DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

ON THE GRAMMAR OF PROOF

Warrick Macmillan

Master's Thesis:	30 credits
Programme:	Master's Programme in Language Technology
Level:	Advanced level
Semester and year:	Fall, 2021
Supervisor	Aarne Ranta
Examiner	(name of the examiner)
Report number	(number will be provided by the administrators)
Keywords	Grammatical Framework, Natural Language Generation,

Abstract

Brief summary of research question, background, method, results…

Preface

Acknowledgements, etc.

Contents

1803 Addsection numbers to sections

Preliminaries

We give brief but relevant overviews of the background ideas and tools that went into the generation of this thesis.

Agda

Overview

Agda is an attempt to faithfully formalize Martin-Löf's intensional type theory [3]. Referencing our previous distinction, one can think of Martin-Löf's original work as a specification, and Agda as one possible implementation.

Agda is a functionaly programming language which, through an interactive environment, allows one to iteratively apply rules and develop constructive mathematics. It's current incarnation, Agda2 (but just called Agda), was preceded by ALF, Cayenne, and Alfa, and the Agda1. On top of the basic MLTT, Agda incorporates dependent records, inductive definitions, pattern matching, a versatile module system, and a myriad of other bells and whistles which are of interest generally and in various states of development but not relevant to this work.

For our purposes, we will only look at what can in some sense be seen as the kernel of Agda. Developing a full-blown GF grammar to incorporate more advanced Agda features would require efforts beyond the scope of this work.

Agda's purpose is to manifest the propositions-as-types paradigm in a practical and useable programming language. And while there are still many reasons one may wish to use other programming languages, or just pen and paper to do her work, there is a sense of purity one gets when writing Agda code. There are many good resources for learning Agda [1] [4] [2] [5] so we'll only give a cursory overview of what is relevant for this thesis, with a particular emphasis on the syntax.

Formalizing The Twin Prime Conjecture

Inspired by Escardos's formalization of the twin primes conjecture [**?**], we intend to demonstrate that while formalizing mathematics can be rewarding, it can also create immense difficulties, especially if one wishes to do it in a way that prioritizes natural language. The conjecture is incredibly compact

Lemma 1 *There are infinitely many twin primes.*

Somebody reading for the first time might then pose the immediate question : what is a twin prime?

Definition 1 *A* twin prime *is a prime number that is either 2 less or 2 more than another prime number*

Below Escardo's code is reproduced.

```
isPrime : ℕ → Set
isPrime n =
  (n ≥ 2) ×
  ((x y : ℕ) → x * y ≡ n → (x ≡ 1) + (x ≡ n))

twinPrimeConjecture : Set
twinPrimeConjecture = (n : ℕ) → Σ[ p ∈ ℕ ] (p ≥ n)
  × isPrime p
  × isPrime (p ∔ 2)
```

We note there are some both subtle and big differences, between the natural language claim. First, twin prime is defined implicitly via a product expression, ×. Additionally, the "either 2 less or 2 more" clause is oringially read as being interpreted as having "2 more". This reading ignores the symmetry of products, however, and both "p or (p ∔ 2)" could be interpreted as the twin prime. This phenomenon makes translation highly nontrivial; however, we will later see that PGF is capable of adding a semantic layer where the theorem can be evaluated during the translation. Finally, this theorem doesn't say what it is to be infinite in general, because such a definition would require a proving a bijection with the real numbers. In this case however, we can rely on the order of the natural numbers, to simply state what it means to have infinitely many primes.

Despite the beauty of this, mathematicians always look for alternative, more general ways of stating things. Generalizing the notion of a twin prime is a prime gap. And then one immediately has to ask what is a prime gap?

Definition 2 *A* twin prime *is a prime that has a prime gap of two.*

Definition 3 *A* prime gap *is the difference between two successive prime numbers.*

Now we're stuck, at least if you want to scour the internet for the definition of "two successive prime numbers". That is because any mathematician will take for granted what it means, and it would be considered a waste of time and space to include something *everyone* alternatively knows. Agda, however, must know in order to typecheck. Below we offer a presentation which suits Agda's needs, and matches the number theorists presentation of twin prime.

```
isSuccessivePrime : (p p′ : ℕ) → isPrime p → isPrime p′ → Set
isSuccessivePrime p p′ x x₁ =
  (p″ : ℕ) → (isPrime p″) →
  p ≤ p′ → p ≤ p″ → p′ ≤ p″

primeGap :
  (p p′ : ℕ) (pIsPrime : isPrime p) (p′IsPrime : isPrime p′) →
```

```
  (isSuccessivePrime p p' pIsPrime p'IsPrime) →
  ℕ
primeGap p p' pIsPrime p'IsPrime p'-is-after-p = p - p'

twinPrime : (p : ℕ) → Set
twinPrime p =
  (pIsPrime : isPrime p) (p' : ℕ) (p'IsPrime : isPrime p')
  (p'-is-after-p : isSuccessivePrime p p' pIsPrime p'IsPrime) →
  (primeGap p p' pIsPrime p'IsPrime p'-is-after-p) ≡ 2

twinPrimeConjecture' : Set
twinPrimeConjecture' = (n : ℕ) → Σ[ p ∈ ℕ ] (p ≥ n)
  × twinPrime p
```

We see that `isSuccessivePrime` captures this meaning, interpreting "successive" as the type of suprema in the prime number ordering.

Defining a prime gap, the term `primeGap`, then has to reference this data, even though most of it is discarded and unused in the actual program. One could keep this data around via extra record fields, to facilitate with the next definition, but ultimately the developer has to decide what is relevant. We also use propositional equality here, which is another departure from classical mathematics, as will be elaborated later.

Finally, {twinPrime is a specialized version of `primeGap` to 2. "has a prime gap of two" needs to be interpreted "whose prime gap is equal to two", and writing a GF grammar capable of disambiguating *has* in mathematics generally is likely impossible.

While working on this example, I tried to prove that 2 is prime in Agda, which turned out to be nontrivial. When I told this to an analyst, he remarked that couldn't possibly be the case because it's something which a simple algorithm can compute (or generate). This exchange was incredibly stimulating, for the mathematian didn't know about the *propositions as types* principle, and was simply taking for granted his internal computational capacity to confuse it for proof, especially in a constructive setting. He also seemed perplexed that anyone would find it interesting to prove that 2 is prime. As is hopefully revealed by this discussion, seemingly trivial things, when treated by the type theorist or linguist, can become wonderful areas of exploration.

Natural Language and Mathematics

References

[1] Bove, A. & Dybjer, P. (2009). *Dependent Types at Work*, (pp. 57–99). Springer Berlin Heidelberg: Berlin, Heidelberg.

[2] Bove, A., Dybjer, P., & Norell, U. (2009). A brief overview of agda – a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, & M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics* (pp. 73–78). Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] Howard, W. A. (1986). Per martin-löf. intuitionistic type theory. (notes by giovanni sambin of a series of lectures given in padua, june 1980.) studies in proof theory. bibliopolis, naples1984, ix 91 pp. *Journal of Symbolic Logic*, 51(4), 1075–1076.

[4] Stump, A. (2016). *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan Claypool.

[5] Wadler, P., Kokke, W., & Siek, J. G. (2020). *Programming Language Foundations in Agda*.

Appendices