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 a lot of laws describing how votes are to be distributed. These laws are often short and to the point, which can be both good and bad. It is good because they are not hard to read. It is bad because 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):

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

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 is 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.

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

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

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

Adding the connectives together gives the following. **Note: Is the "A ::=" and "∀x:A" part correct?**

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

3.1.2 Splitting the connectives

The type above immediately poses 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, the derivation of the simultaneous conjunction is the following:

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

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

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

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

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

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

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

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

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

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

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. Furthermore, some operations (such as minus and equality operators) are different in Celf as well. The differences are the following:

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

This can be used to translate linear logic formulas into something a program 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

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.

```
abstract Hello = {
    flags startcat = Greeting ;

    cat
        Recipient ; Greeting ;
    fun
        World, Friend, Enemy : Recipient ;
        Hello : Recipient -> Greeting ;
}
```

Code 3.1: A simple abstract syntax.

In the abstract syntax shown in Code 3.1, there are two categories: `Greeting` and `Recipient`. Furthermore, there are two functions. The first function determines that `World`, `Mum` and `Friends` are considered `Recipients`. The second function determines that `Hello` takes a `Recipient` and returns a `Greeting`.

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. Using the abstract syntax, the program turn the abstract syntax into an actual language through a concrete implementation.

```
concrete HelloEng of Hello = {
    lincat
        Recipient, Greeting = {s : Str} ;
    lin
        World = {s = ”world”} ;
        Friend = {s = ”friend”} ;
        Enemy = {s = ”enemy”} ;
        Hello recip = {s = ”hello” ++ recip.s} ;
}
```

Code 3.2: A concrete English implementation of the abstract syntax from Code 3.1.

In Code 3.1 both `Greeting` and `Recipient` have been defined as records that can hold a `Str` (a string). `World`, `Mum` and `Friends` each have simple meanings. `Hello` has a more advanced meaning, however, as it consists of the string `"hello"` concatenated with the value of `recip` (the `Recipient` it takes as an argument).

Note: Remember to write that the code comes from http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html#toc9

```
concrete HelloIta of Hello = {
    lincat
        Recipient , Greeting = {s : Str} ;
    lin
        World = {s = ”mondo”} ;
        Friend = {s = ”amico”} ;
        Enemy = {s = ”nemico”} ;
        Hello recip = {s = ”ciao” ++ recip.s} ;
}
```

Code 3.3: A concrete Italian implementation of the abstract syntax from Code 3.1.

With another concrete implementation (such as the Italian one in Code 3.3), the program will be able to translate the simple sentences from one language into the other.

Another feature of GF is operations (`oper`), also known as functions in other programming languages.

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

Code 3.4: A simple operation in GF.

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.

```
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.

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

```
concrete HelloEng of Hello = open StringOper in {
    lincat
        Recipient, Greeting = SS ;
    lin
        World = ss (”world”) ;
        Friend = ss (”friend”) ;
        Enemy = ss (”enemy”) ;
        Hello recip = ss (”hello” ++ recip.s) ;
}
```

Code 3.6: Using the resource module.

GF also features parameters that can be used to change words. 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).

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

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

The `lincat` of a type also needs to reflect the fact that it takes a parameter. This is done as shown in 3.8.

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

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

The last thing that needs to change, 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 3.9 shows a table for Friend.

```
lin
    Friend = {
        s = table {
            Sg => ”friend” ;
            Pl => ”friends”
        }
    } ;
```

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

Code 3.10 shows how the parameters are used in the Hello program. Assuming Friend is the recip, line 1 creates a string saying ”hello friend” where line 2 creates a string saying ”hello friends”.

```
1 lin
2     HelloSg recip = {s = ”hello” ++ recip.s ! Sg} ;
3     HelloPl recip = {s = ”hello” ++ recip.s ! Pl} ;
```

Code 3.10: Giving the parameter as an argument.

Note: Need some sort of ending here.

Bibliography

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