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):

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

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 $\land$ 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 $\land$. 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 $\land$. A $\land$ 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

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

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. $\overline{A \vdash A}$
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:

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

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.

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

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

$$\begin{array}{ll} \text{Negative Types} & A^- ::= P^- \mid \forall x : A^+. \mid A^+ \multimap A^- \mid \{A^+\} \\ \text{Positive Types} & A^+ ::= A^+ \otimes A^+ \mid \mathbf{1} \mid !A^- \mid A^- \end{array}$$

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.

3.2.1 The abstract module

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.

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} ;
```

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

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

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

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

```
oper
```

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

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

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

```
resource StringOper = {
    oper
        SS : Type = {s : Str} ;
        ss : Str -> SS = \x -> {s = x} ;
        cc : SS -> SS -> SS = \x,y -> ss (x.s ++ y.s) ;
        prefix : Str -> SS -> SS = \p,x -> ss (p ++ x.s) ;
}
```

Code 3.5: A simple resource module.

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

Code 3.6: Using the resource module.

One should note that GF comes with built-in resource modules. The operations in Code 3.5 are some of the operations found in the `Prelude` module.

3.2.4 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`.

```
    cat
        Set ;
```

```
    El Set ;
fun
    Pi : (A : Set) -> (El A -> Set) -> Set ;
    Eq : (A : Set) -> (a,b : El A) -> Set ;
    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 3.2.4, with the addition of `B.$0` while `B.s` is still used.

GF

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

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 3.2.4, `( 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 )`. 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 flexible 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.

Note: Ending of some sort

4 Implementation

Note: Write an introduction

The program has been written specificly for the formulas describing the voting protocol.

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.

It is, therefore, a good thing that we split the connectives of linear logic up in section 3.1.2. 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 ; El 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) -> (El 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 a `El A` in the function of _Pi. The Atomic is used to represent "variables/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 El 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 universal 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 variable/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 variable/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 variables/functions(?), but that it also contains arithmetic and inequality operations (for example !(N + 1 < Q)). This is in place of a function/variable(?), 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 variables/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; Ident and MathFormula. The Ident represents the functions of the formula ("hopeful", "!elect-all", etc.) through the variable/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 it uses. 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 ;
    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_UnitOne : Ident ;

    -- Arguments
    _Arg : (A : ArgType) -> (a : El A) -> Arg ;
    _ArgNil, _ArgZ, _Arg1 : Arg ;
    _ArgMinus, _ArgPlus : Arg -> Arg ;
    _ArgList : Arg -> Arg -> Arg ;
    _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 C from *hopeful* C N. This is the basic argument, but not the only one.

In addition to `_Arg`, there are three predefined arguments: `_ArgNil`, `_ArgZ` and `_Arg1`. Each of them represents an argument that can be used in all logical formulas and is not bound through any `_Pi`. `_ArgNil` is the value "nil", `_ArgZ` is the value "z" or "zero" and `_Arg1` is the value **1**.
The last `Args` 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 `Args`, which are used to construct the `Ident`.

With the `Idents` in place (and thereby half the `Atomics` 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

```
cat
    Arg ; Math ; MathFormula ; ArithmeticOperation ; InequalityOperation ;

fun
    -- Mathematic operations
    _MathArg : Arg -> Math ;
    _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
    _MathArgs : Math -> ArithmeticOperation -> Math -> Math ;

    _Division, _Addition, _Subtraction, _Multiplication : ArithmeticOperation ;
    _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, arithmetic operations a `_MathArgs` can be nested inside a `_MathArgs`.

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

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 first 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 posetive and negative types, the concrete syntax will be explained starting with the arguments.

GF

```
-- Arguments
_Arg A a                       = ss ( a.s ) ;
_ArgNil                        = ss ( "nil" ) ;
_ArgZ                          = ss ( "z" ) ;
_Arg1                          = ss ( "1" ) ;
_ArgMinus a                    = ss ( "( s !" ++ a.s ++ ")" ) ;
_ArgPlus a                     = ss ( "( p !" ++ a.s ++ ")" ) ;
_ArgList a b                   = ss ( "( cons !" ++ a.s ++ "!" ++ b.s ++ ")" ) ;
_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 have it 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 `Args` 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 coded statically 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 be 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

```
-- Mathematic operations
_MathArg arg1                  = ss ( arg1.s ) ;
_FinalFormula m1 ms m2         = ss ( ms.s ++ m1.s ++ m2.s ) ;
_MathArgs arg1 mo arg2         = ss ( "(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")" ) ;

-- Arithmetic operations
_Division                      = ss ( "/" ) ;
_Multiplication                = ss ( "*" ) ;
_Addition                      = ss ( "+" ) ;
_Subtraction                   = ss ( "-" ) ;

-- Inequality operations
_Greater                       = ss ( "!nat-greater" ) ;
_GreaterEqual                  = ss ( "!nat-greatereq" ) ;
_Equal                         = ss ( "!nat-eq" ) ;
_LessEqual                     = ss ( "!nat-lesseq" ) ;
```

```
        _Less                          = ss ( "!nat-less" ) ;
```

Code 4.6: The linearization of `Math`

GF

The values for the `ArithmeticOperations` are their normal symbol. The `InequalityOperations`, however, use the syntax described in section 3.1.2. For example, $>$ becomes "!nat-less". The end formula (`_FinalFormula`) uses Celf's syntax. $x > y$ is therefore `!nat-less x y`

Looking at the `Math` part, it is a bit more advanced. `_Math` is simple enough. Its value is simply the any argumet. `_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

```
        -- Ident
        Ident_Hopeful c n s h u q l m w
            = ss ("hopeful" ++ c.s ++ n.s) ;
        Ident_Tally c n s h u q l m w
            = ss ("tally-votes" ++ s.s ++ h.s ++ u.s | "count-ballots" ++ s.s ++ h.s ++ u.s) ;
        Ident_BangElectAll c n s h u q l m w
            = ss ("!elect-all") ;
        Ident_Elected c n s h u q l m w
            = ss ("!elected" ++ c.s) ;
        Ident_Defeated c n s h u q l m w
            = ss ("!defeated" ++ c.s) ;
        Ident_Quota c n s h u q l m w
            = ss ("!quota" ++ q.s) ;
        Ident_Minimum c n s h u q l m w
            = ss ("minimum" ++ c.s ++ n.s) ;
        Ident_DefeatMin c n s h u q l m w
            = ss ("defeat-min" ++ s.s ++ h.s ++ m.s) ;
        Ident_DefeatMin' c n s h u q l m w
            = ss ("defeat-min'" ++ s.s ++ h.s ++ m.s) ;
        Ident_Transfer c n s h u q l m w
            = ss ("transfer" ++ c.s ++ n.s ++ s.s ++ h.s ++ u.s) ;
        Ident_Uncounted c n s h u q l m w
            = ss ("uncounted-ballot" ++ c.s ++ l.s) ;
        Ident_Counted c n s h u q l m w
            = ss ("counted-ballot" ++ c.s ++ l.s) ;
        Ident_Winners c n s h u q l m w
            = ss ("winners" ++ w.s) ;
        Ident_Begin c n s h u q l m w
            = ss ("begin" ++ s.s ++ h.s ++ u.s) ;
```

Code 4.7: The linearization of `Ident`

GF

There are a couple of things to note here. The first is that all the `Idents` can take nine arguments (`c n s h u q l m w`), but none of them use more than five. The abstract syntax for all the `Ident` was given nine arguments to avoid having to make a seperate abstract for each `Ident`. The reason for choosing nine arguments instead of five (the highest number used), was to have each argument represented. Each argument corresponds to one of the `Args` (with the exception of z, 1 and nil), making it easy to figoure out what the `Ident` uses. It is important to note that while "s" is used to represent `Arg_S`, it does not mean the value has to be the value of `Arg_S`. It can be any of the `Args`.

Just like the `_FinalFormula`, the arguments do not have to be used in the order listed. They do not even have to be used. `Ident_Hopeful` will only look for any two arguments following and only use those two arguments. The rest will be ignored. The same goes for the rest of the `Idents`.
The other thing to note is in `Ident_Tally`, where there is a | in the value of the ident. This line means that value for `Ident_Tally` can be any of the strings on either side of the |. It is an easy way of letting a function use multiple keywords.

The last things to examine are the positive and negative types and the atomics.

GF

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

-- Pos
_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) ;

-- Neg
_Pi arg                        = ss ("Pi" ++ arg.s ++ ":" ++ ("nat" | "list" | "candidate") ++ ".") ;
_Lolli pos lolli neg           = ss (pos.s ++ lolli.s ++ neg.s) ;
_Mon pos                       = ss ("{" ++ pos.s ++ "}") ;

-- Atomic
Atom_Ident ident               = ss (ident.s) ;
Atom_Math mathf                = ss (mathf.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. This is necessary to attach the universal quantifiers to the rest of the formula.

`_Pi` looks for a string that describes the universal quantification according to Celf's syntax (Pi x : t). It uses an argument as the x with the t being either "nat", "list" or "candidate". Originally, the idea was to use GF's variable bindings. They would allow the user to specify variables with certain meanings (like saying that "S" should be "a set of seats"), making it possible to define variables dynamically for each formula. However, due to a lack of time and lack of information about the variable bindings, this idea was scrapped and the variables were made static (hard-coded into the program with predefined values).

Note: Some sort of ending.
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

```
    -- Neg
    --_Pi arg                          = ss (arg.s ++ "is" ++ ("a list" | "a set" | "a candidate")) ;
    _Pi arg                          = ss ("") ;
```

Code 4.9: The universal quantification in the English implementation

GF

As stated adter Code 4.8, `_Pi` does not work the way originally intended. It was therefore decided that, to avoid a huge mess, the lineariation of `_Pi` should not print anything.

GF

```
    -- Ident
    Ident_Hopeful c n s h u q l m w
        = ss ("there is a hopeful" ++ c.s ++ " with" ++ n.s) ;

    -- Arg
    Arg_C                            = ss ("candidate (C)") ;
    Arg_N                            = ss ("a set of counted ballots (N)") ;
```

Code 4.10: `Ident`*s and* `Arg`*s*

GF

Both `Ident`s and `Arg`s have been given predefined values in the English implementation. This can be done, as each `Ident` and `Arg` mean something specific in relation to the voting protocols. Each `Ident` has a certain meaning, where the `Arg`s are used to specify what the sentence talks about. Likewise, each `Arg` has a certain meaning, which can either be about something specific (such as a candidate) or something that can be one or more things. The latter is referred to as "a set of things", as that can indicate both plural and singular values.

Note: More interesting?

Note: Some sort of ending.

4.3 Running the Program

With the program written, all that is left is to test it. To do that, I decided to use three formulas, as they gave a nice spread of the functions and arguments. The formulas I chose were GF:count/2, GF:count/4_1 and GF:defeat-min'/1. **Note: Reference to where they have been acquired from.**

We start with GF:count/2. **Note: Make sure it's like the one from the example**

GF:count/2
*LawsLin: Pi C : nat . Pi H : nat . Pi L : nat . Pi N : nat . Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : nat . tally-votes (s ! (s ! S)) (s ! H) (s ! U) * uncounted-ballot C L * hopeful C N * !quota Q * !nat-lesseq Q (s ! N) * winners W -o { counted-ballot C L * !elected C * winners (cons ! C ! W) * tally-votes (s ! S) H U }*

LawsEng: if [we are tallying votes and there is a set of seats (S) minus 1 minus 1 open, a set of hopeful candidates (H) minus 1 , and a set of uncounted ballots (U) minus 1 cast] and [there is an uncounted ballot with highest preference for a certain candidate (C) with a list (L) lower preferences] and [there is a hopeful candidate (C) with a set of counted ballots (N)] and [a set of votes (Q) are needed to be elected] and [the amount of a set of votes (Q) is less than or equal to the amount of a set of counted ballots (N) minus 1] and [the candidates in a list of winners (W) have been elected thus far] then { [there is a counted ballot with highest preference for a certain candidate (C) with a list (L) lower preferences] and [candidate (C) has been (and will remain) elected] and [the candidates in list containing candidate (C) and a list of winners (W) have been elected thus far] and [we are tallying votes and there is a set of seats (S) minus 1 open, a set of hopeful candidates (H) , and a set of uncounted ballots (U) cast] }

Note: Something about the results.

GF:count/4_1
*LawsLin: Pi C : nat . Pi C' : nat . Pi H : nat . Pi L : nat . Pi S : nat . Pi U : nat . tally-votes S H U * uncounted-ballot C (cons ! C' ! L) * !elected C -o { uncounted-ballot C' L * tally-votes S H U }*

LawsEng: if [we are tallying votes and there is a set of seats (S) open, a set of hopeful candidates (H) , and a set of uncounted ballots (U) cast] and [there is an uncounted ballot with highest preference for a certain candidate (C) with list containing second candidate (C') and a list (L) lower preferences] and [candidate (C) has been (and will remain) elected] then { [there is an uncounted ballot with highest preference for a certain second candidate (C') with a list (L) lower preferences] and [we are tallying votes and there is a set of seats (S) open, a set of hopeful candidates (H) , and a set of uncounted ballots (U) cast] }

GF:defeat-min'/1
*LawsLin: Pi C : nat . Pi C' : nat . Pi H : nat . Pi M : nat . Pi N : nat . Pi N' : nat . Pi S : nat . defeat-min' S H (s ! M) * minimum C N * minimum C' N' * !nat-less N N' -o { minimum C N * hopeful C' N' * defeat-min' S (s ! H) M }*

LawsEng: if [we are in the second step of determining which candidate has the fewest votes and there is a set of seats (S) open, a set of hopeful candidates (H) , and a set of potential minimums (M) minus 1 remaining] and [candidate (C) 's count of a set of counted ballots (N) is a potential minimum] and [second candidate (C') 's count of a modified set of counted ballots (N) is a potential minimum] and [the amount of a set of counted ballots (N) is less than the amount of a modified set of counted ballots (N)] then { [candidate (C) 's count of a set of counted ballots (N) is a potential minimum] and [there is a hopeful second candidate (C') with a modified set of counted ballots (N)] and [we are in the second step of determining which candidate has the fewest votes and there is a set of seats (S) open, a set of hopeful candidates (H) minus 1 , and a set of potential minimums (M) remaining] }

5 Conclusion

Note: Conclusion goes here.

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 ; InequalityOperation ; El ArgType ;

    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) -> (El 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_UnitOne : Ident ;

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

```
        _ArgMinus, _ArgPlus : Arg -> Arg ;
        _ArgList : Arg -> Arg -> Arg ;
        _ArgEmptyList : Arg ;

        -- Mathematic operations
        _MathArg : Arg -> Math ;
        _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
        _MathArgs : Math -> ArithmeticOperation -> Math -> Math ;

        _Division, _Addition, _Subtraction, _Multiplication : ArithmeticOperation ;
        _Greater, _GreaterEqual, _Equal, _LessEqual, _Less : InequalityOperation ;
}
```

Code A.1: The full abstract syntax.

GF

B Concrete Linear Logic Implementation

GF

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

    lincat
        Logic, Prod, Neg, Pos, Lolli, Bang, Atomic, Ident, Arg, ArgColl, Conj, Math, MathFormula,
        ArithmeticOperation, InequalityOperation = {s : Str} ;

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

        -- Pos
        _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) ;

        -- Neg
        _Pi arg                         = ss ("Pi" ++ arg.s ++ ":" ++ ("nat" | "list" | "candidate") ++ ".") ;
        _Lolli pos lolli neg            = ss (pos.s ++ lolli.s ++ neg.s) ;
        _Mon pos                        = ss ("{" ++ pos.s ++ "}") ;

        -- Atomic
        Atom_Ident ident                = ss (ident.s) ;
        Atom_Math mathf                 = ss (mathf.s) ;

        -- Ident
        Ident_Hopeful c n s h u q l m w
            = ss ("hopeful" ++ c.s ++ n.s) ;
        Ident_Tally c n s h u q l m w
            = ss ("tally-votes" ++ s.s ++ h.s ++ u.s | "count-ballots" ++ s.s ++ h.s ++ u.s) ;
        Ident_BangElectAll c n s h u q l m w
            = ss ("!elect-all") ;
        Ident_Elected c n s h u q l m w
            = ss ("!elected" ++ c.s) ;
        Ident_Defeated c n s h u q l m w
            = ss ("!defeated" ++ c.s) ;
        Ident_Quota c n s h u q l m w
            = ss ("!quota" ++ q.s) ;
        Ident_Minimum c n s h u q l m w
            = ss ("minimum" ++ c.s ++ n.s) ;
        Ident_DefeatMin c n s h u q l m w
            = ss ("defeat-min" ++ s.s ++ h.s ++ m.s) ;
        Ident_DefeatMin' c n s h u q l m w
            = ss ("defeat-min'" ++ s.s ++ h.s ++ m.s) ;
        Ident_Transfer c n s h u q l m w
            = ss ("transfer" ++ c.s ++ n.s ++ s.s ++ h.s ++ u.s) ;
        Ident_Uncounted c n s h u q l m w
            = ss ("uncounted-ballot" ++ c.s ++ l.s) ;
        Ident_Counted c n s h u q l m w
            = ss ("counted-ballot" ++ c.s ++ l.s) ;
        Ident_Winners c n s h u q l m w
            = ss ("winners" ++ w.s) ;
```

```
        Ident_Begin c n s h u q l m w
            = ss ("begin" ++ s.s ++ h.s ++ u.s) ;

        -- Arg
        Arg_C                           = ss ("C") ;
        Arg_C'                          = ss ("C'") ;
        Arg_N                           = ss ("N") ;
        Arg_N'                          = ss ("N'") ;
        Arg_S                           = ss ("S") ;
        Arg_H                           = ss ("H") ;
        Arg_U                           = ss ("U") ;
        Arg_Q                           = ss ("Q") ;
        Arg_L                           = ss ("L") ;
        Arg_M                           = ss ("M") ;
        Arg_W                           = ss ("W") ;
        Arg_0                           = ss ("z") ;
        Arg_1                           = ss ("1") ;
        Arg_Nil                         = ss ("nil") ;
        _Arg arg                        = ss (arg.s) ;
        _ArgPlus arg                    = ss ("( p !" ++ arg.s ++ ")") ;
        _ArgMinus arg                   = ss ("( s !" ++ arg.s ++ ")") ;
        _ArgListEmpty                   = ss ("[]") ;
        _ArgList arg1 arg2              = ss ("( cons !" ++ arg1.s ++ "!" ++ arg2.s ++ ")") ;

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

        -- Math
        _FinalFormula m1 ms m2          = ss (ms.s ++ m1.s ++ m2.s) ;
        _Math arg1                      = ss (arg1.s) ;
        _MathArgs arg1 mo arg2          = ss ("(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")") ;

        -- ArithmeticOperation
        _Division                       = ss ("/") ;
        _Multiplication                 = ss ("*") ;
        _Addition                       = ss ("+") ;
        _Subtraction                    = ss ("-") ;

        -- InequalityOperation
        Greater                         = ss ("!nat-greater") ;
        GreaterEqual                    = ss ("!nat-greatereq") ;
        Equal                           = ss ("!nat-eq") ;
        LessEqual                       = ss ("!nat-lesseq") ;
        Less                            = ss ("!nat-less") ;
}
```

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

GF

C Concrete English Implementation

GF

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

    lincat
        Logic, Prod, Neg, Pos, Lolli, Bang, Atomic, Ident, Arg, ArgColl, Conj, Math, MathFormula,
        ArithmeticOperation, InequalityOperation = {s : Str} ;

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

        -- Pos
        _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) ;

        -- Neg
        --_Pi arg                         = ss (arg.s ++ "is" ++ ("a list" | "a set" | "a candidate")) ;
        _Pi arg                         = ss ("") ;
        _Lolli pos lolli neg            = ss ("if" ++ pos.s ++ lolli.s ++ "{" ++ neg.s ++ "}") ;
        _Mon pos                        = ss (pos.s) ;

        -- Atomic
        Atom_Ident ident                = ss ("[" ++ ident.s ++ "]") ;
        Atom_Math math                  = ss ("[" ++ math.s ++ "]") ;

        -- Ident
        Ident_Hopeful c n s h u q l m w
            = ss ("there is a hopeful" ++ c.s ++ " with" ++ n.s) ;
        Ident_Tally c n s h u q l m w
            = ss ("we are tallying votes and there is" ++ s.s ++ "open," ++ h.s ++ ", and" ++ u.s ++ "cast") ;
        Ident_BangElectAll c n s h u q l m w
            = ss ("there is more open seats than hopefuls") ;
        Ident_Elected c n s h u q l m w
            = ss (c.s ++ "has been (and will remain) elected") ;
        Ident_Defeated c n s h u q l m w
            = ss (c.s ++ "has been (and will remain) defeated") ;
        Ident_Quota c n s h u q l m w
            = ss (q.s ++ "are needed to be elected") ;
        Ident_Minimum c n s h u q l m w
            = ss (c.s ++ "'s count of" ++ n.s ++ "is a potential minimum") ;
        Ident_DefeatMin c n s h u q l m w
            = ss ("we are in the first step of determining which candidate has the fewest votes and there is"
                    ++ s.s ++ "open, " ++ h.s ++ ", and" ++ m.s ++ "remaining") ;
        Ident_DefeatMin' c n s h u q l m w
            = ss ("we are in the second step of determining which candidate has the fewest votes and
                    there is" ++ s.s ++ "open, " ++ h.s ++ ", and" ++ m.s ++ "remaining") ;
        Ident_Transfer c n s h u q l m w
            = ss ("we are in the second step of determining which candidate has the fewest votes and
                    there is" ++ s.s ++ "open, " ++ h.s ++ ", and" ++ m.s ++ "remaining") ;
        Ident_Uncounted c n s h u q l m w
            = ss ("there is an uncounted ballot with highest preference for a certain" ++ c.s ++ "with"
                    ++ l.s ++ " lower preferences") ;
        Ident_Counted c n s h u q l m w
            = ss ("there is a counted ballot with highest preference for a certain" ++ c.s ++ "with" ++ l.s
```

```
                ++ " lower preferences") ;
    Ident_Winners c n s h u q l m w
        = ss ("the candidates in" ++ w.s ++ "have been elected thus far") ;
    Ident_Begin c n s h u q l m w
        = ss ("we are beginning the tallying and there is " ++ s.s ++ "open," ++ h.s ++ ", and" ++ u.s
                ++ "cast") ;

    -- Arg
    Arg_C                           = ss ("candidate (C)") ;
    Arg_C'                          = ss ("second candidate (C')") ;
    Arg_N                           = ss ("a set of counted ballots (N)") ;
    Arg_N'                          = ss ("a modified set of counted ballots (N)") ;
    Arg_S                           = ss ("a set of seats (S)") ;
    Arg_H                           = ss ("a set of hopeful candidates (H)") ;
    Arg_U                           = ss ("a set of uncounted ballots (U)") ;
    Arg_Q                           = ss ("a set of votes (Q)") ;
    Arg_L                           = ss ("a list (L)") ;
    Arg_M                           = ss ("a set of potential minimums (M)") ;
    Arg_W                           = ss ("a list of winners (W)") ;
    Arg_0                           = ss ("0") ;
    Arg_1                           = ss ("1") ;
    Arg_Nil                         = ss ("nil") ;
    _Arg arg                        = ss (arg.s) ;
    _ArgPlus arg                    = ss (arg.s ++ "plus 1") ;
    _ArgMinus arg                   = ss (arg.s ++ "minus 1") ;
    _ArgListEmpty                   = ss ("empty list") ;
    _ArgList arg1 arg2              = ss ("list containing" ++ arg1.s ++ "and" ++ arg2.s) ;

    _Conj2                          = ss ("and") ;
    _Lolli2                         = ss ("then") ;
    _Bang2                          = ss ("that is not consumed") ;

    -- Math
    _FinalFormula m1 ms m2          = ss (m1.s ++ ms.s ++ m2.s) ;
    _Math arg1                      = ss ("the amount of" ++ arg1.s) ;
    _MathArgs arg1 mo arg2          = ss ("(" ++ "the amount of" ++ arg1.s ++ mo.s ++ "the amount of"
                ++ arg2.s ++ ")") ;

    -- ArithmeticOperation
    _Division                       = ss ("divided by") ;
    _Multiplication                 = ss ("multiplied with") ;
    _Addition                       = ss ("plus") ;
    _Subtraction                    = ss ("minus") ;

    -- InequalityOperation
    Greater                         = ss ("is greater than") ;
    GreaterEqual                    = ss ("is greater than or equal to") ;
    Equal                           = ss ("is equal to") ;
    LessEqual                       = ss ("is less than or equal to") ;
    Less                            = ss ("is less than") ;
}
```

Code C.1: The full concrete English implementation