''\
        Lexical unfolding of proof structures for LG
        Formula language:
                A,B ::= p |                 atoms (use alphanum)
                A*B | B\A | A/B |           product
                A(*)B | A(/)B | A(\)B       coproduct

        To use LaTeX commands as atoms, use |.
        For example: |phi will be translated as \phi'''),
                                    usage = 'proofnet.py [options] formula')
        self.p.add_argument('formula', metavar='F', type=str, nargs='+',
                        help='a formula in LG to unfold')
        self.p.add_argument('--tex', '-t', action = 'store_true',
                    help = 'do not print result to LaTeX')
        self.p.add_argument('--abstract', '-a', action = 'store_true',
                    help = 'hide internal node decoration')
        self.p.add_argument('--main', '-m',
                    help = 'hide main formula as argument given')
        self.p.add_argument('--rotate', '-r', action = 'store_true',
                    help = 'rotate structure 90 degrees counterclockwise')
        self.p.add_argument('--conclusion', '-c', action = 'store_true',
                    help = 'conclusion unfolding instead of hypothesis unfolding')
        self.arguments = self.p.parse_args()

    def get_arguments(self):
        return self.arguments
```

D: classes_linear.py

```
from helper_functions import *
import argparser
import sys

order = [0]
drawn = []

class ProofStructure(object):

    def __init__(self, formula, vertex):
        self.formula = formula
        self.main = vertex
        self.links = []

    def print_debug(self):
        print order
        for t in self.links:
            print t.index
```

```
def add_link(self, link):
    self.links.append(link)
    link.index = len(self.links) - 1
    link.alpha = 't' + chr(len(self.links) + 96)

def toTeX(self):
    # Erase file
    f = open('formula.tex', 'w')
    f.close()

    # Write to formula.tex
    # Header
    f = open('formula.tex', 'a')

    f.write('\documentclass[class=minimal,border=0pt]{standalone}\n\n')
    f.write('\usepackage{tikz-qtree}\n')
    f.write('\usepackage{stmaryrd}\n\n')
    f.write('\\begin{document}\n\n')

    # Tikzpicture

    rotate = ""
    # Toggle rotation
    p = argparser.Parser()
    args = p.get_arguments()
    if args.rotate:
        rotate = "rotate=270,"

    f.write('\\begin{tikzpicture}[')
    f.write(rotate)
    f.write('scale=.8,')
    f.write('cotensor/.style={minimum size=2pt,fill,draw,circle},\n')
    f.write('tensor/.style={minimum size=2pt,fill=none,draw,circle},')
    f.write('sibling distance=1.5cm,level distance=1cm,auto]\n\n')

    x = 0
    y = 0

    if not self.links:
        f.write(self.main.toTeX(x, y, self.main))

    else:
        # Shuffle self.links according to order
        self.links = map(lambda x: self.links[x],order)
        previous_link = None

    for link in self.links:

        if previous_link is not None:
            (x_adj,y_adj) = adjust_xy(previous_link, link)
            x += x_adj
            y += y_adj

        f.write('{0} at ({1},{2}) {{}};\n'.format(link.toTeX(),x,y))
        f.write(link.hypotheses_to_TeX(x, y))
        f.write(link.conclusions_to_TeX(x, y))
        y -= 3
        previous_link = link

    for line in texlist:
        f.write(line)
```

```
        f.write('\n\end{tikzpicture}\n\n')

        # End of document
        f.write('\end{document}')
        f.close()


def adjust_xy(previous, current):
    if isinstance(previous, OnePremise):
        if previous.bottomLeft.conclusion is current:
            if isinstance(current, OnePremise):
                if current.top.hypothesis is previous:
                    return (-1,1)
            else:
                if current.topRight.hypothesis is previous:
                    return (-2,1)
        elif previous.bottomRight.conclusion is current:
            if isinstance(current, OnePremise):
                if current.top.hypothesis is previous:
                    return (1,1)
            else:
                if current.topLeft.hypothesis is previous:
                    return (2,1)
    else:
        if previous.bottom.conclusion is current:
            if isinstance(current, TwoPremise):
                if current.topLeft.hypothesis is previous:
                    return (1,1)
                elif current.topRight.hypothesis is previous:
                    return (-1,1)
    return (0,0)


class Vertex(object):

    def __init__(self, formula=None, hypo=None):
        global vertices
        self.set_hypothesis(None)
        self.set_conclusion(None)
        self.alpha = chr(vertices + 97)
        vertices += 1
        if formula is not None:
            self.main = formula
            self.attach(formula, hypo)

    def set_hypothesis(self, hypo):
        self.hypothesis = hypo

    def set_conclusion(self, con):
        self.conclusion = con

    def toTeX(self, x, y, link):
        global texlist, drawn
        co = ""
        if link.is_cotensor() and link.arrow is self.alpha:
            co = "[->]"
        line = "\draw{0} ({1}) -- ({2});\n".format(co,link.alpha,self.alpha)
        if self.internal() and self.conclusion is link:
            if order.index(link.index) != order.index(self.hypothesis.index) + 1:
                line = self.curved_tentacle(link, self.hypothesis)
        texlist.append(line)
        if self.alpha in drawn:
```

```
            return ""
        drawn.append(self.alpha)
        label = operators_to_TeX(self.main)
        tex = "\\node ({0}) at ({1},{2}) {{${3}$}};\n".format(self.alpha,
                x, y, label)
        return tex

    def curved_tentacle(self, link, prev_link):
        co = ""
        if link.is_cotensor() and link.arrow is self.alpha:
            co = "[->]"
        start = "\draw{0} ({1}) .. controls ".format(co, link.alpha)
        direction = "west"
        if isinstance(link, TwoPremise):
            if link.topRight is self:
                direction = "east"
        elif isinstance(prev_link, OnePremise):
            if prev_link.bottomRight is self:
                direction = "east"
        controls = "+(north {0}:4) and +(south {0}:4.0)".format(direction)
        end = " .. ({0});\n".format(self.alpha)
        line = start + controls + end
        return line

    def internal(self):
        return (isinstance(self.hypothesis, Link) and
                    isinstance(self.conclusion, Link))

    def attach(self, label, hypo):
        if hypo:
            self.set_hypothesis(label)
        else:
            self.set_conclusion(label)

    # This is the source of the recursion
    def unfold(self, formula, hypo, structure, i=None):
        regexp = re.compile(
        r"""(\(.+\)|[\w'{}|$]+)                    #left formula
            (\*|\\|/|\(\*\)|\(/\)|\(\\\))          #main connective
            (\(.+\)|[\w'{}|$]+)$                   #right formula
        """, re.X)
        search = regexp.match(formula)
        try:
            (left, connective, right) = search.groups()
        except AttributeError:
            print "Syntax error in formula"
            sys.exit()
        (h, geometry) = type(connective,hypo)
        if h == 1:
            t = (OnePremise(left, right, geometry, self, structure, hypo, i))
        else:
            t = (TwoPremise(left, right, geometry, self, structure, hypo, i))


class Link(object):

    def __init__(self):
        print "error"

    def toTeX(self):
        co = ''
        if self.is_cotensor():
```

```
            co = 'co'
        return '\\node [{0}tensor] ({1})'.format(co,self.alpha)

    def parse_geometry(self, geometry, vertex):
        index = geometry.find("<")
        if index > -1:
            self.arrow = vertex.alpha
        geometry = geometry.replace("<", "")
        return geometry

    def get_lookup(self, left, right, vertex):
        lookup = {
            'f':(Link.attach,vertex),
            'l':(Link.eval_formula,left),
            'r':(Link.eval_formula,right)
        }
        return lookup

    def set_structure(self, struc, hypo, origin_index):
        global order
        if origin_index is not None:
            new = len(order)
            origin_index = order.index(origin_index)
            if hypo:
                order.insert(origin_index + 1,new)
            else:
                order.insert(origin_index,new)
        struc.add_link(self)
        self.structure = struc

    def is_cotensor(self):
        return hasattr(self, 'arrow')

    def attach(self, vertex, hypo):
        flipped_hypo = flip_hypo(hypo)
        vertex.attach(self, flipped_hypo)
        return vertex

    def eval_formula(self, part, hypo):
        if simple_formula(part):
            return self.attach(Vertex(part, hypo), hypo)
        else:
            vertex = Vertex()
            self.attach(vertex, hypo)
            part = part[1:-1]
            flipped_hypo = flip_hypo(hypo)
            vertex.unfold(part, flipped_hypo, self.structure, self.index)
            # Toggle abstract
            p = argparser.Parser()
            args = p.get_arguments()
            if args.abstract:
                vertex.main = "."
            else:
                vertex.main = part
            return vertex


class OnePremise(Link):

    def __init__(self, left, right, geometry, vertex, struc, hypo, i):
        Link.set_structure(self, struc, hypo, i)
        geometry = Link.parse_geometry(self, geometry, vertex)
```

```
        lookup = Link.get_lookup(self, left, right, vertex)
        (function,arg) = lookup[geometry[0]]
        self.top = function(self, arg, 1)
        (function,arg) = lookup[geometry[1]]
        self.bottomLeft = function(self, arg, 0)
        (function,arg) = lookup[geometry[2]]
        self.bottomRight = function(self, arg, 0)

    def get_hypotheses(self):
        return [self.top]

    def get_conclusions(self):
        return [self.bottomLeft, self.bottomRight]

    def num_hyp(self):
        return 1

    def num_con(self):
        return 2

    def hypotheses_to_TeX(self, x, y):
        return self.top.toTeX(x, y + 1, self)

    def conclusions_to_TeX(self, x, y):
        s1 = self.bottomLeft.toTeX(x - 1, y - 1, self)
        s2 = self.bottomRight.toTeX(x + 1, y - 1, self)
        return s1 + s2


class TwoPremise(Link):

    def __init__(self, left, right, geometry, vertex, struc, hypo, i):
        Link.set_structure(self, struc, hypo, i)
        geometry = Link.parse_geometry(self, geometry, vertex)
        lookup = Link.get_lookup(self, left, right, vertex)
        (function,arg) = lookup[geometry[0]]
        self.topLeft = function(self, arg, 1)
        (function,arg) = lookup[geometry[1]]
        self.topRight = function(self, arg, 1)
        (function,arg) = lookup[geometry[2]]
        self.bottom = function(self, arg, 0)

    def get_hypotheses(self):
        return [self.topLeft, self.topRight]

    def get_conclusions(self):
        return [self.bottom]

    def num_hyp(self):
        return 2

    def num_con(self):
        return 1

    def hypotheses_to_TeX(self, x, y):
        s1 = self.topLeft.toTeX(x - 1, y + 1, self)
        s2 = self.topRight.toTeX(x + 1, y + 1, self)
        return s1 + s2

    def conclusions_to_TeX(self, x, y):
        return self.bottom.toTeX(x, y - 1, self)
```

```
def operators_to_TeX(string):
    string = string.replace("\\", "\\backslash ")
    string = string.replace("(*)", "\oplus ")
    string = string.replace("*", "\otimes ")
    string = string.replace("(/)", "\oslash ")
    string = string.replace("(\\backslash )", "\obslash ")
    string = string.replace("|", "\\")
    return string
```