Laws and Logic

Bachelor Project

Bachelor in Software Development,
IT-University of Copenhagen

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

FIX DATE

Contents

1 Introduction

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

The goal of the project was to write a program using Grammatical Framework[1] that could translate formulas written in linear logic into a natrual language. The idea behind the program was to translate logical formulas describing the laws about voting protocols into sentences more descriptive than the current ones. (?)

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. (?)

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. It is also possible to use the formulas as source code directly.

1.2 NEEDS TITLE (problemformulering)

All of the above does not change the original issue; the legal text is the same and is still translated wrongly into code. This could be avoided if the legal text was changed, for example to a textified(?) version of the logical formulas mentioned above.

This project will explore the possibilities of using Grammatical Framework[2] to translate linear logic into understandable sentences in one or more natrual languages, for the purpose of using them as law texts.

See if it is possible to do this

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

2 Technical Description

2.1 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.).

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

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

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

2.2 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.

```
1 abstract Hello = {
2     flags startcat = Greeting ;
3
4     cat
5         Recipient ; Greeting ;
6     fun
7         World, Mum, Friends : Recipient ;
8         Hello : Recipient -> Greeting ;
9 }
```

Code 2.1: A simple abstract syntax.

In the abstract syntax shown in Code 2.1 , 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.

```
1 concrete HelloEng of Hello = {
2     lincat
3         Recipient, Greeting = {s : Str} ;
4     lin
5         World = {s = "world"} ;
6         Mum = {s = "mum"} ;
7         Friends = {s = "friends"} ;
8         Hello recip = {s = "hello" ++ recip.s} ;
9 }
```

Code 2.2: A concrete English implementation of the abstract syntax from Code 2.1 .

In Code 2.1 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).

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"} ;
        Mum = {s = "mamma"} ;
        Friends = {s = "amici"} ;
        Hello recip = {s = "ciao" ++ recip.s} ;
}
```

Code 2.3: A concrete Italian implementation of the abstract syntax from Code 2.1 .

With another concrete implementation (such as the Italian one in Code 2.3), the program will be able to translate the simple sentences from one language into the other.

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

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

Code 2.4: 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 2.4 has the name "cc" (concatenate).
- **Arguments, their types and the return type.** The operation in Code 2.4 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 2.4 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 2.5: 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 2.7), which can be accessed by multiple concrete implementations.

How to access the operations!
Params

```
concrete HelloIta of Hello = {
    lincat
        Recipient, Greeting = {s : Str} ;
    lin
        World = {s = "mondo"} ;
        Mum = {s = "mamma"} ;
        Friends = {s = "amici"} ;
        Hello recip = {s = "ciao" ++ recip.s} ;
}
```

Code 2.6: A simple resource module.

GF

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

    cat
        Recipient ; Greeting ;
    fun
        World, Mum, Friends : Recipient ;
        Hello : Recipient -> Greeting ;
}
```

Code 2.7: A simple resource module.

GF