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] http://www.grammaticalframework.org/

2 Project Overview

2.1 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.

2.2 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[1] to translate linear logic into understandable sentences in one or more natrual languages, for the purpose of using them as law texts.

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

3 Problem Analysis

4 Technical Description

4.1 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 4.1: A simple abstract syntax.

In the abstract syntax shown in Code 4.1, 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. From the abstract syntax, the program can generate another language, assuming it is a concrete implementation of the same abstract syntax.

```
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 4.2: A concrete English implementation of the abstract syntax from Code 4.1.

In Code 4.1 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

```
1 concrete HelloIta of Hello = {
2     lincat
3         Recipient, Greeting = {s : Str} ;
4     lin
5         World = {s = "mondo"} ;
6         Mum = {s = "mamma"} ;
7         Friends = {s = "amici"} ;
8         Hello recip = {s = "ciao" ++ recip.s} ;
9 }
```

Code 4.3: A concrete Italian implementation of the abstract syntax from Code 4.1.

With another concrete implementation (such as the Italian one in Code 4.3), the program will be able to translate the simple sentences from one language into the other.

5 Testing

6 Conclusion