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

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

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).

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`.

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 ++ "}"} ;
```

```
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".

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

```
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.

GF

GF

```
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.

GF

Looking back at line 8 in Code 3.3, `_VoteCard` and `_BlankBallot` only exist in singular version. Suppose we want to be able to use the plural version of them, we could do two things: We could make a `_VoteCardSg` and a `_VoteCardPl` that holds the singular and plural version respectively. Or we could use parameters to choose.

A parameter consists of the parameter's name and its values, seperated by a horizonal line. Parameters can be added to either a resource module (by which the implementations using the resource module have access to it) or to a concrete implementation (limiting the parameter to that implementation).

GF

```
param
    Number = Sg | Pl ;
```

Code 3.7: Defining a parameter for singular and plural versions of a word.

GF

The `lincat` of a type also needs to reflect the fact that it takes a parameter. This is done as shown in 3.7.

GF

```
lincat
    Recipient = {s : Number => Str} ;
```

Code 3.8: Telling the `lincat` that it takes a parameter and generates a Str.

GF

The last thing that needs to be changed is the `lin` of the category that has been changed. Tables are used for this purpose. A table holds the different possibilities for the parameter, along with what is returned based on the parameter. Code 3.8 shows a table for _VoteCard.

GF

```
lin
    _VoteCard = {
        s = table {
            Sg => "an authorization card" ;
            Pl => "multiple authorization cards"
        }
    } ;
```

Code 3.9: A table for _VoteCard that returns the singular or plural version of the word depending on the parameter.

GF

Code 3.10 shows how the parameters are used in the ConcreteEnglish program. Assuming i1 is `_VoteCard`, and i2 is `_BlankBallot`, FormulaSg creates a string saying "if i give an authorization card then i get a blank ballot" where line 2 creates a string saying "if i give multiple authorization cards i get multiple blank ballots".

GF

```
lin
    FormulaSg i1 c i2 = {s = "if i give" ++ i1.s ! Sg ++ c.s ++ "i get" ++ i2.s ! Sg} ;
    FormulaPl i1 c i2 = {s = "if i give" ++ i1.s ! Pl ++ c.s ++ "i get" ++ i2.s ! Pl} ;
```

Code 3.10: Giving the parameter as an argument.

GF

Note: Need some sort of ending here.

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.

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 ;
7          Pi Pos ;
8
9      fun
10         Formular : Neg -> Logic ;              -- Represents the entire logical formula
11
12         -- Positive types
13         _Atom : Atomic -> Pos ;                -- Turning an atomic into a positive type
14         _Bang : Bang -> Atomic -> Pos ;        -- Using the unrestricted modality
15         _Conj : Pos -> Conj -> Pos -> Pos ;    -- Using the simultaneous conjunction
16
17         -- Negative types
18         _Pi : (k : Pos) -> Neg -> Neg ;        -- Using the universal quantification
19         _Lolli : Pos -> Lolli -> Neg -> Neg ;  -- Using the linear implication
20         _Mon : Pos -> Neg ;                    -- Turning a positive into a negative
21
22         _Conj2 : Conj ;                        -- Simultaneous conjunction
23         _Lolli2 : Lolli ;                      -- Linear implication
24         _Bang2 : Bang ;                        -- Unrestricted modality
25 }
```

Code 4.1: The first abstract syntax.

GF

Going through the abstract syntax, one will notice two things that do not stem from the connectives: There is something called an `"Atomic"` (line 13) and the `_Pi` has a strange syntax compared to the rest. The `Atomic` is used to represent "variables/functions(?)" along with the arguments they take. They will be explained later in this section.

Note: Check if this part is factually right

The syntax for `_Pi` is a so-called "dependent type" in GF. It can be used for binding variables in GF, which would make it possible to use the program for virtually anything written in linear logic/Celf. Unfortunately, due to time constraints, it was not implemented and `_Pi` is actually not used in the program. **Note: Should I write this or say it in another way?**

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

Now we have the `Atomic` defined, but we have introduced two new categories at the same time; `Ident` and `MathFormula`. The `Ident` represents the variables 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 needed. The abstract syntax is once again extended.

GF

```
cat
    Ident ; Arg ; ArgColl ;

fun
    -- Identifiers
    Ident_Hopeful, Ident_Tally, Ident_BangElectAll, Ident_Elected, Ident_Defeated, Ident_Quota, Ident_Minimum,
    Ident_DefeatMin, Ident_Transfer, Ident_Counted, Ident_Uncounted, Ident_Winners, Ident_Begin : ArgColl ->
    ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> Ident ;

    -- Arguments
    Arg_C, Arg_N, Arg_S, Arg_H, Arg_U, Arg_Q, Arg_L, Arg_M, Arg_W, Arg_0, Arg_1, Arg_Nil : Arg ;
    _Arg : Arg -> ArgColl ;
    _ArgPlus, _ArgMinus : ArgColl -> ArgColl ;
    _ArgListEmpty : ArgColl ;
    _ArgList : ArgColl -> ArgColl ->  ArgColl ;
```

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

GF

This looks confusing, but there is a method to the madness. Starting with the arguments, there are a lot with the name `Arg_something`. Each of them represents one of the arguments used by the different variables (S, N, H, etc.). The `Arg`s have to be converted into `ArgColl` to be used. This is done to make it possible to use lists of arguments along with Celf's notation of plus and minus.
Both `_ArgPlus` and `_ArgMinus` take an `ArgColl` as argument and produces an `ArgColl`. The reason behind this, is that a logical formula can say "(S - 1) - 1". If they took an `Arg` as the argument, it would not be possible to represent such a case.
`_ArgListEmpty` is simply an `ArgColl`. It does not take any arguments. `_ArgList`, on the other hand, takes

two ArgColls to produce an ArgColl. This way, it is possible to use _ArgPlus/_ArgMinus inside the list, which may or may not be needed.

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 up to nine ArgColls and produces an Ident. Nine arguments may seem excessive, but it is up to the concrete implementations to decide how many of the arguments that are used. An Ident does not have to use all nine. **Note: Explain better**

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
        ArgColl ; Math ; MathFormula ; ArithmeticOperation ; InequalityOperation ;

    fun
        -- Mathematic operations
        _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
        _Math : ArgColl -> Math ;
        _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 ArgColl and produces a Math. This can be used either for the _FinalFormula directly, or for the _MathArgs that takes two Maths and a ArithmeticOperation and returns a Math. That way, arithmetic operations can be either a simple operation, or the Math(s) it is made of can be made _MathArgs.
The InequalityOperation and ArithmeticOperation are simple and need no futher explanation.

With the connectives, atomics, identifiers, arguments and mathematical operations explained, the final abstract syntax is the following: **Note: Put this into appendix?**

GF

```
abstract Laws = {

    flags startcat = Logic ;

    cat
        Logic ; Prod ; Neg ; Pos ; Lolli ; Bang ; Atomic ; Ident ; Arg ; ArgColl ; Conj ; Math ; MathFormula ;
        ArithmeticOperation ; InequalityOperation ;
        Pi Pos ;

    fun
        Formular : Neg -> Logic ;

        -- Pos
        _Atom : Atomic -> Pos ;
        _Bang : Bang -> Atomic -> Pos ;
        _Conj : Pos -> Conj -> Pos -> Pos ;

        -- Neg
```

```
        _Pi : (k : Pos) -> Neg -> Neg ;
        _Lolli : Pos -> Lolli -> Neg -> Neg ;
        _Mon : Pos -> Neg ;

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

        -- Ident
        Ident_Hopeful, Ident_Tally, Ident_BangElectAll, Ident_Elected, Ident_Defeated, Ident_Quota, Ident_Minimum,
        Ident_DefeatMin, Ident_Transfer, Ident_Counted, Ident_Uncounted, Ident_Winners, Ident_Begin : ArgColl ->
        ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> ArgColl -> Ident ;

        -- Arg
        Arg_C, Arg_N, Arg_S, Arg_H, Arg_U, Arg_Q, Arg_L, Arg_M, Arg_W, Arg_0, Arg_1, Arg_Nil : Arg ;
        _Arg : Arg -> ArgColl ;
        _ArgPlus, _ArgMinus : ArgColl -> ArgColl ;
        _ArgListEmpty : ArgColl ;
        _ArgList : ArgColl -> ArgColl ->  ArgColl ;

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

        -- Math
        _FinalFormula : Math -> InequalityOperation -> Math -> MathFormula ;
        _Math : ArgColl -> Math ;
        _MathArgs : Math -> ArithmeticOperation -> Math -> Math ;

        _Division, _Addition, _Subtraction, _Multiplication : ArithmeticOperation ;
        Greater, GreaterEqual, Equal, LessEqual, Less : InequalityOperation ;
}
```

Code 4.5: The full abstract syntax.

GF

4.2 Building 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.

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. Unlike the abstract syntax, we will start with looking at the parts that will look for specific values. Without understanding them, attempting to understand what happens when they are put together is not an easy task.

As everything else is built up around the arguments, they are the simplest of it all. We will start examining them.

GF

```
        -- Arg
        Arg_C                   = ss ("C") ;
        Arg_N                   = ss ("N") ;
        Arg_S                   = ss ("S") ;
        Arg_H                   = ss ("H") ;
        Arg_U                   = ss ("U") ;
```

```
7       Arg_Q                       = ss ("Q") ;
8       Arg_L                       = ss ("L") ;
9       Arg_M                       = ss ("M") ;
10      Arg_W                       = ss ("W") ;
11      Arg_0                       = ss ("z") ;
12      Arg_1                       = ss ("1") ;
13      Arg_Nil                     = ss ("nil") ;
14      _Arg arg                    = ss (arg.s) ;
15      _ArgPlus arg                = ss ("( p !" ++ arg.s ++ ")") ;
16      _ArgMinus arg               = ss ("( s !" ++ arg.s ++ ")") ;
17      _ArgListEmpty               = ss ("[]") ;
18      _ArgList arg1 arg2          = ss ("( cons !" ++ arg1.s ++ "!" ++ arg2.s ++ ")") ;
19
20      _Conj2                      = ss ("*") ;
21      _Lolli2                     = ss ("-o") ;
22      _Bang2                      = ss ("!") ;
```

Code 4.6: The linearization of the `Arguments`.

GF

The arguments themselves are fairly straight-forward. Their value is the value that corresponds to them in the formulas. Similarily, the value of `_Arg` is the string of any of the arguments. The `ArgColl`s are also self-explanatory. The `_ArgPlus`, `_ArgMinus` and `_ArgList` have been written to use Celf's syntax (see section 3.1.2). The connectives have been included here and are also using Celf's syntax.

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       -- Math
2       _FinalFormula m1 ms m2      = ss (ms.s ++ m1.s ++ m2.s) ;
3       _Math arg1                  = ss (arg1.s) ;
4       _MathArgs arg1 mo arg2      = ss ("(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")") ;
5
6       -- ArithmeticOperation
7       _Division                   = ss ("/") ;
8       _Multiplication             = ss ("*") ;
9       _Addition                   = ss ("+") ;
10      _Subtraction                = ss ("-") ;
11
12      -- InequalityOperation
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 4.7: 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-less".
Looking at the `Math` part, it is a bit more advanced. `_Math` is simple enough. It has the value of any argumet (we will get to them later). `_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       -- Ident
2       Ident_Hopeful c n s h u q l m w
3           = ss ("hopeful" ++ c.s ++ n.s) ;
4       Ident_Tally c n s h u q l m w
5           = ss ("tally-votes" ++ s.s ++ h.s ++ u.s | "count-ballots" ++ s.s ++ h.s ++ u.s) ;
6       Ident_BangElectAll c n s h u q l m w
7           = ss ("!elect-all") ;
8       Ident_Elected c n s h u q l m w
9           = ss ("!elected" ++ c.s) ;
10      Ident_Defeated c n s h u q l m w
11          = ss ("!defeated" ++ c.s) ;
12      Ident_Quota c n s h u q l m w
13          = ss ("!quota" ++ q.s) ;
14      Ident_Minimum c n s h u q l m w
15          = ss ("minimum" ++ c.s ++ n.s) ;
16      Ident_DefeatMin c n s h u q l m w
17          = ss ("defeat-min" ++ s.s ++ h.s ++ m.s) ;
18      Ident_Transfer c n s h u q l m w
19          = ss ("transfer" ++ c.s ++ n.s ++ s.s ++ h.s ++ u.s) ;
20      Ident_Uncounted c n s h u q l m w
21          = ss ("uncounted-ballot" ++ c.s ++ l.s) ;
22      Ident_Counted c n s h u q l m w
23          = ss ("counted-ballot" ++ c.s ++ l.s) ;
24      Ident_Winners c n s h u q l m w
25          = ss ("winners" ++ w.s) ;
26      Ident_Begin c n s h u q l m w
27          = ss ("begin" ++ s.s ++ h.s ++ u.s) ;
```

Code 4.8: The linearization of `Ident`

GF

There are a couple of things to note here. The first is that all the `Ident`s 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 `Arg`s (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 `Arg`s.
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 `Ident`s.
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

```
1       -- Logic
2       Formular neg                    = ss (neg.s) ;
3
```

```
4       -- Pos
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
9       -- Neg
10      _Pi _ neg                     = ss (neg.s) ;
11      _Lolli pos lolli neg          = ss ("(" ++ pos.s ++ ")" ++ lolli.s ++ neg.s) ;
12      _Mon pos                      = ss ("{" ++ pos.s ++ "}") ;
13
14      -- Atomic
15      Atom_Ident ident              = ss (ident.s) ;
16      Atom_Math mathf               = ss (mathf.s) ;
```

Code 4.9: The linearization of the positive/negative types and the atomics

GF

Note: write stuff here

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