Laws and Logic

Bachelor Project

Bachelor in Software Development,
IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk
Carsten Schürmann, carsten@itu.dk

May 22nd, 20130

Contents

1 Introduction

This paper is the result of a bachelor project on the bachelor line in Software Development at the IT-University of Copenhagen spanning from the February 2nd, 2013 to May 22nd, 2013.

1.1 Background

When it comes to elections, there are laws describing how votes are to be distributed. These laws are often short and to the point, which results in both advantages and disadvantages. One advantage is that they are not difficult to read. A disadvantage is that there are situations where they are not informative enough, especially when it comes to writing computer programs to assist in the distribution.

When people write code following the legal text, they have to make assumptions about certain things that are not fully described. If a person has to make assumptions, how can others then trust the end result? They cannot and that is a problem. Even if the program is correct, it is not easy to certify that the code meets the legal specifications.

One way to get around these issues is to introduce something that can describe the legal specifications fully, while still being easy to convert to code. This is where linear logic enters the picture. Linear logic is a type of formal logic well-suited to write trustworthy specifications and implementations of voting protocols. **Note: rewrite**

Translating legal text into linear logic results in logical formulas. These formulas are basically algorithms at a high level of abstraction, which makes it easy to translate into code and can actually be used as source code. Logical formulas are also well-suited for proving the correctness of what they describe and can thus bridge the gap between the legal text and code.

1.2 Goal of the project

Another possibility is to change the legal text based on the logical formulas. This way, the logical formulas will only have to be written once, after which the legal text will be easier to translate into code (and understand in general).

The goal of this project is to explore the possibility of writing a program in Grammatical Framework[1] to translate linear logic formulas into understandable sentences in one or more natural languages, for the purpose of using them as law texts.

[1] http://www.grammaticalframework.org/

2 Overview

2.1 Example

As described in section 1.1, the legal text is short, to the point and not that good for translating into source code. An example of the legal text can be seen here:

> *"If a candidate reaches the quota, he is declared elected."*

This piece of text shows the lack of detail. How do we ensure the candidate has reached the quota? What do we do with the ballot that makes him reach the quota? What happens with him when he is elected? And most importantly, can he be elected even if there are no open seats? The latter is a very important question, one that is not answered by the legal text. Common sense dictates that the answer is "no", but that is an assumption one has to make and assumptions are best avoided when it comes to voting.

Note: Make sure first sentence is actually correct when it comes to facts
According to the **Note: insert reference to deyoung-schuuermann-voteid2011**, translating the legal text into linear logic results in this formula (how linear logic works is explained in section 3.1):

$$
\begin{aligned}
&\textit{tally-votes(S, H, U)} \otimes \\
&\textit{uncounted-ballot(C, L)} \otimes \\
&\textit{hopeful(C, N)} \otimes \\
&!\textit{quota(Q)} \otimes !\textit{(N + 1 < Q)} \otimes \\
&!\textit{(S} \geq \textit{1)} \\
&\quad \multimap \{\ \textit{counted-ballot(C, L)} \otimes \\
&\qquad\quad !\textit{elected(C)} \otimes \\
&\qquad\quad \textit{tally-votes(S-1, H-1, U-1)}\ \}
\end{aligned}
$$

This formula accurately describes the entire process involved in checking if a candidate reaches the quota and then marking him as elected. Understanding this formula requires knowledge of how linear logic works (see section 3.1) and is definitely not suited for use as legal text. DeYoung and Schürmann has taken this into account, however, and have come up with a formalized version of it:

> If we are tallying votes and
> there is an uncounted vote for C and
> C is a hopeful with running tally N and
> this vote would meet the quota and
> there is at least one seat left,
> then mark the ballot as counted and
> declare candidate C to be elected and
> tally the remaining U-1 ballots among the H-1 hopefuls and S-1 seats left.

Each line in the formalized version corresponds to a line of the logical formula and describes the process perfectly. Nothing has been left to assumptions. The formalized version has been written manually and makes it easy to test the program, as we now have something to aim for.

3 Technologies

3.1 Linear Logic

Linear logic is a type of logic where truth is not free, but a consumable resource. In traditional logic any logical assumption may be used an unlimited number of times, but in Linear Logic each assumption is "consumed" upon use.

Because the resources are consumable, they may not be duplicated and can thereby only be used once. This makes the resources valuable and also means that they cannot be disposed of freely and therefore must be used once. With this, Linear Logic can be used to describe things/operations(?) that must occur only once. This is important, as voting protocols rely on things being able to occur only once (each voter can only be registered once, each ballot may only be counted once, etc.).

3.1.1 Connectives

Traditional Logic contains connectives that, unfortunately, are not specific enough for the purpose of these formulas. The implies $\rightarrow$ and the logical conjunction $\wedge$ do not deal with resources. A $\rightarrow$ B means that if A is true then B is true. It says nothing about A or B being consumed. The same goes for $\wedge$. Another notation is therefore needed.

As **Linear Logic** is based around the idea of resources, the connectives reflect that. Linear Logic has a lot of connectives that can be used to express logical formulas, but the logical formulas studied in the project are only concerned with some of them. They are Linear Implication, Simultaneous Conjunction, Unrestricted Modality and the Universal Quantification.

Linear Implication, $\multimap$. Linear implication is linear logic's version of $\rightarrow$. $\multimap$ consumes the resources on the left side to produce the resources on the right side. The logical formula

$$\textit{voting-auth-card} \multimap \{\ \textit{blank-ballot}\ \}$$

therefore consumes a voter's authorization card and gives a blank ballot to the voter in exchange. The idea behind the { and } are explained at the end of section 3.1.2.

Simultaneous Conjunction, $\otimes$. Simultaneous conjunction is linear logic's version of the $\wedge$. A $\wedge$ B means "if A and B" and does not take the resources into account. It is simply concerned whether A and B are true and/or false. A $\otimes$ B means "if resource A and recourse B are given" and thereby fulfills the criteria of working with resources.The logical formula

$$\textit{voting-auth-card} \otimes \textit{photo-id} \multimap \{\ \textit{blank-ballot}\ \}$$

will consume a voter's authorization card and photo ID and give a blank ballot to the voter in exchange. It should be noted that $\otimes$ binds more tightly than $\multimap$. There is a special unit for simultaneous conjunction, **1** (meaning "nothing"). **1** represents an empty collection of resources and is mainly used when some resources are consumed but nothing is produced.

Unrestricted Modality, !. The unrestricted modality is unique to linear logic. In the formula *voting-auth-card* $\otimes$ *photo-id* $\multimap$ { *blank-ballot* }, the photo ID of the voter is consumed, which means the voter must give up their photo ID to vote. This is a lot of lost passports/driver's licenses! The unrestricted modality, !, solves that problem. !A is a version of A that is not consumed and can be used an unlimited number of times (even no times at all). Using the unrestricted modality, the logical formula

$$\textit{voting-auth-card} \otimes !\textit{photo-id} \multimap \{\ \textit{blank-ballot}\ \}$$

now consumes only the authorization card and checks the photo ID (without consuming it) before giving the voter a blank ballot in exchange.

Universal Quantification, $\forall$x:r. The universal quantification is found both in traditional logic and linear logic and is necessary to complete the formula. As it is now, one simply needs to give an authorization card and show a photo ID. The name on the authorization card does not have to match the one on the photo ID. In linear logic, the universal quantification works the same was as in traditional logic and it says that "all x belongs to r". Using the universal quantification, the logical formula is changed to

$$\forall \textit{v:voter.}\ (\textit{voting-auth-card(v)} \otimes !\textit{photo-id(v)} \multimap \{\ \textit{blank-ballot}\ \})$$

now requires voter v to give *his* authorization card and show *his own* photo ID before *he* can receive a blank ballot.

Adding the connectives together gives the following.

$$A,\ B ::= P \mid A \multimap B \mid A \otimes B \mid !A \mid \forall x{:}r.\ A \mid \mathbf{1}$$

3.1.2 Splitting the connectives

The types at the end of section 3.1.1 immediately pose a problem. As each connective gives an A, that A can be used in another connective. In theory, one could have a !!!*voting-auth-card*, which does not make sense. They need to be split up to prevent this from happening.

Each connective its own derivation and they determine how the connectives are split up. Each derivation has a right and a left "side". If the side can be "reversed" (ie. the top and bottom part can be switched and it is still correct), the side is said to be inversible(?). This can only hold true for either the right or the left side, thus labeling the the derivation either "left inversible" or "right inversible"(?).
This right/left inversability(?) is what will be used to split the types up. The right inversible types will be grouped as "negative" types and the left inversible will be grouped as "positive" types.

To use an example[1], the derivation of the simultaneous conjunction is the following:

$$\text{Left}\ \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \qquad\qquad \frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \text{Right}$$

One will notice two new symbols in these derivations (Δ and $\vdash$) that require some explanation. Δ is a symbol representing some sort of resource. In essence, $\Delta \multimap A$ would mean "some resource produces A". The $\vdash$ symbol has a meaning that is a bit like the $\multimap$ symbol. $\vdash$ means "the resource on the right-hand side can be produced by using the resource(s) on the left-hand side **exactly once**". Using this definition, $\Delta_1 \vdash A$ means that A can be produces by using the resoruces of Δ exactly once.

Before we can determine if it is right or left inversible, we need to remember these two rules:

[1] **Note: Find the right slides from http://www.cs.cmu.edu/ fp/courses/15816-s12/schedule.html**

1. $\frac{}{A \vdash A}$
2. $\frac{\Delta_1 \vdash A \qquad \Delta_2 A \vdash C}{\Delta_1, \Delta_2 \vdash C}$

Number 1 says "A can be produced from A". Very straightforward. Number 2 says "if A can be produced from Δ_1 and C can be produced from Δ_2 and A, then C can be produced from Δ_1 and Δ_2 together". Now let us look at the two sides of the simultaneous conjunction, starting with the right side:

Original $\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}$ $\qquad\qquad$ $\frac{\Delta_1, \Delta_2 \vdash A \otimes B}{\Delta_1 \vdash A \quad \Delta_2 \vdash B}$ Reversed

The original derivation says "A can be produced from Δ_1 and B can be produced from Δ_2, so therefore A⊗B can be produced from $\Delta_1\Delta_2$". The reverse derivation says "A⊗B can be produced from $\Delta_1\Delta_2$, so therefore A can be produced from Δ_1 and B can be produced from Δ_2". That is not correct, however, as B might actually be produced from Δ_1. As the right side is not inversible, the left must be. Let us check.

Original $\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}$ $\qquad\qquad$ $\frac{\Delta, A \otimes B \vdash C}{\Delta, A, B \vdash C}$ Reversed

The original derivation says "C can be produced from Δ, A and B, so therefore C can also be produced from Δ, A⊗B". The reversed side says "C can be produced from Δ, A⊗B, so therefore C can also be produced from Δ, A and B". As the right side of the derivation says $\Delta_1, \Delta_2 \vdash A \otimes B$, it also means that $A \otimes B \vdash \Delta_1, \Delta_2$ as it goes both ways. Therefore, $A, B \vdash A \otimes B$ is the same as $A \otimes B \vdash A, B$. The left side is therefore inversible, making the simultaneous conjunction a "positive" type.

Doing the same for the rest of the types results in the following positive and negative types

Negative Types $A^- ::= P^- \mid \forall x : A^+. \mid A^+ \multimap A^- \mid \{A^+\}$
Positive Types $A^+ ::= A^+ \otimes A^+ \mid \mathbf{1} \mid !A^- \mid A^-$

Note: Is it $\{A^-\}$ or $\{A^+\}$? Also needs explanation.
One may notice that a "new" type ($\{A^+\}$) has been added here.
With the types split up like this, we have eliminated the cases that were not supposed to happen. It will also help when it comes to writing the program, as Grammatical Framework is built up around types.

3.1.3 Celf

The negative and positive types are all well and good, but while humans can read and understand signs like ⊗, ⊸ and ∀, they are not easy to represent in some computer programs (for example in Grammatical Framework). To get around this issue, the Celf framework is used.

> *"CLF (Concurrent LF) is a logical framework for specifying and implementing deductive and concurrent systems from areas, such as programming language theory, security protocol analysis, process algebras, and logics. Celf is an implementation of the CLF type theory that extends the LF type theory by linear types to support representation of state and a monad to support representation of concurrency."* [1]

Celf has its own syntax for the connectives of linear logic, which will be used as the standard for the GF program. The Celf syntax for the connectives are the following

⊗	::= *
⊸	::= -o
∀x:r	::= Pi x : r

The connectives are not the only things in linear logic. It is also possible to use arithmetic operations in formulas written with linear logic (see chapter 2.1 for an example where arithmetic operations are included). Celf uses a special syntax for the arithmetic operations as well, which are described below.

x > y	::= !nat-greater x y
x >= y	::= !nat-greatereq x y
x = y	::= !nat-eq x y
x <= y	::= !nat-lesseq x y
x < y	::= !nat-less x y
x − 1	::= (s ! x)
x + 1	::= (p ! x)
[x | y]	::= (cons ! x ! y)

Using the Celf syntax, we can translate a formula written in linear logic into something a computer can read more easily. The example from section 2.1 therefore becomes:

```
tally-votes S H U *
uncounted-ballot C L *
hopeful C N *
!quota Q * !nat-lesseq Q (p ! N)
  -o { counted-ballot C L *
       !elected C *
       tally-votes (s ! S) (s ! H) (s ! U) }
```

Now we have an understanding of how linear logic works and how to represent it in an easy way on the computer.
Note: Other things that needs to be said about Celf?
Note: Other ending?

3.2 Grammatical Framework

Note: Some sort of transition from linear logic to this section is needed here. End it with "throughout this section, we will construct a very simple program for interpreting LL."

Grammatical Framework (GF) is an open-source multilingual programming language. With GF, one can write programs that can translate other languages. This works through parsing (analyzing a language), linearization (generating the language) and translation (analyzing one language to generate another one). In this section, I will go over the aspects of GF that have been used in writing the program.

A GF program consists of an abstract module and one or more concrete modules. The abstract module defines what meanings can be interpreted and parsed by the grammar. The concrete module maps the abstract meanings to strings, thereby forming complete sentences.

3.2.1 The abstract module

The abstract module contains category declarations (`cat`) and function declarations (`fun`). The `cat` list the different categories (meanings) used in the language, where the `fun` dictates how the categories fit together to create meaning-building functions. The abstract syntax furthermore has a `flag startcat` that indicates what category the program should start with.

GF

```
abstract AbstractLinearLogic = {
    flags startcat = Formula ;

    cat
        Formula ; Connective ; Ident ;
    fun
        _VoteCard, _BlankBallot : Ident ;
        _Lolli : Connective ;
        _Formula : Ident -> Connective -> Ident -> Formula;
}
```

Code 3.1: A simple abstract syntax.

GF

In the abstract syntax shown in Code 3.1, there are three categories: `Formula`, `Connective` and `Ident`. Furthermore, there are three functions. The first function says that `_VoteCard` and `_BlankBallot` are of the type `Ident`. The second function says that `_Lolli` is of the type `Connective`. The last function says that `_Formula` takes three arguments (an `Ident`, a `Connective` and an `Ident`) and returns something of the type `Formula`.

3.2.2 The concrete module

The concrete module contains linearization type definitions (`lincat`) and linearization definitions (`lin`). The `lincat` determines the type of object for each category in the abstract syntax and the `lin` determines what value is assigned to each abstract meaning.

When the program parses a language, it will look for the values being held by the meanings and translate each into the abstract syntax. This abstract syntax forms an abstract syntax tree. The program can then turn the abstract syntax tree into any language supported by concrete implementations.

GF

```
concrete ConcreteLinearLogic of AbstractLinearLogic = {
    lincat
        Formula, Connective, Ident = {s = Str} ;
```

```
4     lin
5         _VoteCard = {s = "voting-auth-card"} ;
6         _BlankBallot = {s = "blank-ballot"} ;
7         _Lolli = {s = "-o"} ;
8         _Formula i1 c i2 = {s = i1.s ++ c.s ++ "{" ++ i2.s ++ "}"} ;
9 }
```

Code 3.2: A concrete implementation of the abstract syntax from Code 3.1 that understands linear logic.

GF

In Code 3.2 `Formula`, `Connective` and `Ident` have been defined as records that can hold a `Str` (a string). `_VoteCard`, `_BlankBallot` and `_Lolli` corresponds to a certain string. `_Formula` has a more advanced meaning, however, as it consists of the value of `i1` concatenated with the value of `c` and `i2` inside curly brackets. `i1`, `c` and `i2` are the arguments it takes according to the abstract syntax.

GF

```
1 concrete ConcreteEnglish of AbstractLinearLogic = {
2     lincat
3         Formula, Connective, Ident = {s = Str} ;
4     lin
5         _VoteCard = {s = "an authorization card"} ;
6         _BlankBallot = {s = "a blank ballot"} ;
7         _Lolli = {s = "then"} ;
8         _Formula i1 c i2 = {s = "if i give" ++ i1.s ++ c.s ++ "i get" ++ i2.s} ;
9 }
```

Code 3.3: A concrete implementation of the abstract syntax from Code 3.1 that understands English.

GF

With another concrete implementation (such as the English one in Code 3.3), the program will be able to translate sentences from one language into the other, as long as the sentences adhere to the structure set by the abstract syntax. Or in this case, translate linear logic into English or the other way around.

Using the modules above, one would be able to translate ”voting-auth-card -o { blank-ballot }” into ”if i give an authorization card then i get a blank ballot”.

3.2.3 Operations

Some thing may happen a lot of times in a concrete implementation (such as concatenating two strings). GF can make this easier through operations (`oper`), also known as functions in other programming languages.

Operations can do two things. They can define a new type and they can be used with arguments to produce something. The latter type of operation consists of the following:

- **A name** that defines the `oper` and is used when calling it. The operation in Code 3.4 has the name ”cc” (concatenate).
- **Arguments, their types and the return type.** The operation in Code 3.4 takes two arguments of the type Str called ”x” and ”y” and returns something of the type Str.
- **The actual operation**. The operation in Code 3.4 concatenates the two given strings and returns the result.

GF

```
1 oper
```

```
2     cc : Str -> Str -> Str = \x,y -> (x.s ++ y.s) ;
```

Code 3.4: A simple operation in GF that concatenates two strings.

GF

An operation can be placed in a concrete implementation (if it is only needed in one of them) or in a so-called `resource` module (see Code 3.5), which can be accessed by multiple concrete implementations. To access the resource module, one adds "open <nameOfModule> in" to the first line of a concrete implementation, as shown in 3.6

GF

```
1 resource StringOper = {
2     oper
3         SS : Type = {s : Str} ;
4         ss : Str -> SS = \x -> {s = x} ;
5         cc : SS -> SS -> SS = \x,y -> ss (x.s ++ y.s) ;
6         prefix : Str -> SS -> SS = \p,x -> ss (p ++ x.s) ;
7 }
```

Code 3.5: A simple resource module.

GF

GF

```
1 concrete HelloEng of AbstractLinearLogic = open StringOper in {
2     lincat
3         Formula, Connective, Ident = SS ;
4     lin
5         _VoteCard = ss ("voting-auth-card") ;
6         _BlankBallot = ss ("blank-ballot") ;
7         _Lolli = ss ("-o") ;
8         _Formula i1 c i2 = ss (i1.s ++ c.s ++ "{" ++ i2.s ++ "}") ;
9 }
```

Code 3.6: Using the resource module.

GF

3.2.4 Dependent types and variable bindings

As there are variables used in the logical formula, we need a way to represent that in the program. GF supports this through variable bindings, which binds variables to allow them to be used in parts of the program. In the universal quantification `( Pi x : nat ) ( x = x )`, the variable `x` has a binding (`( Pi x : nat )`) that says it is of the type `nat` (a natural number), and that it is bound in the body `B(x)`, where it can be used.

To use variable bindings in GF, it is necessary to use functions as arguments. In Code 3.7, `Pi` takes an argument `A` that is required to be of the type `Set`. It also takes an `El A` argument, a type that takes a `Set` as argument (in this case `A`), and uses that in a `Set`. These two groups of arguments are used to create a `Set`. Furthermore, `Eq` also takes an argument `A` that is of the type `Set`. It also takes two other arguments, `a` and `b`, which both must be of the type `El` with `A` as its argument. This is turned into a `Set`.

GF

```
1     cat
2         Set ;
3         El Set ;
4     fun
5         Pi : (A : Set) -> (El A -> Set) -> Set ;
```

```
6       Eq : (A : Set) -> (a,b : El A) -> Set ;
7       Nat : Set ;
```

Code 3.7: Abstract syntax for variable bindings.

GF

The concrete syntax needs to use a special syntax for the variable bindings as well. Previously, we have used `B.s` to return the string value of the argument. With variable bindings the syntax is expanded, as shown in Code 3.8, with the addition of `B.$0` while `B.s` is still used.

GF

```
1   lincat
2       Set, El = SS ;
3   lin
4       Pi A B = ss ( "(" ++ "Pi" ++ B.$0 ++ ":" ++ A.s ++ ")" ++ B.s ) ;
5       Eq A a b = ss ( "(" ++ a.s ++ "=" ++ b.s ++ ")" ) ;
6       Nat = ss ( "nat" ) ;
```

Code 3.8: The concrete syntax for variable bindings.

GF

A normal argument would have the type `s : Str`, but a function argument has the type `s : Str ; $0 : Str`. It means that the argument `arg.$0` is bound and can be used in `arg.s`. If the function had more arguments `.$1`, `.$2`, etc. would have been used to refer to them.

In Code 3.8, `( Pi x : nat )` will bind the variable `x` (`B.$0`) as the type of `A`, and use it in `B.s`. `B.s` could be `Eq`, which would end up being `( x = x )`. At this point, one might wonder why the `A` argument is not used in `Eq`. The `A` argument shows the type of `a` and `b`, but as we already do that with `Pi`, there is no reason to write it next to the arguments.

The use of variable bindings allows for a more fexible program, and is important when working with logical formulas where it is never certain how many variables are going to be used, and what they are going to represent.

4 Implementation

Note: Write an introduction

The program has been written specifically for the formulas the voting protocols.

4.1 Constructing the Abstract Syntax

The first step in writing the program was to construct the abstract syntax. The abstract syntax determines how the parts of the "language" (linear logic in this case) are put together to form sentences. It is important to make sure the abstract syntax is well constructed, or strange things may happen in the program.

In this section, the different parts of the abstract syntax will explained individually, but the full abstract implementation can be found in appendix A if the reader wants to look at it.

The fact that we split the connectives of linear logic up in section 3.1.2 will help immensely. The resulting types can be translated almost directly into an abstract tree, which means we have a basic syntax already.

GF

```
1  abstract Laws = {
2
3      flags startcat = Logic ;
4
5      cat
6          Logic ; Neg ; Pos ; Lolli ; Bang ; Atomic ; Conj ; ArgType ; Argument ArgType ;
7
8      fun
9          Formular : Neg -> Logic ;
10
11         -- Positive types
12         _Atom : Atomic -> Pos ;                      -- Turning an atomic into a positive type
13         _Bang : Bang -> Atomic -> Pos ;              -- Using the unrestricted modality
14         _Conj : Pos -> Conj -> Pos -> Pos ;          -- Using the simultaneous conjunction
15         _Unit : Neg -> Pos ;                         -- Turns a negative into a positve
16         _MPos : Pos -> Pos -> Pos ;                  -- Attaches multiple positives to each other
17
18         -- Negative types
19         _Pi : (A : ArgType) -> (Argument A -> Neg) -> Neg ; -- The universal quantification
20         _Lolli : Pos -> Lolli -> Neg -> Neg ;        -- Using the linear implication
21         _Mon : Pos -> Neg ;                          -- Turning a positive into a negative
22
23         -- Connectives
24         _Conj2 : Conj ;                              -- Simultaneous conjunction
25         _Lolli2 : Lolli ;                            -- Linear implication
26         _Bang2 : Bang ;                              -- Unrestricted modality
27
28         -- Argument types
29         _Nat, _Candidate, _List : ArgType ;
30 }
```

Code 4.1: The first abstract syntax.

GF

Going through the abstract syntax, one will notice a couple of things that do not come from the connectives: There is something called an "`Atomic`" (line 12) and there is both an `ArgType` and an `Argument A` in the

function of _Pi. The Atomic is used to represent functions along with the arguments they take. They will be explained in more detail later in this section.

The _Pi should seem familiar, as it was used for demonstrating variable bindings in section 3.2.4. The only things that have changed are the types involved. ArgType is the type of the argument (a natural number, a candidate or a list), and Argument A uses this ArgType to bind a variable for use in the Neg. This will allow the use of variables in the program.

There are two other things that needs an explanation as well. The _Unit turns a negative type into a positive type (it is actually a positive type, so there is no cheating here). The _MPos allows for multiple positive types to follow each other without connectives and is mainly used for allowing multiple universan quantifiers to follow each other.

The abstract syntax in Code 4.1 introduced the Atomic. The Atomic needs an explanation and it needs to be defined as a function before the rest of the syntax can be written. An Atomic represents a function in linear logic. This could be the voting-auth-card or the blank-ballot mentioned in section 3.1.1. An Atomic can also represent a function that takes parameters, such as the ones in the example in section 2.1.

Looking at the example in section 2.1, one will notice that the formula not only contains functions, but that it also contains arithmetic and inequality operations (for example $!(N + 1 < Q)$). This is in place of a function, so the Atomic needs to be able to represent that as well.

Putting that together, it means we need two kinds of atomics, one for functions and one for mathematical operations. The Atomic is therefore represented in the following way in the abstract syntax.

GF

```
cat
    Atomic ; Ident ; MathFormula ;

fun
    -- Atomic
    Atom_Ident : Ident -> Atomic ;          -- Represents the atomic variables/functions
    Atom_Math : MathFormula -> Atomic ;     -- Represents the mathematical operations
```

Code 4.2: Defining the Atomic *in the abstract syntax.*

GF

With Atomic defined, we have introduced two new categories at the same time; Ident and MathFormula. The Ident represents the variables through the function name and the arguments it takes. To be able to use it, we need to define not just the Ident, but also the arguments needed. The abstract syntax is once again extended.

GF

```
cat
    Ident ; Arg ;

fun
    -- Identifiers
    Ident_Uncounted : Arg -> Arg -> Ident ;
    Ident_Counted: Arg -> Arg -> Ident ;
    Ident_Hopeful : Arg -> Arg -> Ident ;
    Ident_Defeated : Arg -> Ident ;
    Ident_Elected : Arg -> Ident ;
    Ident_Quota : Arg -> Ident ;
    Ident_Winners : Arg -> Ident ;
    Ident_Begin : Arg -> Arg -> Arg -> Ident ;
    Ident_Count: Arg -> Arg -> Arg -> Ident ;
    Ident_BangElectAll : Ident ;
```

```
16        Ident_BangDefeatAll : Ident ;
17        Ident_DefeatMin : Arg -> Arg -> Arg -> Ident ;
18        Ident_DefeatMin' : Arg -> Arg -> Arg -> Ident ;
19        Ident_Minimum : Arg -> Arg -> Ident ;
20        Ident_Transfer : Arg -> Arg -> Arg -> Arg -> Arg -> Ident ;
21        Ident_Run : Arg -> Arg -> Arg -> Ident ;
22        Ident_UnitOne : Ident ;
23
24        -- Arguments
25        _Arg : (A : ArgType) -> (a : Argument A) -> Arg ;
26        _ArgNil, _ArgZ, _Arg1 : Arg ;
27        _ArgMinus, _ArgPlus : Arg -> Arg ;
28        _ArgList : Arg -> Arg -> Arg ;
29        _ArgEmptyList : Arg ;
```

Code 4.3: Defining the `Ident` *and the arguments it needs.*

GF

Each `Ident` uses the `Arg`, so that will be explained first. `Arg` represents any argument used in the formulas. In the example in section 3.1.3, *hopeful C N* has the arguments *C* and *N*. They are the arguments bound through `_Pi`. To turn the variable binding into something that is easier to work with, `_Arg` is used. It takes two arguments and converts them to an `Arg`, which could be the same as the *C* from *hopeful C N*.

In addition to `_Arg`, there are three predefined arguments: `_ArgNil`, `_ArgZ` and `_Arg1`. Each of them represents an argument that has the same meaning no matter what logical formula it is used in. `_ArgNil` is the value "nil", `_ArgZ` is the value "z" or "zero" and `_Arg1` is the value 1. The last `Arg`s are used to represent Celf's plus, minus and lists, and should require no deep explanation.

Having understood the arguments, the identifiers are a bit simpler. Each variable used for the logical formulas concering laws (Note: Appendix with them?) is represented by an `Identifier`. Each `Identifier` takes between zero and five `Arg`s, that are used to construct the Ident.

With the `Ident`s in place (and thereby half the `Atomic`s covered), it is time to look at the other `Atomic`: The one concerning mathematical formulas. The mathematical formulas needs arithemetic operations and inequality operations, so we extend the abstract syntax with the following:

GF

```
1     cat
2         Arg ; Math ; MathFormula ; ArithmeticOperation ; InequalityOperation ;
3
4     fun
5         -- Mathematic operations
6         _MathArg : Arg -> Math ;
7         _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
8         _MathArgs : Math -> ArithmeticOperation -> Math -> Math ;
9
10        _Division, _Addition, _Subtraction, _Multiplication : ArithmeticOperation ;
11        _Greater, _GreaterEqual, _Equal, _LessEqual, _Less : InequalityOperation ;
```

Code 4.4: Defining the mathematical operations.

GF

All mathematical formulas in the logical formulas have some sort of inequality operation. Therefore, `_FinalFormula` is the only `MathFormula`, and it is made from a `Math`, an `InequalityOperation` and a second `Math`.

`_Math` simply takes an `Arg` and produces a `Math`. This can be used either for the `_FinalFormula` directly, or for the `_MathArgs` that takes two `Math`s and a `ArithmeticOperation` and returns a `Math`. That way, a `_MathArgs` can be used inside a `_MathArgs` to signify multiple operations.

The `InequalityOperation` and `ArithmeticOperation` represents the different inequality- and arithmetic operations and should require no explanation

Note: Some sort of ending.

4.2 Constructing the Concrete Implementation

With the abstract syntax in place, the concrete syntax is the next step. The goal here, is to give each function from the abstract syntax a proper linearization so it can understand linear logic. We will need two concrete implementations in total. One for reading and understanding the Celf syntax for linear logic (like the example at the end of section 3.1.3), and one for understanding English.

4.2.1 Concrete linear logic implementation

The

rst concrete implementation we will look at is the one for understanding linear logic, as it is the most important one. Without it, we will not be able to parse the formulas and therefore will not be able to translate them into other languages. Where the abstract syntax was explained starting with the positive and negative types, the concrete syntax will be explained starting with the arguments.

GF

```
1       -- Arguments
2       _Arg A a                    = ss ( a.s ) ;
3       _ArgNil                     = ss ( "nil" ) ;
4       _ArgZ                       = ss ( "z" ) ;
5       _Arg1                       = ss ( "1" ) ;
6       _ArgMinus a                 = ss ( "( s !" ++ a.s ++ ")" ) ;
7       _ArgPlus a                  = ss ( "( p !" ++ a.s ++ ")" ) ;
8       _ArgList a b                = ss ( "( cons !" ++ a.s ++ "!" ++ b.s ++ ")" ) ;
9       _ArgEmptyList               = ss ( "[]" ) ;
```

Code 4.5: The linearization of the `Arguments`.

GF

Remember that `_Arg` is the function that handles the bound variables. While it takes both `ArgType` as argument (`A`), it is not used. With the type of the argument already given in the universal quantification, there is no need to the type of the argument next to the argument itself. `_Arg` therefore only returns the bound variable, which makes it easier for everything else to work with it.

The rest of the `Arg`s are self-explanatory. `_ArgNil`, `_ArgZ` and `_Arg1` simply look for the value they represent. Again, they are arguments that can be used in any logical formula, and are therefore hardcoded into the program. Their values will never change, no matter what formulas are being worked with. `_ArgPlus`, `_ArgMinus` and `_ArgList` have been written to use Celf's syntax (see section 3.1.2) and should look familiar.

Next after the arguments, `Math` and `Ident` are the simplest. As they take up a lot of room, we will look at them individually, starting with `Math`.

GF

```
1       -- Mathematic operations
2       _MathArg arg1               = ss ( arg1.s ) ;
3       _FinalFormula m1 ms m2      = ss ( ms.s ++ m1.s ++ m2.s ) ;
4       _MathArgs arg1 mo arg2      = ss ( "(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")" ) ;
5
6       -- Arithmetic operations
7       _Division                   = ss ( "/" ) ;
```

```
8       _Multiplication             = ss ( "*" ) ;
9       _Addition                   = ss ( "+" ) ;
10      _Subtraction                = ss ( "-" ) ;
11
12      -- Inequality operations
13      _Greater                    = ss ( "!nat-greater" ) ;
14      _GreaterEqual               = ss ( "!nat-greatereq" ) ;
15      _Equal                      = ss ( "!nat-eq" ) ;
16      _LessEqual                  = ss ( "!nat-lesseq" ) ;
17      _Less                       = ss ( "!nat-less" ) ;
```

Code 4.6: The linearization of `Math`

GF

The values for the `ArithmeticOperation`s are their normal symbol. The `InequalityOperation`s, however, use the syntax described in section 3.1.2. For example, > becomes "!nat-greater".

Looking at the `Math` part, it is a bit more advanced. `_Math` is simple enough. Its value is that of the `Arg` is is given. `_MathArgs` is the function that takes care of arithmetic operations between arguments and is surronded by a pair of paratheses. Throughout the voting protocol formulas, this is actually not used, but we will support it anyway.

`_FinalFormula` is the formula that handles inequality operations. Looking at it, one will see that the value it returns has the inequality operation first followed by the two parameters. This is the syntax Celf uses and is thus not an error. One may also notice that the order of the parameters for `_FinalFormula` is not the same as the linearization of it. The normal way to read "x is greater than y" is "x > y" and is how the abstract tree is put together (see line 7 in Code 4.4 on page 14). The concrete implementation is allowed to choose its own way of using the parameters, and can therefore use Celf's notation easily. The formula will be parsed correctly into the abstract syntax anyway.

GF

```
1       -- Identifiers
2       Ident_Uncounted a b         = ss ( "uncounted-ballot" ++ a.s ++ b.s ) ;
3       Ident_Counted a b           = ss ( "counted-ballot" ++ a.s ++ b.s ) ;
4       Ident_Hopeful a b           = ss ( "hopeful" ++ a.s ++ b.s ) ;
5       Ident_Defeated a            = ss ( "!defeated" ++ a.s ) ;
6       Ident_Elected a             = ss ( "!elected" ++ a.s ) ;
7       Ident_Quota a               = ss ( "!quota" ++ a.s ) ;
8       Ident_Winners a             = ss ( "winners" ++ a.s ) ;
9       Ident_Begin a b c           = ss ( "begin" ++ a.s ++ b.s ++ c.s ) ;
10      Ident_Count a b c           = ss ( "count-ballots" ++ a.s ++ b.s ++ c.s ) ;
11      Ident_BangElectAll          = ss ( "!elect-all" ) ;
12      Ident_BangDefeatAll         = ss ( "!defeat-all" ) ;
13      Ident_DefeatMin a b c       = ss ( "defeat-min" ++ a.s ++ b.s ++ c.s ) ;
14      Ident_DefeatMin' a b c      = ss ( "defeat-min'" ++ a.s ++ b.s ++ c.s ) ;
15      Ident_Minimum a b           = ss ( "minimum" ++ a.s ++ b.s ) ;
16      Ident_Transfer a b c d e    = ss ( "transfer" ++ a.s ++ b.s ++ c.s ++ d.s ++ e.s ) ;
17      Ident_Run a b c             = ss ( "run" ++ a.s ++ b.s ++ c.s ) ;
18      Ident_UnitOne               = ss ( "1" ) ;
```

Code 4.7: The linearization of `Ident`

GF

The `Ident`s are rather simple. They take between zero and five arguments, and their value is the name of the function the represent with the amount of arguments attached to it. In the case where the `Ident` takes no arguments, its value is simply that of the function it represents.

With the `Ident`s explained, the only things left are the the positive and negative types and the atomics.

GF

```
1   -- Logic
2   Formular neg                = ss ( neg.s ) ;
3
4   -- Positive types
5   _Atom atom                  = ss ( atom.s ) ;
6   _Bang bang atom             = ss ( bang.s ++ atom.s ) ;
7   _Conj pos1 conj pos2        = ss ( pos1.s ++ conj.s ++ pos2.s ) ;
8   _Unit neg                   = ss ( neg.s ) ;
9   _MPos pos1 pos2             = ss ( pos1.s ++ pos2.s ) ;
10
11  -- Negative types
12  _Pi A B                     = ss ( "Pi" ++ B.$0 ++ ":" ++ A.s ++ "." ++ B.s ) ;
13  _Lolli pos lolli neg        = ss ( pos.s ++ lolli.s ++ neg.s ) ;
14  _Mon pos                    = ss ( "{" ++ pos.s ++ "}" ) ;
15
16  -- Connectives
17  _Conj2                      = ss ( "*" ) ;
18  _Lolli2                     = ss ( "-o" ) ;
19  _Bang2                      = ss ( "!" ) ;
20
21  -- Argument types
22  _Nat                        = ss ( "nat" ) ;
23  _Candidate                  = ss ( "candidate" ) ;
24  _List                       = ss ( "list" ) ;
25
26  -- Atomics
27  Atom_Ident ident            = ss ( ident.s ) ;
28  Atom_Math math              = ss ( math.s ) ;
```

Code 4.8: The linearization of the positive/negative types and the atomics

GF

Like the `_Arg` in Code 4.5, the `Atomics` return the value of the `Ident` or `Math` it takes as a parameter. `_Atom`, `_Neg` and `Formular` work in the same way, though with different parameter types. `_MPos` glues two positives together and is necessary to attach the universal quantifiers to the rest of the formula. `_Lolli`, `_Mon`, the connectives and argument types should not require any explanation.

The only thing that is out of the ordinary is `_Pi`, which should look very familiar, as it uses the same syntax as the demonstration of variable bindings in 3.2.4. The only things that have changed are the names of the types involved. Instead of `Set` and `El`, it is `ArgType` and `Argument`. `_Pi` binds the variable (`B.$0`) and lets the program use it in `B.s`, which is the rest of the formula. `A.s` represents the type of the variable (a candidate, a list or a natuarl number).

With this, the concrete linear logic implementation has been explained. The full concrete implementation can be found in appendix B.

4.2.2 Concrete English implementation

The concrete English implementation is similar in structure to the concrete linear logic implementation. The main difference is in how the strings are put together and we will therefore only examine a few parts of the code. The full implementation can be found in appendix C.

GF

```
1   -- Neg
2   _Pi A B                     = ss ( B.$0 ++ "is a" ++ A.s ++ "." ++ B.s  ) ;
```

Code 4.9: The universal quantification in the English implementation

GF

_Pi is used in the English implementation to let the user know what type the variable has. It is very similar to the one in the linear logic implementation, with the only change being that there is text instead of the colon. Apart from that, it works the same way.

GF

```
1       -- Ident
2       Ident_Hopeful a b
3           = ss ( "there is a hopeful candidate " ++ a.s ++ "with" ++ b.s ++ "counted ballots" ) ;
```

Code 4.10: Idents *and* Args

GF

The Idents have been changed to let them explain the function in English. They are rather static, and the only thing that can change is the argument they recieve, which can be either the bound variables or the modified versions of them.

Everything else is very similar to the linear logic implementation, with the exception of it having been translated into proper English sentences. We will therefore not go into detail with it.

Note: More interesting?
Note: Some sort of ending.

4.3 Running the Program

In this section, I will show the result of parsing some of the logical formulas into the program.

4.3.1 The example

The first formula to be run, is the one from the example in section 2.1. As the program has been written to accept Celf's syntax, I will use the version of the formula from section 3.1.3:

> *tally-votes S H U * uncounted-ballot C L * hopeful C N * !quota Q * !nat-lesseq Q (p ! N) -o { counted-ballot C L * !elected C * tally-votes (s ! S) (s ! H) (s ! U) }*

Before this will run, however, the variables will have to be specified through universal quantifications. Without that, the formula will not parse. Furthermore, GF needs spaces around each individual element, and the plus/minus (p/s ! x) will need a couple of spaces introduced. I will this manually and let the reader confirm that the result is, in fact, the same formula:

> *Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : list . tally-votes S H U * uncounted-ballot C L * hopeful C N * !quota Q * !nat-lesseq Q (p ! N) -o { counted-ballot C L * !elected C * tally-votes (s ! S) (s ! H) (s ! U) }*

With the formula changed to fit the syntax of the program, we will parse it and have it translated into English. We parse it in GF by writing

> p -lang=LawsLin "Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : list . tally-votes S H U * uncounted-ballot C L * hopeful C N * !quota Q * !nat-lesseq Q (p ! N) -o { counted-ballot C L * !elected C * tally-votes (s ! S) (s ! H) (s ! U) }" | l -treebank

What this does, is to tell GF that we want to parse it as the language defined by the -lang flag. After it has been parsed, we tell GF to linearize it and use the -treebank flag to have it linearized in all the languages that have been loaded. The result is the following:

> LawsEng: C is a candidate . H is a natural number . L is a list . N is a natural number . Q is a natural number . S is a natural number . U is a natural number . W is a list .
> If we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play , and
> there is an uncounted vote with highest preference for candidate C with a list L of lower preferences , and
> candidate C is a hopeful with N votes , and
> Q votes are needed to be elected , and (Q is less than or equal to (N plus 1))
> then { there is a counted vote with highest preference for candidate C with a list L of lower preferences , and
> candidate C has been elected , and
> we are counting votes and there are (S minus 1) seats open, (H minus 1) hopefuls, and (U minus 1) uncounted votes in play }

The program output is not the same as the formalized version in section 2.1. One reason for this, is that the formalized version does not always talk about the arguments. For example, the formalized example says "*If we are tallying votes and ...*", but does not speak about the three arguments related to tallying votes.

Another thing that sets the output apart from the formalized example, is the construction of the sentences. Due to some limitations in the program (more about them in 4.4.2), the program uses the same sentence construction for `Idents`, no matter what side of the $\multimap$ they are on. This can lead to confusing sentences, such as the last one in the output.

While the output is not the same as the formalized example, it is nonetheless understandable. **Note: more?**

4.4 Limitations of the program

The program can parse logical formulas and translate them into decent English sentences. There are some limitations, however, which will be described in this section.

4.4.1 Static identifiers

The identifiers are all static. This means that if other identifiers were to be used, or one wanted to use it for other logcial formulars, the code has to be changed. **Note: more**

4.4.2 Sentence construction

The sentences in the program are static. The only thing that changes is the argument given. Because of this, the sentences may not convey the exact meaning in some cases, mainly when they are on the right side of the $\multimap$. **Note: more**

4.4.3 Additional languages

With the way the program is built right now, translating it into other languages is a bit difficult. It requires the translator to know the other language well, as he has to translate complete sentences properly. **Note: More. Remeber grammar resources in GF**

5 Conclusion

Note: Conclusion goes here.

Note: Make sure appendencies are correct.

Bibliography

[1] Short Talk: Celf – A Logical Framework for Deductive and Concurrent Systems: http://www.itu.dk/ carsten/papers/lics08short.pdf

Appendices

A Abstract Implementation

GF

```
abstract Laws = {

    flags startcat = Logic ;

    cat
        Logic ; Prod ; Neg ; Pos ; Lolli ; Bang ; Atomic ; Ident ; Arg ; ArgType ; Conj ; Math ; MathFormula ; ArithmeticOperation ; Ine

    fun
        Formular : Neg -> Logic ;

        -- Positive types
        _Atom : Atomic -> Pos ;
        _Bang : Bang -> Atomic -> Pos ;
        _Conj : Pos -> Conj -> Pos -> Pos ;
        _Unit : Neg -> Pos ;
        _MPos : Pos -> Pos -> Pos ;

        -- Negative types
        _Pi : (A : ArgType) -> (Argument A -> Neg) -> Neg ;
        _Lolli : Pos -> Lolli -> Neg -> Neg ;
        _Mon : Pos -> Neg ;

        -- Connectives
        _Conj2 : Conj ;
        _Lolli2 : Lolli ;
        _Bang2 : Bang ;

        -- Argument types
        _Nat, _Candidate, _List : ArgType ;

        -- Atomics
        Atom_Ident : Ident -> Atomic ;
        Atom_Math : MathFormula -> Atomic ;

        -- Identifiers
        Ident_Uncounted : Arg -> Arg -> Ident ;
        Ident_Counted: Arg -> Arg -> Ident ;
        Ident_Hopeful : Arg -> Arg -> Ident ;
        Ident_Defeated : Arg -> Ident ;
        Ident_Elected : Arg -> Ident ;
        Ident_Quota : Arg -> Ident ;
        Ident_Winners : Arg -> Ident ;
        Ident_Begin : Arg -> Arg -> Arg -> Ident ;
        Ident_Count: Arg -> Arg -> Arg -> Ident ;
        Ident_BangElectAll : Ident ;
        Ident_BangDefeatAll : Ident ;
        Ident_DefeatMin : Arg -> Arg -> Arg -> Ident ;
        Ident_DefeatMin' : Arg -> Arg -> Arg -> Ident ;
        Ident_Minimum : Arg -> Arg -> Ident ;
        Ident_Transfer : Arg -> Arg -> Arg -> Arg -> Arg -> Ident ;
        Ident_Run : Arg -> Arg -> Arg -> Ident ;
        Ident_UnitOne : Ident ;

        -- Arguments
        _Arg : (A : ArgType) -> (a : Argument A) -> Arg ;
        _ArgNil, _ArgZ, _Arg1 : Arg ;
```

```
57          _ArgMinus, _ArgPlus : Arg -> Arg ;
58          _ArgList : Arg -> Arg -> Arg ;
59          _ArgEmptyList : Arg ;
60
61          -- Mathematic operations
62          _MathArg : Arg -> Math ;
63          _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
64          _MathArgs : Math -> ArithmeticOperation -> Math -> Math ;
65
66          _Division, _Addition, _Subtraction, _Multiplication : ArithmeticOperation ;
67          _Greater, _GreaterEqual, _Equal, _LessEqual, _Less : InequalityOperation ;
68  }
```

Code A.1: The full abstract syntax.

GF

B Concrete Linear Logic Implementation

GF

```
concrete LawsLin of Laws = open Prelude in {

    lincat
        Logic, Prod, Neg, Pos, Lolli, Bang, Atomic, Ident, Arg, ArgType, Conj, Math, MathFormula, ArithmeticOperation, InequalityOperati

    lin
        -- Logic
        Formular neg                    = ss ( neg.s ) ;

        -- Positive types
        _Atom atom                      = ss ( atom.s ) ;
        _Bang bang atom                 = ss ( bang.s ++ atom.s ) ;
        _Conj pos1 conj pos2            = ss ( pos1.s ++ conj.s ++ pos2.s ) ;
        _Unit neg                       = ss ( neg.s ) ;
        _MPos pos1 pos2                 = ss ( pos1.s ++ pos2.s ) ;

        -- Negative types
        _Pi A B                         = ss ( "Pi" ++ B.$0 ++ ":" ++ A.s ++ "." ++ B.s ) ;
        _Lolli pos lolli neg            = ss ( pos.s ++ lolli.s ++ neg.s ) ;
        _Mon pos                        = ss ( "{" ++ pos.s ++ "}" ) ;

        -- Connectives
        _Conj2                          = ss ( "*" ) ;
        _Lolli2                         = ss ( "-o" ) ;
        _Bang2                          = ss ( "!" ) ;

        -- Argument types
        _Nat                            = ss ( "nat" ) ;
        _Candidate                      = ss ( "candidate" ) ;
        _List                           = ss ( "list" ) ;

        -- Atomics
        Atom_Ident ident                = ss ( ident.s ) ;
        Atom_Math math                  = ss ( math.s ) ;

        -- Identifiers
        Ident_Uncounted a b             = ss ( "uncounted-ballot" ++ a.s ++ b.s ) ;
        Ident_Counted a b               = ss ( "counted-ballot" ++ a.s ++ b.s ) ;
        Ident_Hopeful a b               = ss ( "hopeful" ++ a.s ++ b.s ) ;
        Ident_Defeated a                = ss ( "!defeated" ++ a.s ) ;
        Ident_Elected a                 = ss ( "!elected" ++ a.s ) ;
        Ident_Quota a                   = ss ( "!quota" ++ a.s ) ;
        Ident_Winners a                 = ss ( "winners" ++ a.s ) ;
        Ident_Begin a b c               = ss ( "begin" ++ a.s ++ b.s ++ c.s ) ;
        Ident_Count a b c               = ss ( "count-ballots" ++ a.s ++ b.s ++ c.s ) ;
        Ident_BangElectAll              = ss ( "!elect-all" ) ;
        Ident_BangDefeatAll             = ss ( "!defeat-all" ) ;
        Ident_DefeatMin a b c           = ss ( "defeat-min" ++ a.s ++ b.s ++ c.s ) ;
        Ident_DefeatMin' a b c          = ss ( "defeat-min'" ++ a.s ++ b.s ++ c.s ) ;
        Ident_Minimum a b               = ss ( "minimum" ++ a.s ++ b.s ) ;
        Ident_Transfer a b c d e        = ss ( "transfer" ++ a.s ++ b.s ++ c.s ++ d.s ++ e.s ) ;
        Ident_Run a b c                 = ss ( "run" ++ a.s ++ b.s ++ c.s ) ;
        Ident_UnitOne                   = ss ( "1" ) ;
```

```
54
55        -- Arguments
56        _Arg A a                      = ss ( a.s ) ;
57        _ArgNil                       = ss ( "nil" ) ;
58        _ArgZ                         = ss ( "z" ) ;
59        _Arg1                         = ss ( "1" ) ;
60        _ArgMinus a                   = ss ( "( s !" ++ a.s ++ ")" ) ;
61        _ArgPlus a                    = ss ( "( p !" ++ a.s ++ ")" ) ;
62        _ArgList a b                  = ss ( "( cons !" ++ a.s ++ "!" ++ b.s ++ ")" ) ;
63        _ArgEmptyList                 = ss ( "[]" ) ;
64
65        -- Mathematic operations
66        _MathArg arg1                 = ss ( arg1.s ) ;
67        _FinalFormula m1 ms m2        = ss ( ms.s ++ m1.s ++ m2.s ) ;
68        _MathArgs arg1 mo arg2        = ss ( "(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")" ) ;
69
70        -- Arithmetic operations
71        _Division                     = ss ( "/" ) ;
72        _Multiplication               = ss ( "*" ) ;
73        _Addition                     = ss ( "+" ) ;
74        _Subtraction                  = ss ( "-" ) ;
75
76        -- Inequality operations
77        _Greater                      = ss ( "!nat-greater" ) ;
78        _GreaterEqual                 = ss ( "!nat-greatereq" ) ;
79        _Equal                        = ss ( "!nat-eq" ) ;
80        _LessEqual                    = ss ( "!nat-lesseq" ) ;
81        _Less                         = ss ( "!nat-less" ) ;
82 }
```

Code B.1: The full concrete implementation for reading linear logic

GF

C Concrete English Implementation

GF

```
concrete LawsEng of Laws = open Prelude in {

    lincat
        Logic, Prod, Neg, Pos, Lolli, Bang, Atomic, Ident, Arg, ArgType, Conj, Math, MathFormula,
        ArithmeticOperation, InequalityOperation, Argument = SS ;

    lin
        -- Logic
        Formular neg                    = ss ( neg.s ) ;

        -- Positive types
        _Atom atom                      = ss ( atom.s ) ;
        _Bang bang atom                 = ss ( atom.s ++ bang.s ) ;
        _Conj pos1 conj pos2            = ss ( pos1.s ++ conj.s ++ pos2.s ) ;
        _Unit neg                       = ss ( neg.s ) ;
        _MPos pos1 pos2                 = ss ( pos1.s ++ pos2.s ) ;

        -- Negative types
        _Pi A B                         = ss ( B.$0 ++ "is a" ++ A.s ++ "." ++ B.s  ) ;
        _Lolli pos lolli neg            = ss ( "If" ++ pos.s ++ lolli.s ++ neg.s ) ;
        _Mon pos                        = ss ( "{" ++ pos.s ++ "}" ) ;

        -- Connectives
        _Conj2                          = ss ( ", and" ) ;
        _Lolli2                         = ss ( "then" ) ;
        _Bang2                          = ss ( "!" ) ;

        -- Argument types
        _Nat                            = ss ( "natural number" ) ;
        _Candidate                      = ss ( "candidate" ) ;
        _List                           = ss ( "list" ) ;

        -- Atomics
        Atom_Ident ident                = ss ( ident.s ) ;
        Atom_Math math                  = ss ( math.s ) ;

        -- Identifiers
        Ident_Uncounted a b
            = ss ( "there is an uncounted vote with highest preference for candidate" ++ a.s ++ "with
                    a list" ++ b.s ++ "of lower preferences" ) ;
        Ident_Counted a b
            = ss ( "there is a counted vote with highest preference for candidate" ++ a.s ++ "with a
                      list" ++ b.s ++ "of lower preferences" ) ;
        Ident_Hopeful a b
            = ss ( "candidate" ++ a.s ++ " is a hopeful with" ++ b.s ++ "votes" ) ;
        Ident_Defeated a
            = ss ( "candidate" ++ a.s ++ "has been defeated" ) ;
        Ident_Elected a
            = ss ( "candidate" ++ a.s ++ "has been elected" ) ;
        Ident_Quota a
            = ss ( a.s ++ "votes are needed to be elected" ) ;
        Ident_Winners a
            = ss ( "the candidates in the list" ++ a.s ++ "have been elected so far" ) ;
        Ident_Begin a b c
            = ss ( "we are beginning the tallying and there are" ++ a.s ++ "seats open," ++ b.s
                      ++ " hopefuls, and" ++ c.s ++ "uncounted votes" ) ;
```

```
57          Ident_Count a b c
58              = ss ( "we are counting votes and there are" ++ a.s ++ "seats open," ++ b.s ++ "hopefuls,
59                        and" ++ c.s ++ "uncounted votes in play" ) ;
60          Ident_BangElectAll
61              = ss ( "there are more open seats than hopefuls" ) ;
62          Ident_BangDefeatAll
63              = ss ( "there are no open seats left" ) ;
64          Ident_DefeatMin a b c
65              = ss ( "we are in the first stage of determining which candidate has the fewest votes and
66                        there are" ++ a.s ++ "seats open, " ++ b.s ++ "hopefuls, and" ++ c.s ++ "potentiel
67                        minimums remaining" ) ;
68          Ident_DefeatMin' a b c
69              = ss ( "we are in the second stage of determining which candidate has the fewest votes and
70                        there are" ++ a.s ++ "seats open, " ++ b.s ++ "hopefuls, and" ++ c.s ++ "potentiel
71                        minimums remaining" ) ;
72          Ident_Minimum a b
73              = ss ( "candidate" ++ a.s ++ "'s with a count of" ++ b.s ++ "votes is a potential minimum" ) ;
74          Ident_Transfer a b c d e
75              = ss ( "the newly defeated candidate" ++ a.s ++"'s" ++ b.s ++ "votes are being tranferred and
76                        there are" ++ c.s ++ "open seats," ++ d.s ++ "hopeful candidates and" ++ e.s ++ "uncounted
77                        votes" ) ;
78          Ident_Run a b c
79              = ss ( "run" ++ a.s ++ b.s ++ c.s ) ;
80          Ident_UnitOne
81              = ss ( "1" ) ;
82
83          -- Arguments
84          _Arg A a                        = ss ( a.s ) ;
85          _ArgNil                         = ss ( "nil" ) ;
86          _ArgZ                           = ss ( "zero" ) ;
87          _Arg1                           = ss ( "1" ) ;
88          _ArgMinus a                     = ss ( "(" ++ a.s ++ "minus 1 )" ) ;
89          _ArgPlus a                      = ss ( "(" ++ a.s ++ "plus 1 )" ) ;
90          _ArgList a b                    = ss ( "consisting of" ++ a.s ++ "and" ++ b.s ) ;
91          _ArgEmptyList                   = ss ( "an empty list" ) ;
92
93          -- Mathematic operations
94          _MathArg arg1                   = ss ( arg1.s ) ;
95          _FinalFormula m1 ms m2          = ss ( "(" ++ m1.s ++ ms.s ++ m2.s ++ ")" ) ;
96          _MathArgs arg1 mo arg2          = ss ( "(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")" ) ;
97
98          -- Arithmetic operations
99          _Division                       = ss ( "/" ) ;
100         _Multiplication                 = ss ( "*" ) ;
101         _Addition                       = ss ( "+" ) ;
102         _Subtraction                    = ss ( "-" ) ;
103
104         -- Inequality operations
105         _Greater                        = ss ( "is greater than" ) ;
106         _GreaterEqual                   = ss ( "is greater than or equal to" ) ;
107         _Equal                          = ss ( "is equal to" ) ;
108         _LessEqual                      = ss ( "is less than or equal to" ) ;
109         _Less                           = ss ( "is less than" ) ;
110
111 }
```

Code C.1: The full concrete English implementation

GF