

MATHEMATICAL PROOF AND  
COMPUTER SCIENCE

SNEHA POPLEY

SUPERVISING PROFESSOR: DR. LOREN SPICE

TEXAS CHRISTIAN UNIVERSITY

May 3, 2010

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	Abstract Algebra . . . . .	3
2.2	Propositional Logic . . . . .	3
2.3	Sequent Calculus . . . . .	4
2.4	First-Order Logic . . . . .	4
2.5	Prolog . . . . .	5
2.6	Higher-Order Logic . . . . .	5
2.7	Lambda Cube . . . . .	5
2.8	Intuitionistic First-Order Logic . . . . .	6
2.9	Curry Howard Isomorphism . . . . .	6
2.10	LCF . . . . .	6
2.11	Standard ML . . . . .	6
2.12	Coq . . . . .	6
2.13	Twelf . . . . .	7
<b>3</b>	<b>Comparison</b>	<b>7</b>
3.1	Sequent Calculus and Intuitionistic Logic . . . . .	7
3.2	First-Order Logic and Higher-Order Logic . . . . .	7
3.3	Lambda Cube . . . . .	8
3.3.1	LF . . . . .	8
3.3.2	CoC . . . . .	9
3.4	Prolog and ML . . . . .	9
3.4.1	Prolog . . . . .	9
3.4.2	ML . . . . .	10
3.5	Coq and Twelf . . . . .	10
3.5.1	Coq . . . . .	10
3.5.2	Twelf . . . . .	11
<b>4</b>	<b>Conclusion</b>	<b>12</b>

# 1 Introduction

This paper details the topics covered in an independent study with Loren Spice in Spring 2010. Our focus was the development of mathematical proofs in the context of computer science. The different ideas in this paper were explored for a semester through weekly meetings and presentations. My supervising professor, Loren, and I decided on the schedule for the semester after taking into account our interests, my previous research experience, as well as possible tools and ideas that could help me in the future (as a graduate student in programming language theory).

While computer science has significantly gained ground as a field over the past 100 years or so (with the introduction of punch cards, transistors, and multicore processors), mathematicians such as Euler started laying the foundation for mathematics as we know it almost 400 years ago. That said, greek philosophers and mathematicians were proving mathematical concepts even close to 500 BCE. In retrospect, computer science ideas have been around longer than people realize; the long division of whole numbers introduced in middle school can be considered to be an algorithm. Like the shortest-path problem (an algorithm to compute the shortest path between two given points in a graph), long division too has a specific set of steps that lead to the determination of the quotient.

Proof is derived from the Latin word *probare* which means "to test" [3]. In mathematics, a proof can be defined as a method to show that a given statement is true [9]. With this vague definition, a proof can take many forms, based on the context. It can be in natural language, a computer calculation, a picture, or a simulation [11]. In this case, we look at mathematical proofs that consist of steps derived consecutively by already-established rules in the logic at hand.

Theoretical computer science encompasses the combination of mathematical proofs and logic with computation. Additionally, logic in computer science involves analyzing the semantics of programming languages as well as proving theorems with the aid of computers (interactive theorem proving). This paper describes formal systems such as propositional logic, first-order logic, higher-order logic, LCF, and CoC. It also delves into some type theory as well as languages that use formal systems and/or types such as Coq, Standard ML, and Twelf. The paper then compares and contrasts the different formal systems and programming languages to give a survey of a few prevalent ideas in computer science and logic.

## 2 Overview

Since this paper covers a myriad of topics, a brief introduction of each topic is essential to be able to compare how and which ideas can be expressed in each logic/programming language. Along with these introductions, this paper provides a brief description of the specific topics covered in the independent study in each area of computer science and logic.

### 2.1 Abstract Algebra

Abstract algebra is the study of fundamental algebraic structures such as groups, rings, and fields [7]. Instead of specifically considering operations on numbers, abstract algebra characterizes elements in sets according to different properties. Abstract algebra played a role in this course for two reasons: firstly, proofs are an important component of abstract algebra. A review of the basics was essential to understand higher-level proofs in computer assistants such as Coq. Secondly, a boolean logic is an example of a logical system that is based on a boolean algebra, an algebraic structure [13]. A boolean algebra is a set  $B$  with two binary operations ( $\wedge$  and  $\vee$ ), a unary operation ( $\neg$ ), and elements 0 and 1 such that for  $a, b, c \in B$ :

- Commutativity:  $a \vee b = b \vee a$  and  $a \wedge b = b \wedge a$
- Associativity:  $a \vee (b \vee c) = (a \vee b) \vee c$  and  $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- Distributivity:  $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$  and  $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- Complements:  $a \vee \neg a = 1$  and  $a \wedge \neg a = 0$
- Absorption:  $a \vee (a \wedge b) = a$  and  $a \wedge (a \vee b) = a$

### 2.2 Propositional Logic

Propositional logic is a simple calculus that has truth values (true **t** and false **f**) and logical connectives (and  $\wedge$ , or  $\vee$ , not  $\neg$ ), but no variables [14]. It is the basis for other logics such as first-order logic (see section 2.4) as well as electric circuits. There are two ways to reason about assertions in propositional logic:

- Semantic: Calculate the meaning using standard truth tables. (i.e. writing down every possible combination of truth assignments)
- Syntactic: Use mechanical methods and laws to determine the equivalence of two formulas

In this case, a truth assignment is the substitution of some combination of **t** and **f** in a formula. A formula is satisfiable if it evaluate to true for at least one truth assignment. Two formulas,  $A$  and  $B$  are equivalent ( $\simeq$ ), if the truth assignments that satisfy  $A$  satisfy  $B$  and vice versa. A set of formulae  $S$  is a tautology if every possible truth assignment for  $S$  satisfies every formula in  $S$ . Some examples of propositional logic formulas are:

- $A \wedge B \simeq B \wedge A$
- $\neg A \simeq A \rightarrow \mathbf{f}$
- $A \rightarrow B \simeq \neg A \vee B$
- $A \leftrightarrow B \simeq (A \rightarrow B) \wedge (B \rightarrow A)$
- $A \wedge (A \vee B) \simeq (B \vee B)$

To make formulas easier to process, normal forms are used. They restrict the pattern and nature of connectives that can be used in a formula. For example, Conjunctive Normal Form (CNF) has a conjunction of maxterms, i.e.  $A_1 \wedge A_2 \dots \wedge A_m$  where  $A_i$  is a disjunction of literals.

## 2.3 Sequent Calculus

The sequent calculus is propositional logic with a more concrete natural deduction since it makes the assumptions explicit. A sequent is of the form  $\Gamma \Rightarrow \Delta$  in which  $\Gamma$  represents the context or assumptions [14]. Both  $\Gamma$  and  $\Delta$  are assumed to be finite sets of formulas. Another way to represent a sequent is  $A_1, A_2 \dots A_m \Rightarrow B_1, B_2 \dots B_n$ . A sequent is true if the conjunction of assumptions implies the disjunction of the formulas to the right ( $B_1, B_2 \dots B_n$ ). So, at least one  $B$  needs to be true. Proofs that use rules in the sequent calculus are described in section 3.1.

## 2.4 First-Order Logic

First-order logic extends propositional logic with quantifiers (forall  $\forall$  and exists  $\exists$ ) [14]. In other words, it allows variables that allow us to reason about a non-empty universe. These quantifiers range over individuals, not functions; higher-order logic (section 2.6) supports variables that can represent functions. Some examples of first-order logic are:

- Every senior is graduating:  $\forall x(\text{senior}(x) \rightarrow \text{graduating}(x))$
- Some seniors are graduating:  $\exists x(\text{senior}(x) \rightarrow \text{graduating}(x))$
- At least one senior is graduating:  $\exists x(\text{senior}(x) \rightarrow \text{graduating}(x))$

There are some basic laws for reasoning in first-order logic, and there are also the more concrete sequent laws for the quantifiers introduced in first-order logic.

## 2.5 Prolog

Prolog is a logic programming language used in artificial intelligence and theorem proving [14]. It is a declarative programming language (as compared to languages such as C which are imperative and describe the process, rather than define it). Pure Prolog is based on first-order logic and exhibits unique querying abilities due to its relational binding and SLDNF resolution. It also includes negation as failure, which means that failing to prove  $\neg B$  is equivalent to proving  $B$ . A detailed explanation of this resolution is in section 3.4. An implementation of Prolog, called  $\lambda$ Prolog, incorporates higher-order logic.

## 2.6 Higher-Order Logic

Higher-order logic allows quantification over functions and predicates [12]. It can be thought of as a kind of type theory with applications in theorem proving and natural language processing. Higher-order logic can be used to encode a meta-language for theorem provers. The higher-order abstract syntax obtained can then be used to bind the object language's constructs in the meta-language's constructs [15]. This higher-order abstract syntax actually represents programs or rules in formal systems that are involved in some form of matching or substitution. So, higher-order logic allows for the easier encoding of systems, especially for their verification or formalization.

## 2.7 Lambda Cube

The lambda cube was proposed by Henk Barendregt to represent the building up of types and terms to the Calculus of Constructions (CoC) [5]. The eight vertices of the cube each represent typed lambda calculi with edges representing an inclusion relation. The lambda cube can be thought to start with the simply-typed lambda calculus to build up to CoC. Two main ideas are seen in the different systems on the vertices of the cube: construction of function spaces and type-term dependencies. In the case of the construction of function spaces for types  $A$  and  $B$ ,  $A \rightarrow B$  is a type of function that accepts variables of type  $A$ , and returns the type  $B$ . As for dependencies, types and terms are dependent on each other:

- Types depend on terms: Church numerals
- Types depend on types: For a given type  $A$ ,  $A \rightarrow A$
- Terms depend on terms:  $MN$
- Terms depend on types:  $I = \lambda x:A.x$

The abstract syntax for the pseudo terms  $\Phi$  in the lambda cube is represented by

$$\Phi = x \mid c \mid \lambda x: \Phi. \Phi \mid \Pi x: \Phi. \Phi$$

in which  $x$  are variables and  $c$  are constants. Section 3.3 goes into more details about the differences between CoC and LF (all represented in the lambda cube).

## 2.8 Intuitionistic First-Order Logic

Intuitionistic first-order logic shows constructive independence of first-order logic operators and quantifiers [8]. This implies that  $\exists x A(x)$  is not equivalent to  $\neg \forall x \neg A(x)$  in intuitionistic logic while it is in the classical first-order logic. It is a constructive logic, so we assume a statement is true if there exists a valid construction for it. There is no construction of  $\perp$ , so  $\neg \varphi$  means  $\varphi \rightarrow \perp$  which means that assuming  $\varphi$  leads to absurdity.

## 2.9 Curry Howard Isomorphism

The Curry-Howard Isomorphism refers to the syntactic correspondence between formal logic and the lambda calculus [8]. Proofs in a deductive logic can represent programs in a computational calculus while formulas represent types in the simply-typed lambda calculus. This relationship can also be extended to other type systems such as System F or CoC.

## 2.10 LCF

LCF (Logic for Computable Functions) is an automated theorem prover that has ML as its underlying language [10]. LCF was also the inspiration for HOL, a proof assistant that supports higher-order logic. PCF is a programming language based on LCF described by Plotkin [17].

## 2.11 Standard ML

SML is a (mostly) functional statically typed programming language that supports parametric polymorphism, Hindley-Milner type-inference, and exception handling [18]. SML has an interactive compiler (SML/NJ) (similar to Prolog) and also supports pattern-matching (exhibited in case expressions). These properties are shown in more detail in section 3.4.

## 2.12 Coq

Coq is an interactive theorem prover that relies on the theory of CoC [6]. It has syntax similar to OCaml and allows the user to interactively prove theorems in mathematics and programming language theory

by allowing them to use built-in tactics. It embraces the concept of proof-carrying code in which a proof of an algorithm serves as an implementation of that algorithm for future execution.

### 2.13 Twelf

Twelf is an LF-based logic programming language that can be used to analyze and research programming language theory as well as formalize mathematics [19]. It is dependently-typed and supports high-order abstract syntax.

## 3 Comparison

### 3.1 Sequent Calculus and Intuitionistic Logic

The behavior of natural deduction varies in the sequent calculus and intuitionistic logic. The applicable laws vary, partly due to the setup of the sequent as well as the constructive independence of first-order logic operations. Below is a sample derivation in the sequent calculus.

$$\frac{\frac{\frac{\neg q, \neg p \Rightarrow \neg p}{\neg q \Rightarrow \neg p, p}}{(p \rightarrow q), \neg q \Rightarrow \neg p} \quad \frac{\frac{\neg q \Rightarrow \neg q, \neg p}{\neg q, q \Rightarrow \neg p}}{(p \rightarrow q) \Rightarrow (\neg q \rightarrow \neg p)}}{\Rightarrow (p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)}$$

The same derivation looks a little different in the intuitionistic calculus.

$$\frac{\frac{\frac{(p \rightarrow q), (q \rightarrow \perp), p \vdash p \rightarrow q \quad (p \rightarrow q), (q \rightarrow \perp), p \vdash p}{(p \rightarrow q), (q \rightarrow \perp), p \vdash q \rightarrow \perp} \quad \frac{(p \rightarrow q), (q \rightarrow \perp), p \vdash q}{(p \rightarrow q), (q \rightarrow \perp), p \vdash \perp}}{(p \rightarrow q), (q \rightarrow \perp) \vdash p \rightarrow \perp}}{(p \rightarrow q) \vdash (q \rightarrow \perp) \rightarrow (p \rightarrow \perp)}}{\vdash (p \rightarrow q) \rightarrow ((q \rightarrow \perp) \rightarrow (p \rightarrow \perp))}$$

In the sequent calculus, we are looking to prove that one of the formulae on the right-hand side is true since it is joined together with disjunctives. This makes the proof less rigorous than in the intuitionistic calculus. The intuitionistic calculus focuses on the absurdity of  $\perp$  and uses that assumption to build up to a conclusion.

### 3.2 First-Order Logic and Higher-Order Logic

First-order logic is the integration of quantifiers (forall  $\forall$  and exists  $\exists$ ) to the propositional calculus. While these quantifiers range over individuals in the first-order, they can range over functions and more in the higher-order. For example, one can express formulae such as  $\forall x \exists y(x > y)$ , in which  $x$  and  $y$  are

variables that can represent individuals. In the case of higher-order logic, one has greater power, even having the ability to represent Peano numbers with just second-order logic.

$$\forall X(X0 \ \& \ \forall y(Xy \rightarrow XSy) \rightarrow \forall y Xy)$$

Higher-order logic builds on similar constructs. It can be used to represents programs and rules, especially in the process of mechanizing metatheory.

### 3.3 Lambda Cube

Each vertex of the lambda cube has a few associated general rules as well as specific rules [5]. The specific rules are introduction rules that are parametrized by two sorts,  $s_1, s_2 \in S$

$\Pi$  rule

$$\frac{\Gamma \vdash a:s_1 \quad \Gamma, x:A \vdash B:s_2}{\Gamma \vdash (\Pi x : A.B):s_2}$$

$\lambda$ -rule

$$\frac{\Gamma \vdash A:s_1 \quad \Gamma, x:A \vdash b:B \quad \Gamma, x:A \vdash B:s_2}{\Gamma \vdash (\lambda x:A.b):(\Pi x:A.B)}$$

The eight vertices of the lambda cube represent eight systems that are defined by a subset of rule pairs where  $(s_1, s_2) \in \{(*, *), (*, \square), (\square, *), (\square, \square)\}$  in which  $*$  can be thought of as types and  $\square$  as kinds. Additionally, as shown below, in the presence of the patterns the following dependencies are found:

- $(*, *)$ : Terms depend on terms
- $(*, \square)$ : Terms depend on terms
- $(\square, *)$ : Terms depend on types
- $(\square, \square)$ : Terms depend on types

#### 3.3.1 LF

It allows for the patterns  $(*, *)$  and  $(*, \square)$  which forms a dependently-typed lambda calculus. So, while the statement below are well-typed in LF,

- $A:* \vdash (A \rightarrow *):\square$  in which  $A$  is a type, and  $A \rightarrow *$  is a kind

However, the statement below are not well-typed because of the added dependence of terms on types,

- $A:*, P:A \rightarrow * \vdash (\lambda a:A.Pa \rightarrow \perp):A \rightarrow *$

### 3.3.2 CoC

CoC (Calculus of Constructions) is the most inclusive system described in the lambda cube, showing all four patterns mentioned earlier. The two statements mentioned in the previous section are well-typed in this system along with more complex statements. These extended capability makes it a useful logic for the interactive theorem prover Coq.

## 3.4 Prolog and ML

While Prolog is a logic programming language, ML is a functional programming language that supports some imperative programming. The following subsection will detail some characteristics of the language.

### 3.4.1 Prolog

Prolog has relational binding and tries to compute all possible solutions to a formula. For instance, take the following definition of `append`,

```
append([], Ys, Ys).  
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

The output generated for the query `append(Y,X,[3,4,5,2])` is

```
X = [3,4,5,2]
```

```
Y = [] ? ;
```

```
X = [4,5,2]
```

```
Y = [3] ? ;
```

```
X = [5,2]
```

```
Y = [3,4] ? ;
```

```
X = [2]
```

```
Y = [3,4,5] ? ;
```

```
X = []
```

```
Y = [3,4,5,2] ? ;
```

```
no
```

The compiler computes all possible combinations of  $X$  and  $Y$  that satisfy the query. It does so by creating a tree of pattern-matching and branching on each possible matching.

### 3.4.2 ML

Standard ML supports parametric polymorphism in which functions can be written to support different values. So, an append function for a list can be written to support lists of integers or booleans. The definition of append is two-part, with the possibility of the list being empty and non-empty [1].

```
fun append(x:'a List, y:'a List):'a List =
  case x of
    Nil => y
  | Cons(hd:'a, tl:'a List) =>
    Cons(hd, append(tl, y))
```

Unlike Prolog, the behavior of the function does not vary if one changes the order of declaration of the cases. In some cases, we need not explicitly mention the types of parameters. The Hindley-Milner type inference algorithm allows us that freedom, it reduces it to unification based on the functionality of the statement.

## 3.5 Coq and Twelf

The comparison of the implementation of the simply typed lambda calculus (STLC) in Coq and Twelf tells us a little bit more about the nature of encodings in the language.

### 3.5.1 Coq

Below is a representation of the STLC [16]. We start by declaring a set of base types and the arrow type. The inductive definition of `ty` is shown below:

```
Inductive ty : Type :=
  | ty_base : id -> ty
  | ty_arrow : ty -> ty -> ty.
```

The terms in the lambda calculus also have a corresponding inductive definition. It has three parts: `tm_var` (a variable), `tm_app` (application), and `tm_abs` (abstraction). In the case of `tm_abs`, the definition takes in a variable of a specified type and a term, to return a term.

```
Inductive tm : Type :=
  | tm_var : id -> tm
```

```

| tm_app : tm -> tm -> tm
| tm_abs : id -> ty -> tm -> tm.

```

The typing rules associated with STLC are detailed in another inductive definition, `has_type`. It has type `context -> tm -> ty -> Prop`. This means that given a context, term, and its type, we have a system-recognized base type. It is a three-part definition based on the definition of STLC. For example, `T_Var` says that given a context of all `x` of type `T`, if `x` can be proved to have type `T`, variable `x` has type `T`. `T_Abs` is the corresponding typing rule for abstraction, and `T_App` is the corresponding typing rule for application. In the case of `T_App`, given a term `t1` of type `arrow T1 T2` and `t2` is of type `T1`, `tm_app t1 t2` is of type `T2`.

```

Inductive has_type : context -> tm -> ty -> Prop :=
| T_Var : forall Gamma x T,
  Gamma x = Some T ->
  has_type Gamma (tm_var x) T
| T_Abs : forall Gamma x T11 T12 t12,
  has_type (extend Gamma x T11) t12 T12 ->
  has_type Gamma (tm_abs x T11 t12) (ty_arrow T11 T12)
| T_App : forall T1 T2 Gamma t1 t2,
  has_type Gamma t1 (ty_arrow T1 T2) ->
  has_type Gamma t2 T1 ->
  has_type Gamma (tm_app t1 t2) T2.

```

### 3.5.2 Twelf

The types of STLC are encoded as below, with `tp` defined as the type with unit and arrow types (as described earlier) [2].

```

tp : type.
arrow : tp -> tp -> tp.
unit : tp.

```

The terms of STLC are represented by `tm` and can be an application or a lambda abstraction. There is no need to explicitly declare the variable because Twelf takes into account the presence of variables by default.

```

tm : type.
empty : tm.

```

```

app : tm -> tm -> tm.
lam : tp -> (tm -> tm) -> tm.

```

The `of` judgment is the Twelf representation of the `has-type` in Coq. Initially, we define the type of the judgment, before going on to detail its behavior for the three cases of term. The `of-empty` axiom states that a term of type `empty` is of type `unit`. `of-lam` states that if we can assume that `x` is a term of type `T2`, then `E x` is of type `T`. So, explicitly binding `x` in the term `E x` returns a term of type `arrow T2 T`.

```

of : tm -> tp -> type.
of-empty : of empty unit.
of-lam : of (lam T2 ([x] E x)) (arrow T2 T)
        <- ({x: tm} of x T2 -> of (E x) T).
of-app : of (app E1 E2) T
        <- of E1 (arrow T2 T)
        <- of E2 T2.

```

## 4 Conclusion

This paper describes an overview of formal systems and their implementations. It shows the gradual integration of mathematical logic with computer science to facilitate interactive or even automated proofs of mathematical concepts. These different tools and systems have varied capabilities which are suitable for diverse environments: Coq supports a more interactive form of theorem proving while Standard ML is a tool suitable for the development of such theorem provers. Higher-order logic adds more capabilities to systems and makes bindings that represent more programs possible, but its added complexity leaves first-order and propositional logic as useful steps in the learning process. While Barendregt's lambda cube for types in the lambda calculus gives us an idea of the relationships between different typing systems, the more applicable systems such as LF or CoC are still hard to keep track of without the aid of a computer. The development of reliable and automated verification scalable to real-life systems is not yet possible. The POPLmark challenge specifies some basic necessities of a theorem prover to lead programming language researchers in the further development of mechanized metatheory, i.e. verification of software through theorem proving [4]. Only the theorem prover Isabelle has managed to satisfy all its recommended requirements, showing that the analysis of the mathematical logic of programming languages is an important step in that direction.

## References

- [1] CS 312, data structures and functional programming, 2003.
- [2] Simply-typed lambda calculus, 2009.
- [3] Merriam webster's online dictionary, May 2010. <http://www.merriam-webster.com/dictionary/prove>.
- [4] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLS)*, Oxford, UK, Aug. 2005.
- [5] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992.
- [6] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
- [7] J. B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley, 2002.
- [8] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, April 1989.
- [9] A. Goldberg. What are mathematical proofs and why are they important, 2002.
- [10] M. Gordon. From LCF to HOL: a short history. pages 169–185, 2000.
- [11] S. Krantz. The history and concept of mathematical proof, 2007.
- [12] D. Miller. Logic, higher-order. *Encyclopedia for Artificial Intelligence*, February 1991.
- [13] J. D. Monk. The mathematics of boolean algebra. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.
- [14] L. Paulson. Logic and proof, 2002.
- [15] F. Pfenning and C. Schurmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI, 1999.
- [16] B. Pierce, C. Casinghino, and M. Greenberg. Software foundations, 2009.
- [17] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

- [18] R. Pucella. Notes on programming standard ml of new jersey (version 110.0.6), 2001.
- [19] R. J. Simmons. Twelf as a unified framework for language formalization and implementation. Technical report, Princeton University, 2005. Undergraduate Senior Thesis 18679.