Laws and Logic

Bachelor Project

Bachelor in Software Development,
IT-University of Copenhagen

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

Hand-in date: May 22nd, 2013

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

It is generally agreed upon that elections consist of three parts: Pre-Election Day, Election Day and post-election day. In Denmark, computers have been used in the third part for nearly thirty years. A computer program, that was written by Bjørn Rosengreen, implemts an interpretation of the law and how to tabulate the tallies and assign seats in Folketinget to parties. Because of this, the software became the de facto law interpreting the law and resolving possible ambiguities.

The laws regarding elections are written in legal language, which is often short and to the point. Using legal language for laws results in both advantages and disadvantages. There is the disadvantage that there are situations where the legal texts are not informative enough, for example when it comes to writing computer programs.

An important advantage about legal language - for this project - is that it is closely related to logic. This means that it is possible to look at legal language from a logical point of view. As we are talking about resources, a logic that fits well is linear logic.

With a logical system, it is possible to search for proofs in the legal language and it corresponds, in this setting, with interpreting the law. With this, the law can become source code, and the search for proofs can become an algorithm.

1.2 Goal of the project

> *"The goal of this project is to provide another solution to the chicken egg problem. Instead of taking the law for granted, we argue that we can synthesis the law text from the logical specification. Being able to check a law for soundness and the absence of inconsistency using formal mathematical tools is worth a lot. Synthesizing the law text from a formalized and mechanized set of rules is a billion times better."* (Frank Pfenning 2012)

To achieve this goal, we will be using two tools. The first tool is the linear logical framework CLF, which we will use to capture the essence of the logical rules describing the tabulation algorithms. The second tool is the Grammatical Framework[1], which we will use to linguistic domain specific knowledge.

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

2 Overview

As described in section 1.1, the legal text is short, to the point and not that good for translating directly 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 in legal language: How do we ensure the candidate has reached the quota? What is the quota? What do we do with the ballot that makes him reach the quota even? What happens with him when he is elected? 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.

The legal text do not provide much proof that it is correct either. This is something that will have to be done seperately and is called "searching for proof". Searching for proofs of this law will invariably result in a logical formula. As a running example, I will be using the following decleration from the "Linear Logical Voting Protocols"[1] paper by DeYoung and Schürmann, which describes a candidate reaching the quota.

$$\begin{array}{l}
\text{count/2:} \\
\textit{count-ballots}(S,\ H,\ U) \otimes \\
\textit{uncounted-ballot}(C,\ L) \otimes \textit{hopeful}(C,\ N) \otimes \\
!\textit{quota}(Q) \otimes\ !(N{+}1 \geq Q) \otimes \textit{winners}(W) \otimes \\
!(S\text{-}1 > 0) \\
\quad \multimap \{\ \textit{counted-ballot}(C,\ L) \otimes\ !\textit{elected}(C) \otimes \\
\qquad \textit{winners}([C \mid W]) \otimes \textit{count-ballots}(S\text{-}1,\ H\text{-}1,\ U\text{-}1)\ \}
\end{array}$$

This formula accurately describes the process for checking if a candidate reaches the quota and then marking him as elected.

In contrast, we will provide an example from the Votail[1] system written in Java by Dermot Cochran and Joseph Kiniry.

We will notice that the method operates on another level of abstraction, and requires knowledge regarding the methods and fields used, some of which are declared elsewhere, that the programmer and reader understand each other and the different semantics in Java, such as the keyword final, side effects of the code, shared memory and concurrency.

Java

```java
  /**
* Elect any candidate with a quota or more of votes.
*/
  /*@ requires state == COUNTING;
@ assignable candidateList, ballotsToCount, candidates,
@ numberOfCandidatesElected, totalRemainingSeats;
@ assignable countStatus;
@ ensures countStatus.substate == AbstractCountStatus.CANDIDATE_ELECTED ||
@ countStatus.substate == AbstractCountStatus.SURPLUS_AVAILABLE;
@*/
```

[1]https://github.com/demtech/votail

```
  protected void electCandidatesWithSurplus() {
    while (candidatesWithQuota()
        && countNumberValue < CountConfiguration.MAXCOUNT
        && getNumberContinuing() > totalRemainingSeats) {

      updateCountStatus(AbstractCountStatus.CANDIDATES_HAVE_QUOTA);
      final int winner = findHighestCandidate();

      // Elect highest continuing candidate
      updateCountStatus(AbstractCountStatus.CANDIDATE_ELECTED);
      //@ assert 0 <= winner && winner < totalCandidates;
      //@ assert candidateList[winner].getStatus() == Candidate.CONTINUING;
      //@ assert numberElected < seats;
      //@ assert 0 < remainingSeats;
      /*@ assert (hasQuota(candidateList[winner]))
@ || (winner == findHighestCandidate())
@ || (getNumberContinuing() == totalRemainingSeats);
@*/
      electCandidate(winner);
      if (0 < getSurplus(candidates[winner])) {
        updateCountStatus(AbstractCountStatus.SURPLUS_AVAILABLE);
        distributeSurplus(winner);
      }

    }
  }
```

Java

Furthermore, we will realize that is it a complicated process to translate the code into a natural language because of the afore-mentioned challenges. The main problem is the level of abstraction, as a verbalization system needs to abstract from the implementation details in order to generate readable and informative text.

Comparing the declaritive to the imperative formulation, we will notice that linear logic gives use the tools needed to express that a candidate cannot become unelected (the exclamation mark, which will be described later), and that it is more concise. The Java code assumes that the votes were already counted correctly, where the logical formula gives us a way to express how to count it.

Throughout the project, I will show that it is possible to write a program, in GF, that can translate logical formulas into English. I will make the sentences readable and make sure they make sense. Furthermore, I will make it possible to name the arguments used in the formulas, as well as specify what the arguments used in the formula represents

3 Technologies

In this chapter, I will explain what the linear logic framework, Celf and Grammatical Framework is and how they work.

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 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.).

To explain how linear logic works, we will build a small example of a voter who goes to recieve a ballot.

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.

Simultaneous Conjunction, $\otimes$. Simultaneous conjunction is linear logic's version of the $\wedge$. $A \wedge B$ means "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 resource B are given". To prove it, the resources need to be split into two parts, where one part is used to prove A and the other is used to prove B. 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 $\multimap$ and therefore has precedence. 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 given is consumed, which means the voter must give up a 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, ∀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.

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

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, which is done as described on page 69-72 in the PhD thesis "Implementing Substructural Logical Frameworks"[2] by Schack-Nielsen. The resulting types are the following:

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

3.1.2 Celf

The negative and positive types are all well and good, but while humans can read and understand signs like $\otimes$, $\multimap$ and $\forall$, 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."* [3]

Celf is a framework that can be used to check the correctness of logical formulas and works with linear logic. It does not use the native symbols from linear logic, however, as they are difficult to represent properly. Instead, 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

$\otimes$	::= *
$\multimap$	::= -o
∀x:r	::= Pi x : r

The connectives are not the only things that change from linear logic to Celf. It is possible to use arithmetic operations in formulas written with linear logic (see chapter 2 for an example where arithmetic operations are included), and Celf uses a special syntax for the arithmetic operations as well:

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 chapter 2 will look like the following in Celf's syntax. Note that the formula looks a bit different from the example from the "Linear Logical Voting Protocols"[1], as it has been updated since:

```
count-ballots (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) *
         count-ballots (s S) H U}.
```

This is the syntax Celf reads, however, and it is missing one thing: It says nothing about what the different arguments are, it simply shows that they are there. This can be fixed by parsing the formula into Celf. The output is the following after removing excess parantheses:

```
Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat .
Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : list .
count-ballots (s !(s !S)) (s !H) (s !U) *
uncounted-ballot C L *
hopeful C N *
!quota Q *
!nat-lesseq Q (s !N) *v winners W
   -o {counted-ballot C L *
         !elected C *
         winners (cons !C !W) *
         count-ballots (s !S) H U}.
```

We notice a couple of changes in the formula, the main one being the addition of Pi, which represents the universal quantification. Another thing is that there has been added exclamation marks in the "(cons x y)" and the "(p/s x)" parts. While the latter is not a huge difference, it should be remembered.

Being able to generate formulas similar to the one above through Celf will ensure nothing goes wrong while translating them from linear logic's syntax into a syntax that is easier to to work with. The above is a formula corresponding to the formulas from the "Linear Logical Voting Protocols" paper by DeYoung and Schürmann. The rest can be found in appendix D along with their respective GF version.

This formular, along with the rest in the appendix, are what will be used later for testing the program.

3.2 Grammatical Framework

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

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

3.2.1 The abstract module

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

—— GF ——

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

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

Code 3.1: A simple abstract syntax.

—— GF ——

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

3.2.2 The concrete module

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

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

GF

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

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

One should note that GF comes with built-in resource modules. The module "Prelude" has the same operations as the StringOper module shown.

3.2.4 Variable bindings

The last thing we are going to look at is variable bindings. These are used to bind variables, which allows for them to be used dynamically in parts of the program. In the universal quantification `( Pi x : nat ) ( x = x )`, the variable `x` has a binding (`( Pi x : nat )`) that says it is of the type `nat` (a natural number), and that it is bound in the body `B(x)`, where it can be used.

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

GF

```
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 3.8, 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" ) ;
```

Code 3.8: The concrete syntax for variable bindings.

GF

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

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

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

4 Implementation

In this chapter we will look at how the program has been written along with the thoughts and reasoning behind different choices. We will first look at the abstract syntax tree followed by the concrete implementations. After the implementations have been explained, we will run the program on a formula and discuss the result. Finally we will look at the limitations of the program.

4.1 Constructing the Abstract Syntax Tree

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

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

The fact that we split the connectives of linear logic up in section 3.1.1 will help immensely. The resulting types can be translated almost directly into an abstract tree, which means we have a basic syntax already.

GF

```
abstract Laws = {

    flags startcat = Logic ;

    cat
        Logic ; Neg ; Pos ; Lolli ; Bang ; Atomic ; Conj ; ArgType ; Argument ArgType ;

    fun
        Formular : Neg -> Logic ;

        -- Positive types
        _Atom : Atomic -> Pos ;                         -- Turning an atomic into a positive type
        _Bang : Bang -> Atomic -> Pos ;                 -- Using the unrestricted modality
        _Conj : Pos -> Conj -> Pos -> Pos ;             -- Using the simultaneous conjunction
        _Unit : Neg -> Pos ;                            -- Turns a negative into a positve
        _MPos : Pos -> Pos -> Pos ;                     -- Attaches multiple positives to each other

        -- Negative types
        _Pi : (A : ArgType) -> (Argument A -> Neg) -> Neg ; -- The universal quantification
        _Lolli : Pos -> Lolli -> Neg -> Neg ;           -- Using the linear implication
        _Mon : Pos -> Neg ;                             -- Turning a positive into a negative

        -- Connectives
        _Conj2 : Conj ;                                 -- Simultaneous conjunction
        _Lolli2 : Lolli ;                               -- Linear implication
        _Bang2 : Bang ;                                 -- Unrestricted modality

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

Code 4.1: The first abstract syntax.

GF

Going through the abstract syntax, one will notice a couple of things that do not come from the connectives: There is something called an "Atomic" (line 12) and there is both an ArgType and an Argument A in the function of _Pi. The Atomic is used to represent functions along with the arguments they take. They will be explained in more detail later in this section.

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

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

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

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

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

GF

```
cat
    Atomic ; Ident ; MathFormula ;

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

Code 4.2: Defining the Atomic in the abstract syntax.

GF

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

GF

```
cat
    Ident ; Arg ;

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

```
        Ident_BangElectAll : Ident ;
        Ident_BangDefeatAll : Ident ;
        Ident_DefeatMin : Arg -> Arg -> Arg -> Ident ;
        Ident_DefeatMin' : Arg -> Arg -> Arg -> Ident ;
        Ident_Minimum : Arg -> Arg -> Ident ;
        Ident_Transfer : Arg -> Arg -> Arg -> Arg -> Arg -> Ident ;
        Ident_Run : Arg -> Arg -> Arg -> Ident ;
        Ident_UnitOne : Ident ;

        -- Arguments
        _Arg : (A : ArgType) -> (a : Argument A) -> Arg ;
        _ArgNil, _ArgZ, _Arg1 : Arg ;
        _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 3.1.2, *hopeful C N* has the arguments *C* and *N*. They are the arguments bound through `_Pi`. To turn the variable binding into something that is easier to work with, `_Arg` is used. It takes two arguments and converts them to an `Arg`, which could be the same as the *C* from *hopeful C N*.

In addition to `_Arg`, there are three predefined arguments: `_ArgNil`, `_ArgZ` and `_Arg1`. Each of them represents an argument that has the same meaning no matter what logical formula it is used in. `_ArgNil` is the value ”nil”, `_ArgZ` is the value ”z” or ”zero” and `_Arg1` is the value 1. The last `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 (see appendix D) is represented by an `Identifier`. Each `Identifier` takes between zero and five `Args`, that 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 Maths and a ArithmeticOperation and returns a Math. That way, a _MathArgs can be used inside a _MathArgs to signify multiple operations.

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

That covers the entire abstract syntax tree and should let the reader understand how it is put together and why it is put together in this way.

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 3.1.2), 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 positive 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 the type of the argument next to the argument itself. _Arg therefore only returns the bound variable, which makes it easier for everything else to work with it.

The rest of the 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 hardcoded into the program. Their values will never change, no matter what formulas are being worked with. _ArgPlus, _ArgMinus and _ArgList have been written to use Celf's syntax (see section 3.1.2) and should look familiar.

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

GF

```
    -- 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-greater".

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

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

GF

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

Code 4.7: The linearization of `Ident`

GF

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

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

GF

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

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

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

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

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

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

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

GF

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

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

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

4.2.2 Concrete English implementation

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

GF

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

Code 4.9: The universal quantification in the English implementation

GF

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

GF

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

Code 4.10: `Idents` *and* `Args`

GF

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

Everything else is very similar to the linear logic implementation, but has been translated into proper English. This is not correct for everything, however, as there are something that we do not care for being in English. In picture 4.1, everything in the shaded area has not been changed (with the exception of the `ArgType` ”nat” having been changed into ”natural number”). Everything in the clear area has been formalized.

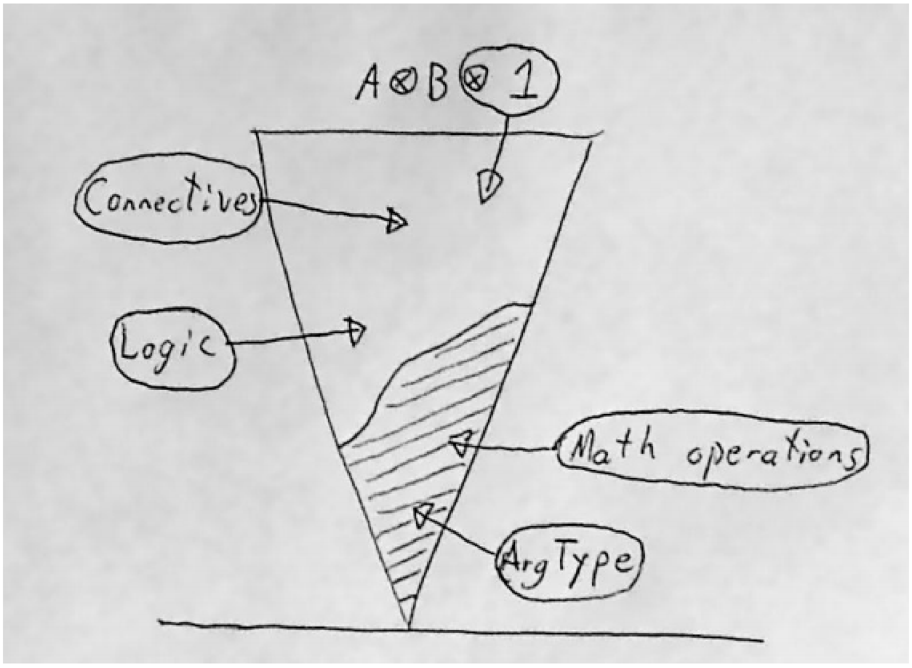

Figure 4.1: Graphical representation of what has been translated into English and what has not.

This covers the important parts in the two concrete implementations, and should give the reader an idea of what the program reads and what the output will be like.

4.3 Running the Program

In this section, I will show the result of parsing a logical formula into the program.

4.3.1 The example

The formula to be run is the one from the example in chapter 2. As the program has been written to accept Celf's output, I will use the version of the formula from the end of section 3.1.2. Due to how GF works, I have had to add extra spaces to the formular, but the reader should be able to verify that it is the same:

```
Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat .
Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : list .
count-ballots ( s ! ( s ! S ) ) ( s ! H ) ( s ! U ) *
uncounted-ballot C L *
hopeful C N *
!quota Q *
!nat-lesseq Q ( s ! N ) *v winners W
   -o {counted-ballot C L *
         !elected C *
         winners ( cons ! C ! W ) *
         count-ballots ( s ! S ) H U}.
```

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

```
p -lang=LawsLin "Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat .
Pi S : nat . Pi U : nat . Pi W : list . count-ballots ( 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 ) * count-ballots ( s ! S
) H U }" | l -treebank
```

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

```
LawsEng:
C is a candidate . H is a natural number .
L is a list . N is a natural number .
Q is a natural number . S is a natural number .
U is a natural number . W is a list .

If we are counting votes and there are ( ( S - 1 ) - 1 ) seats open,
( H - 1 ) hopefuls, and ( U - 1 ) uncounted votes in play , and
there is an uncounted vote with highest preference for candidate C with a list L
of lower preferences , and
candidate C is a hopeful with N votes , and Q votes are needed to be elected , and
( Q is less than or equal to ( N - 1 ) ) ,
and the candidates in the list W have been elected so far
then { there is a counted vote with highest preference for candidate C with a
list L of lower preferences , and
candidate C has been elected , and
the candidates in the list consisting of C and W have been elected so far , and
we are counting votes and there are ( S - 1 ) seats open, H hopefuls, and
U uncounted votes in play }
```

In the output, one will notice that there are cases where a sentence feels a bit odd, for example the last sentence, where it would make more sense if it said that the seats, hopefuls and uncounted votes are what is remaining. This is due to a limitation in the program (more about that in section 4.4.2), which makes the program uses the same sentences for `Idents`, no matter what side of the $\multimap$ they are on.

While there are some limitations, the output is nonetheless understandable and makes sense to some degree.

4.4 Limitations of the program

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

4.4.1 Static identifiers

The identifiers are all static. It was chosen to make them static, as goal of the project was to prove that it was possible to parse and translate a specific voting protocol, which uses specific identifiers. It was therefore not required to make them dynamic.

The static identifiers mean that one can only parse logical formulas written for this voting protocol. If one wants to parse other logical formulas, one will currently have to add identifiers for each identifier in the formula through editing the code. While simple to do, it is not an optimal solution and could potetially be improved upon in the future.

4.4.2 Sentence construction

Like the identifiers, the sentences in the program are static. The only thing that changes is the argument given. Because of this, the sentences may not convey the exact meaning in some cases, which may cause a bit of confusion. It only happens for some sentences and mainly when they are on the right side of the $\multimap$.

This can be solved by making the sentences generic enough to convey their meaning regardless, but they will feel very robotic. With more time, it would be possible to change the program to account for it and modify the sentences to the situation.

4.4.3 Additional languages

With the way the program is built right now, translating it into other languages requires some work. The translator needs to know the other language well, as he has to translate complete sentences properly. With a few identifiers, it is relatively simple, but with more identifiers it can be very time-consuming.

This issue can be soled partially using the grammar resource libraries provided by GF. Each contain operations that can be put together to build sentences. There are libraries for sixteen different languages, and the operations are called the same in each library. Therefore, if the operations are put together properly, all one will have to do, is change which library is used and the sentence should translate into that language instead. It will still be necessary to translate some words, but it will require less knowledge of the language and thereby make it easier to complete a translation.

5 Conclusion

In this thesis, I set out to find a way to turn logical formulas into a natural language. I explained how linear logic works and that it is possible to split the connectives into different types, which makes them easier to work with. I then explained how parts of Grammatical Framework worosk and how they could be used to accomplish certain things.

I then used this knowledge to write a program in Grammatical Framework. First I explained the idea behind the abstract syntax tree, then the concrete implementations. In the program, I made it possible to name arguments and specify what kind of arguments they were (list, candidate or natural number).

Lastly, I showed an example of the output of the program when run on a logical formula, where the result was a meaningful sentence. In appendix D I have the results of every formula for the voting protocol. As the program outputs meaningful sentences for each formula, I have accomplished what I set out to do.

The next step would be to make some improvements for the program. The first improvement would be to get away from the domain specific knowledge. Currently it only works for this specific voting protocol, which needs to be fixed. It could be done in multiple ways, but I will suggest a few. One way would be to change the structure of the program, and make the identifiers dynamic. As the GF code is simple, another way would be to generate the GF code through Celf.

There is also the issue with the static sentences. Making them more dynamic, for example through the use of grammar resource libraries, would improve the quality of the output and make the sentences even more understandable. Use of the grammar resource libraries would also make it easier to translate the output into different languages, another thing that would be a good improvement for the program.

The last thing to improve would be user interaction. The output may say that we are tallying votes, but it does not say how. Giving the program the ability to answer questions from the user (such as "how are the votes tallied?") would make it possible to provide better explanations of how the voting protocols work.

All of this will be reserved for future work, however.

Bibliography

[1] Henry DeYoung and Carsten Schürmann. Linear Logical Voting Protocols, 2011. URL `http://www.cs.cmu.edu/~hdeyoung/papers/voteid11.pdf`. [Online; accessed May 19, 2013].

[2] Anders Schack-Nielsen. Implementing substructural logical frameworks, 2011. URL `http://www.itu.dk/people/anderssn/thesis.pdf`. [Online; accessed May 19, 2013].

[3] Anders Schack-Nielsen and Carsten Schürmann. Short talk: Celf – a logical framework for deductive and concurrent systems.

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 ; Argument 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) -> (Argument A -> Neg) -> Neg ;
        _Lolli : Pos -> Lolli -> Neg -> Neg ;
        _Mon : Pos -> Neg ;

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

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

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

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

        -- Arguments
        _Arg : (A : ArgType) -> (a : Argument A) -> Arg ;
        _ArgNil, _ArgZ : Arg ;
        _ArgMinus, _ArgPlus : Arg -> Arg ;
        _ArgList : Arg -> Arg -> Arg ;

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

        _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 Prelude in {

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

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

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

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

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

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

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

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

        -- Arguments
        _Arg A a                        = ss ( a.s ) ;
        _ArgNil                         = ss ( "nil" ) ;
        _ArgZ                           = ss ( "z" ) ;
        _ArgMinus a                     = ss ( "( s !" ++ a.s ++ ")" ) ;
        _ArgPlus a                      = ss ( "( p !" ++ a.s ++ ")" ) ;
        _ArgList a b                    = ss ( "( cons !" ++ a.s ++ "!" ++ b.s ++ ")" ) ;

        -- 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 ++ ")" ) ;
        _NatDivMod a b c d              = ss ( "!nat-divmod" ++ a.s ++ b.s ++ c.s ++ d.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 B.1: The full concrete implementation for reading linear logic

GF

C Concrete English Implementation

GF

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

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

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

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

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

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

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

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

        -- Identifiers
        Ident_Uncounted a b
            = ss ( "there is an uncounted vote with highest preference for candidate" ++ a.s ++ "with
                      a list" ++ b.s ++ "of lower preferences" ) ;
        Ident_Counted a b
            = ss ( "there is a counted vote with highest preference for candidate" ++ a.s ++ "with
                      a list" ++ b.s ++ "of lower preferences" ) ;
        Ident_Hopeful a b
            = ss ( "candidate" ++ a.s ++ " is a hopeful with" ++ b.s ++ "votes" ) ;
        Ident_Defeated a
            = ss ( "candidate" ++ a.s ++ "has been defeated" ) ;
        Ident_Elected a
            = ss ( "candidate" ++ a.s ++ "has been elected" ) ;
        Ident_Quota a
            = ss ( a.s ++ "votes are needed to be elected" ) ;
        Ident_Winners a
            = ss ( "the candidates in the list" ++ a.s ++ "have been elected so far" ) ;
        Ident_Begin a b c
            = ss ( "we are beginning the tallying and there are" ++ a.s ++ "seats open," ++ b.s
                      ++ " hopefuls, and" ++ c.s ++ "uncounted votes" ) ;
        Ident_Count a b c
            = ss ( "we are counting votes and there are" ++ a.s ++ "seats open," ++ b.s ++ "hopefuls, and"
                      ++ c.s ++ "uncounted votes in play" ) ;
        Ident_BangElectAll
            = ss ( "there are more open seats than hopefuls" ) ;
        Ident_BangDefeatAll
            = ss ( "there are no open seats left" ) ;
        Ident_DefeatMin a b c
            = ss ( "we are in the first stage of determining which candidate has the fewest votes and there
                      are" ++ a.s ++ "seats open, " ++ b.s ++ "hopefuls, and" ++ c.s ++ "potentiel minimums remaining" ) ;
        Ident_DefeatMin' a b c
            = ss ( "we are in the second stage of determining which candidate has the fewest votes and there
                      are" ++ a.s ++ "seats open, " ++ b.s ++ "hopefuls, and" ++ c.s ++ "potentiel minimums remaining" ) ;
        Ident_Minimum a b
            = ss ( "candidate" ++ a.s ++ "'s with a count of" ++ b.s ++ "votes is a potential minimum" ) ;
        Ident_Transfer a b c d e
            = ss ( "the candidate" ++ a.s ++"'s" ++ b.s ++ "votes are being tranferred and there are" ++ c.s
                      ++ "open seats," ++ d.s ++ "hopeful candidates and" ++ e.s ++ "uncounted votes" ) ;
        Ident_Run a b c
            = ss ( "we are tallying votes" ) ;
        Ident_UnitOne
            = ss ( "consume the corresponding resources" ) ;

        -- Arguments
        _Arg A a                        = ss ( a.s ) ;
        _ArgNil                         = ss ( "- that is empty -" ) ;
        _ArgZ                           = ss ( "zero" ) ;
        _ArgMinus a                     = ss ( "(" ++ a.s ++ "- 1 )" ) ;
        _ArgPlus a                      = ss ( "(" ++ a.s ++ "+ 1 )" ) ;
        _ArgList a b                    = ss ( "consisting of" ++ a.s ++ "and" ++ b.s ) ;

        -- Mathematic operations
        _MathArg arg1                   = ss ( arg1.s ) ;
        _FinalFormula m1 ms m2          = ss ( "(" ++ m1.s ++ ms.s ++ m2.s ++ ")" ) ;
        _MathArgs arg1 mo arg2          = ss ( "(" ++ arg1.s ++ mo.s ++ arg2.s ++ ")" ) ;
        _NatDivMod a b c d              = ss ( "(" ++ a.s ++ "=" ++ b.s ++ "*" ++ c.s ++ "+" ++ d.s ++ ")" ) ;

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

        -- Inequality operations
        _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

D Original Celf Formulas

Below are the Celf version of the logical formulas from the "Linear Logical Voting Protocols"[1] paper by DeYoung and Schürmann. The formulas have been given to me by Carsten Schürmann.

Further down is the output from running these formulas through Celf.

```
count/1 : count-ballots S H (s U) *
          uncounted-ballot C L *
          hopeful C N *
          !quota Q *
          !nat-less (s N) Q
            -o {counted-ballot C L *
                hopeful C (s N) *
                count-ballots S H U}.

count/2 : count-ballots (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) *
                count-ballots (s S) H U}.

count/3 : count-ballots (s z) H 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) *
                !defeat-all}.

count/4_1 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !elected C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/4_2 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !defeated C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/5_1 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !elected C
              -o {count-ballots S H U}.

count/5_2 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !defeated C
              -o {count-ballots S H U}.
```

```
count/6 : count-ballots S H z
            -o {defeat-min S H z}.

defeat-min/1 : defeat-min S (s H) M *
               hopeful C N
                 -o {minimum C N *
                     defeat-min S H (s M)}.

defeat-min/2 : defeat-min S z M
                 -o {defeat-min' S z M}.

defeat-min'/1 : 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}.

defeat-min'/2 : defeat-min' S H (s z) *
                minimum C N
                  -o {!defeated C *
                      transfer C N S H z}.

transfer/1 : transfer C (s N) S H U *
             counted-ballot C (cons C' L)
               -o {uncounted-ballot C' L *
                   transfer C N S H (s U)}.

transfer/2 : transfer C (s N) S H U *
             counted-ballot C nil
               -o {transfer C N S H U}.

transfer/3 : transfer C z S H U *
             !nat-less S H
               -o {count-ballots S H U}.

transfer/4 : transfer C z S H U *
             !nat-lesseq H S
               -o {!elect-all}.

defeat-all/1 : !defeat-all *
               hopeful C N
                 -o {!defeated C}.

elect-all/1 : !elect-all *
              hopeful C N *
              winners W
                -o {!elected C *
                    winners (cons C W)}.

cleanup/1 : !defeat-all *
            uncounted-ballot C L
              -o {1}.

cleanup/2 : !defeat-all *
            counted-ballot C L
              -o {1}.

cleanup/3 : !elect-all *
            uncounted-ballot C L
              -o {1}.
```

```
cleanup/4 : !elect-all *
            counted-ballot C L
              -o {1}.

run/1 : run S H U *
        !nat-divmod U (s S) Q _
          -o {!quota (s Q) *
              winners nil *
              count-ballots S H U}.
```

Code D.1: The original formulas in Celf notation.

```
[GF:count/1: Pi C: candidate. Pi H: nat. Pi L: list. Pi N: nat. Pi Q: nat. Pi S: nat. Pi U: nat.
        (count-ballots S H (s !U) *
        (uncounted-ballot C L *
        (hopeful C N *
        (!quota Q *
        !nat-less
        (s !N) Q))))
                -o {counted-ballot C L *
                        (hopeful C (s !N) *
                        count-ballots S H U)}].

[GF:count/2: Pi C: candidate. Pi H: nat. Pi L: list. Pi N: nat. Pi Q: nat. Pi S: nat. Pi U: nat.
Pi W: list.
        (count-ballots (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) *
                        count-ballots (s !S) H U))}].

[GF:count/3: Pi C: candidate. Pi H: nat. Pi L: list. Pi N: nat. Pi Q: nat. Pi U: nat. Pi W: list.
        (count-ballots (s !z) H 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) *
                        !defeat-all))}].

[GF:count/4_1: Pi C: candidate. Pi C’: candidate. Pi H: nat. Pi L: list. Pi S: nat. Pi U: nat.
        (count-ballots S H U *
        (uncounted-ballot C (cons !C’ !L) *
        !elected C))
                -o {uncounted-ballot C’ L *
                        count-ballots S H U}].

[GF:count/4_2: Pi C: candidate. Pi C’: candidate. Pi H: nat. Pi L: list. Pi S: nat. Pi U: nat.
        (count-ballots S H U *
        (uncounted-ballot C (cons !C’ !L) *
        !defeated C))
                -o {uncounted-ballot  C’ L *
                        count-ballots S H U}].

[GF:count/5_1: Pi C: candidate. Pi H: nat. Pi S: nat. Pi U: nat.
        (count-ballots S H (s !U) *
        (uncounted-ballot C nil *
        !elected C))
                -o {count-ballots S H U}].

[GF:count/5_2: Pi C: candidate. Pi H: nat. Pi S: nat. Pi U: nat.
        (count-ballots S H (s !U) *
        (uncounted-ballot C nil *
        !defeated C))
                -o {count-ballots S H U}].

[GF:count/6: Pi H: nat. Pi S: nat.
        count-ballots S H z
                -o {defeat-min S H z}].

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

[GF:defeat-min/2: Pi M: nat. Pi S: nat.
        defeat-min S z M
                -o {defeat-min' S z M}].

[GF:defeat-min'/1: Pi C: candidate. Pi C': candidate. 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)}].

[GF:defeat-min'/2: Pi C: candidate. Pi H: nat. Pi N: nat. Pi S: nat.
        (defeat-min' S H (s !z) *
        minimum C N)
                -o {!defeated C *
                        transfer C N S H z}].

[GF:transfer/1: Pi C: candidate. Pi C': candidate. Pi H: nat. Pi L: list. Pi N: nat.
Pi S: nat. Pi U: nat.
        (transfer C (s !N) S H U *
        counted-ballot C (cons !C' !L))
                -o {uncounted-ballot C' L *
                        transfer C N S H (s !U)}].

[GF:transfer/2: Pi C: candidate. Pi H: nat. Pi N: nat. Pi S: nat. Pi U: nat.
        (transfer C (s !N) S H U *
        counted-ballot C nil)
                -o {transfer C N S H U}].

[GF:transfer/3: Pi C: candidate. Pi H: nat. Pi S: nat. Pi U: nat.
        (transfer C z S H U *
        !nat-less S H)
                -o {count-ballots S H U}].

[GF:transfer/4: Pi C: candidate. Pi H: nat. Pi S: nat. Pi U: nat.
        (transfer C z S H U *
        !nat-lesseq H S)
                -o {!elect-all}].

[GF:defeat-all/1: Pi C: candidate. Pi N: nat.
        (!defeat-all *
        hopeful C N)
                -o {!defeated C}].

[GF:elect-all/1: Pi C: candidate. Pi N: nat. Pi W: list.
        (!elect-all *
        (hopeful C N *
        winners W))
                -o {!elected C *
                        winners (cons !C !W)}].

[GF:cleanup/1: Pi C: candidate. Pi L: list. (!defeat-all *
        uncounted-ballot C L)
                -o {1}].

[GF:cleanup/2: Pi C: candidate. Pi L: list. (!defeat-all *
        counted-ballot C L)
                -o {1}].

[GF:cleanup/3: Pi C: candidate. Pi L: list. (!elect-all *
        uncounted-ballot C L)
                -o {1}].

[GF:cleanup/4: Pi C: candidate. Pi L: list. (!elect-all *
        counted-ballot C L)
                -o {1}].

[GF:run/1: Pi H: nat. Pi Q: nat. Pi S: nat. Pi U: nat. Pi xx3: nat.
        (run S H U *
        !nat-divmod U (s !S) Q xx3)
                -o {!quota (s !Q) *
                        (winners nil *
                        count-ballots S H U)}].
```

Code D.2: The formulas from Code D.1 after bring run through Celf and parsed into GF formulas.

E Tests

Below are each of the formulas for the voting protocol along with the output from the GF program.

E.1 count/1

E.1.1 Logical formula

Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat . Pi S : nat . Pi U : nat . count-ballots S H (s ! U) * uncounted-ballot C L * hopeful C N * !quota Q * !nat-less (s ! N) Q -o { counted-ballot C L * hopeful C (s ! N) * count-ballots S H U }

E.1.2 Result

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

E.2 count/2

E.2.1 Logical formula

Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat . Pi S : nat . Pi U : nat . Pi W : list . count-ballots (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) * count-ballots (s ! S) H U }

E.2.2 Result

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

E.3 count/3

E.3.1 Logical formula

Pi C : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi Q : nat . Pi U : nat . Pi W : list . count-ballots (s ! z) H 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) * !defeat-all }

E.3.2 Result

C is a candidate . H is a natural number . L is a list . N is a natural number . Q is a natural number . U is a natural number . W is a list . If we are counting votes and there are (zero - 1) seats open, H hopefuls, and U uncounted votes in play , and there is an uncounted vote with highest preference for candidate C with a list L of lower preferences , and candidate C is a hopeful with N votes , and Q votes are needed to be elected , and (Q is less than or equal to (N - 1)) , and the candidates in the list W have been elected so far then { there is a counted vote with highest preference for candidate C with a list L of lower preferences ,

E.4 count/4_1

E.4.1 Logical formula

Pi C : candidate . Pi C' : candidate . Pi H : nat . Pi L : list . Pi S : nat . Pi U : nat . count-ballots S H U * uncounted-ballot C (cons ! C' ! L) * !elected C -o { uncounted-ballot C' L * count-ballots S H U }

E.4.2 Result

C is a candidate . C' is a candidate . H is a natural number . L is a list . S is a natural number . U is a natural number . If we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play , and there is an uncounted vote with highest preference for candidate C with a list consisting of C' and L of lower preferences , and candidate C has been elected then { there is an uncounted vote with highest preference for candidate C' with a list L of lower preferences , and we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }

E.5 count/4_2

E.5.1 Logical formula

Pi C : candidate . Pi C' : candidate . Pi H : nat . Pi L : list . Pi S : nat . Pi U : nat . count-ballots S H U * uncounted-ballot C (cons ! C' ! L) * !defeated C -o { uncounted-ballot C' L * count-ballots S H U }

E.5.2 Result

C is a candidate . C' is a candidate . H is a natural number . L is a list . S is a natural number . U is a natural number . If we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play , and there is an uncounted vote with highest preference for candidate C with a list consisting of C' and L of lower preferences , and candidate C has been defeated then { there is an uncounted vote with highest preference for candidate C' with a list L of lower preferences , and we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }

E.6 count/5_1

E.6.1 Logical formula

Pi C : candidate . Pi H : nat . Pi S : nat . Pi U : nat . count-ballots S H (s ! U) * uncounted-ballot C nil * !elected C -o { count-ballots S H U }

E.6.2 Result

C is a candidate . H is a natural number . S is a natural number . U is a natural number . If we are counting votes and there are S seats open, H hopefuls, and (U - 1) uncounted votes in play , and there is an uncounted vote with highest preference for candidate C with a list - that is empty - of lower preferences , and candidate C has been elected then { we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }

E.7 count/5_2

E.7.1 Logical formula

Pi C : candidate . Pi H : nat . Pi S : nat . Pi U : nat . count-ballots S H (s ! U) * uncounted-ballot C nil * !defeated C -o { count-ballots S H U }

E.7.2 Result

C is a candidate . H is a natural number . S is a natural number . U is a natural number . If we are counting votes and there are S seats open, H hopefuls, and (U - 1) uncounted votes in play , and there is an uncounted vote with highest preference for candidate C with a list - that is empty - of lower preferences , and candidate C has been defeated then { we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }

E.8 count/6

E.8.1 Logical formula

Pi H : nat . Pi S : nat . count-ballots S H z -o { defeat-min S H z }

E.8.2 Result

H is a natural number . S is a natural number . If we are counting votes and there are S seats open, H hopefuls, and zero uncounted votes in play then { we are in the first stage of determining which candidate has the fewest votes and there are S seats open, H hopefuls, and zero potentiel minimums remaining }

E.9 defeat-min/1

E.9.1 Logical formula

Pi C : candidate . Pi H : nat . Pi M : nat . Pi N : nat . Pi S : nat . defeat-min S (s ! H) M * hopeful C N -o { minimum C N * defeat-min S H (s ! M) }

E.9.2 Result

C is a candidate . H is a natural number . M is a natural number . N is a natural number . S is a natural number . If we are in the first stage of determining which candidate has the fewest votes and there are S seats open, (H - 1) hopefuls, and M potentiel minimums remaining , and candidate C is a hopeful with N votes then { candidate C 's with a count of N votes is a potential minimum , and we are in the first stage of determining which candidate has the fewest votes and there are S seats open, H hopefuls, and (M - 1) potentiel minimums remaining }

E.10 defeat-min/2

E.10.1 Logical formula

Pi M : nat . Pi S : nat . defeat-min S z M -o { defeat-min' S z M }

E.10.2 Result

M is a natural number . S is a natural number . If we are in the first stage of determining which candidate has the fewest votes and there are S seats open, zero hopefuls, and M potentiel minimums remaining then { we are in the second stage of determining which candidate has the fewest votes and there are S seats open, zero hopefuls, and M potentiel minimums remaining }

E.11 defeat-min'/1

E.11.1 Logical formula

Pi C : candidate . Pi C' : candidate . 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 }

E.11.2 Result

C is a candidate . C' is a candidate . H is a natural number . M is a natural number . N is a natural number . N' is a natural number . S is a natural number . If we are in the second stage of determining which candidate has the fewest votes and there are S seats open, H hopefuls, and (M - 1) potentiel minimums remaining , and candidate C 's with a count of N votes is a potential minimum , and candidate C' 's with a count of N' votes is a potential minimum , and (N is less than N') then { candidate C 's with a count of N votes is a potential minimum , and candidate C' is a hopeful with N' votes , and we are in the second stage of determining which candidate has the fewest votes and there are S seats open, (H - 1) hopefuls, and M potentiel minimums remaining }

E.12 defeat-min'/2

E.12.1 Logical formula

Pi C : candidate . Pi H : nat . Pi N : nat . Pi S : nat . defeat-min' S H (s ! z) * minimum C N -o { !defeated C * transfer C N S H z }

E.12.2 Result

C is a candidate . H is a natural number . N is a natural number . S is a natural number . If we are in the second stage of determining which candidate has the fewest votes and there are S seats open, H hopefuls, and (zero - 1) potentiel minimums remaining , and candidate C 's with a count of N votes is a potential minimum then { candidate C has been defeated , and the candidate C 's N votes are being tranferred and there are S open seats, H hopeful candidates and zero uncounted votes }

E.13 transfer/1

E.13.1 Logical formula

Pi C : candidate . Pi C' : candidate . Pi H : nat . Pi L : list . Pi N : nat . Pi S : nat . Pi U : nat . transfer C (s ! N) S H U * counted-ballot C (cons ! C' ! L) -o { uncounted-ballot C' L * transfer C N S H (s ! U) }

E.13.2 Result

C is a candidate . C' is a candidate . H is a natural number . L is a list . N is a natural number . S is a natural number . U is a natural number . If the candidate C 's (N - 1) votes are being tranferred and there are S open seats, H hopeful candidates and U uncounted votes , and there is a counted vote with highest preference for candidate C with a list consisting of C' and L of lower preferences then { there is an uncounted vote with highest preference for candidate C' with a list L of lower preferences , and the candidate C 's N votes are being tranferred and there are S open seats, H hopeful candidates and (U - 1) uncounted votes }

E.14 transfer/2

E.14.1 Logical formula

Pi C : candidate . Pi H : nat . Pi N : nat . Pi S : nat . Pi U : nat . transfer C (s ! N) S H U * counted-ballot C nil -o { transfer C N S H U }

E.14.2 Result

C is a candidate . H is a natural number . N is a natural number . S is a natural number . U is a natural number . If the candidate C 's (N - 1) votes are being tranferred and there are S open seats, H hopeful candidates and U uncounted votes , and there is a counted vote with highest preference for candidate C with a list - that is empty - of lower preferences then { the candidate C 's N votes are being tranferred and there are S open seats, H hopeful candidates and U uncounted votes }

E.15 transfer/3

E.15.1 Logical formula

Pi C : candidate . Pi H : nat . Pi S : nat . Pi U : nat . transfer C z S H U * !nat-less S H -o { count-ballots S H U }

E.15.2 Result

C is a candidate . H is a natural number . S is a natural number . U is a natural number . If the candidate C 's zero votes are being tranferred and there are S open seats, H hopeful candidates and U uncounted votes , and (S is less than H) then { we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }

E.16 transfer/4

E.16.1 Logical formula

Pi C : candidate . Pi H : nat . Pi S : nat . Pi U : nat . transfer C z S H U * !nat-lesseq H S -o { !elect-all }

E.16.2 Result

C is a candidate . H is a natural number . S is a natural number . U is a natural number . If the candidate C 's zero votes are being tranferred and there are S open seats, H hopeful candidates and U uncounted votes , and (H is less than or equal to S) then { there are more open seats than hopefuls }

E.17 defeat-all/1

E.17.1 Logical formula

Pi C : candidate . Pi N : nat . !defeat-all * hopeful C N -o { !defeated C }

E.17.2 Result

C is a candidate . N is a natural number . If there are no open seats left , and candidate C is a hopeful with N votes then { candidate C has been defeated }

E.18 elect-all/1

E.18.1 Logical formula

Pi C : candidate . Pi N : nat . Pi W : list . !elect-all * hopeful C N * winners W -o { !elected C * winners (cons ! C ! W) }

E.18.2 Result

C is a candidate . N is a natural number . W is a list . If there are more open seats than hopefuls , and candidate C is a hopeful with N votes , and the candidates in the list W have been elected so far then { candidate C has been elected , and the candidates in the list consisting of C and W have been elected so far }

E.19 cleanup/1

E.19.1 Logical formula

Pi C : candidate . Pi L : list . !defeat-all * uncounted-ballot C L -o { 1 }

E.19.2 Result

C is a candidate . L is a list . If there are no open seats left , and there is an uncounted vote with highest preference for candidate C with a list L of lower preferences then { consume the corresponding resources }

E.20 cleanup/2

E.20.1 Logical formula

Pi C : candidate . Pi L : list . !defeat-all * counted-ballot C L -o { 1 }

E.20.2 Result

C is a candidate . L is a list . If there are no open seats left , and there is a counted vote with highest preference for candidate C with a list L of lower preferences then { consume the corresponding resources }

E.21 cleanup/3

E.21.1 Logical formula

Pi C : candidate . Pi L : list . !elect-all * uncounted-ballot C L -o { 1 }

E.21.2 Result

C is a candidate . L is a list . If there are more open seats than hopefuls , and there is an uncounted vote with highest preference for candidate C with a list L of lower preferences then { consume the corresponding resources }

E.22 cleanup/4

E.22.1 Logical formula

Pi C : candidate . Pi L : list . !elect-all * counted-ballot C L -o { 1 }

E.22.2 Result

C is a candidate . L is a list . If there are more open seats than hopefuls , and there is a counted vote with highest preference for candidate C with a list L of lower preferences then { consume the corresponding resources }

E.23 run/1

E.23.1 Logical formula

Pi H : nat . Pi Q : nat . Pi S : nat . Pi U : nat . Pi xx3 : nat . run S H U * !nat-divmod U (s ! S) Q xx3 -o { !quota (s ! Q) * winners nil * count-ballots S H U }

E.23.2 Result

H is a natural number . Q is a natural number . S is a natural number . U is a natural number . xx3 is a natural number . If we are tallying votes , and (U = (S - 1) * Q + xx3) then { (Q - 1) votes are needed to be elected , and the candidates in the list - that is empty - have been elected so far , and we are counting votes and there are S seats open, H hopefuls, and U uncounted votes in play }