Verified Computer Programs and Type Theory - A short report

Satyendra Kumar Banjare (16115102)

20-09-2018

Abstract

This is a report on advancements in type theory and formalized proof-based programming languages often referred to as verified programming languages. I have tried to explain the importance of mathematical modelling and logic system development based on the ancient works of mathematicians like Per-Martin Lof[1] by the applications in present programming toolchain and possible future aspects like applications in a quantum programming toolchain.

Things are explained in chapter-wise manner and sufficient effort has been put to properly introduce things making understanding things easy.The chapters deal with mathematical logic systems first and then proceed to explain how this has been used and developed in real systems. With the current developments that are going on in this field, I have tried to explain the very possible usage in modelling a quantum computer's theoretical behaviour and applications in Machine Learning.

This area of study comes under the umbrella term of Programming Language Research and scientists all over have been using these concepts to do explain the language semantics of any new programming language. This works using a system of inductive logics and as in abstract algebra, various operations on a given mathematical structure (*example : Rings, Groups, Sets etc.*) a computer program is thought of being an operation on the a given type system. Type theory in its most literal meaning deals with the abstract idea of Types as a fundamental mathematical structures. Classical programming languages deals with data types and now some functional languages like *Haskell, Agda, Ocaml etc* consider operations too as a type. Debugging has been made so easy because of the type systems and test driven development.

Consider the case where it is just some control signals flipping the states of bits and without a proper analytical typed contraint. The system's behaviour in this case will be completely unpredictable program-wise. Thus I would assume readers to believe with me that we would all like to have programs check that our programs are correct. Today most people who write software ,people from both academia and industry assume that the costs of formal verification of program outweighs the benefits. One simple example case is of *javascript* programming language which is very informally developed thus has a very weak type system that often leads to bugs. *Haskell* on other hand does not allow programs to disobey the type system used for that program.

Contents

Chapter 1

Introduction and History

In mathematics, logic, and thoretical computer science, a *Type System* is any of a class of formal systems which can serve as alternative standard logic system of sets. We consider every "*object*" has a "type" and all different kinds of operations are restricted to the objects of only a particular type.

Type Theory is closely related to and is often served as the basis of modern computer's type systems, which are a *programming language feature* for common bug reduction. Type theory was studied to avoid paradoxes like KleeneRosser paradox , Rossel's Paradox in a variety of formal logics and to rewrite systems. It describes the correctness of step-by-step working of an abstract model of machine. This along with complexity theory does constitute the system theory for complete functioning of any computational machine.

Organized research for the mathematics foundation started at the end of the 19th century and formed a new mathematical discipline called mathematical logic, which strongly links to theoretical computer science. It went through a lot of paradoxical results, until the discoveries stabilized during the 20th century with finally imparting mathematical knowledge with several new aspects and components known as *set theory, model theory, proof theory, etc.* , whose properties are still an active research field. *Stephan Wolfram* in his book titled New Kind of Science explores the computational aspects of machine and tries to focus on the idea that simple systems can actually reason perfectly for complex behaviour of a large systems. Classical examples of simple Cellular Automaton* and Turing-Complete machines have been studied.

Type theory in a similar sense argues that complex working can be induced from logic constraints of simpler systems. A small Type system can thus account for type-safe* behaviour of a computer system. Now for better comprehension we use the concept of (λ)-calculus to combine abstraction of type models. In mathematics, two well-known type theories that can serve as logic foundation for a system are *Alonzo Church's* **typed (λ)-calculus** and *Per Martin-Lof's* **intuitionistic type theory**.

1.1 λ-Calculus

Lambda calculus is a formal system in mathematical logic for expressing computation based on function abstraction and its application using variable binding and substitution. It is an accepted universal model of computation based on reduction and abstraction that can be used to theoretically understand behaviour of Turing machine. It was introduced in 1930s by mathematician Alonzo Church. This deals with constructing lambda terms (variable terms which change according to conditions and boundness) and performing reduction operations on them. Reduction Operations consist of $\alpha and \beta$ transformations. The former deals with renaming bound variable's name and second with replacing the bound variable with the argument expression in the body of the abstraction.It follows left associtivity i.e *fgh* is just syntactic sugar (alternate form of represetation) for *(fg)h*.

A working thumb rule for performing reductions :

$$\boxed{(\lambda param.output)input \implies output[param := input] \implies result}$$

Example :

$$(\lambda x.xy)z \longrightarrow_{\beta}(xy)[x := z] \longrightarrow_{\beta}(zy) \longrightarrow zy$$

1.2 Intuitional Type theory

Intuitionistic type theory (also known as constructive type theory, or Martin-Lof type theory) has mathematical constructs built to follow a one-to-one correspondence with logical connectives. For example, the logical connective called implication (A $\implies$ B) corresponds to the type of a function (A$\rightarrow$ B). This correspondence is called the *Curry Howard isomorphism.* This basically is an equivalent of unique mapping and existence of inverse as in the theory of sets. Previous type theories also followed this isomorphism, but Martin-Lof's was the first to extend it by introducing **Dependent types** as a basis of *High Order function dependence on basic functions.*

Machine Assisted Proving dates back as early as 1976 when the four color theorem was verified using a computer program. Butterfly Effect's discovery also was possible due to computer simulation of program with given some finite initial states. Most computer-aided proofs to date have been implementations of large *Proofs-By-Case* of a mathematical theorem. It is also called as *Proof-By-Induction* where child cases are considered first in an attempt to fully prove a theorm. Various attempts have been made in the area of *Artificial Intelligence* research to create smaller, explicit proofs of mathematical theorems using machine reasoning techniques such as *heuristic search.*

Such **Automated Theorem Provers** have found new proofs for known theorems also given a number of new results. Additionally, interactive proof assistants allow mathematicians to develop acceptable human-readable proofs which are then again verified too in a similar procedure.

1.3 Proof Assistants Introduction

Machine theorem basically involves model checking, which, in the simplest case involves brute-force enumeration of many possible states (although the actual implementation of model checkers requires much robustness for checking every case, and it does not simply reduce to brute force). There are *hybrid* theorem proving systems which use model checking as an inference rule. There are also programs which were written to prove a particular theorem, with an assumption that if the program finishes with a certain result, then our proposed theorem is true.

In computer science and mathematical logic, a **Proof Assistant or Interactive Theorem Prover** is a software tool to help with the development of formal proofs. This involves interactive editor, or other interface, with which a human can guide the search for proofs. Some steps are reduced by the computer and base cases are enlisted to be proved. We will discuss this part later.

In the present Programming Language research, the correctness of *Computer Programs* is proved using similar notions considering some ”pre” and ”post” cases. This idea can thus be extended to proving correctness of a programing language semantics.

Currently the developments in Quantum Computing strongly uses these proof-based programming language mainly because of the completely random behaviour of a *Quantum states* of Qubits. All that we have are probablities of existence of a given quantum state. Let’s say we can we can organize the chaos to some extent by *categorizing* those probabilities in few cases, then later operations and control logics will have to have an inductive transfer of the qubit’s states. This is analogus to how we deal with things in Type theory. Therefore developing a strong typed abstract model will benefit quantum computing to a much bigger extent.

We will understand the working of proof assistants by firstly revisiting topics in abstract algebra, computer architecture and compiler technology. We will then proceed to introduce the type systems and core elements of type theory in study and finally combine both to reason for correctness of a programming language and a programming stack.

I have later introduced concepts of quantum computing followed by the application of type theory in it to put some light on some ongoing active research in that field.

Chapter 2

Abstract Algebra

Abstract algebra also referred to as modern algebra is the study of algebraic structures. An algebric structure serves as an explinatory basis of functional operations on an underlying set. A set here also is an abstract idea of a collection of things that share certain common features and should not only be confined with sets dealt in classical set theory. Over the time on the basis of types of operations and *logical freedom* to do so on various sets have led to acceptance of various algebric structures defined below. The study of abstract algebra is used primarily in areas of topology of **n** dimensions. For anyone this is insane for physical boundations arise when we try to visualize things out. Thus we need to classify abstractions in one of the algebric structures and then deal with it.

2.1 Group Theory

This is a part of abstract algebra where we deal with **Group** mathematical structures. In mathematics, a **Group** is an abstract algebraic structure that consists of a set of elements and various operations which when performed on any *two* elements of the set results in a *third* element from same set. It satisfies four conditions called the "group axioms" or "group properties", namely closure or closed operations i.e. that maps from set to itself , associativity, identity and invertibility. One of the most common examples of a group structure is the set of integers with the addition operation. This algebric structure is fundamental basis of other more complex algebric structures. It is studied in followings ways.

A group G with the property with a given operation "o" such that

$$a \ o \ b = b \ o \ a \ |\forall a, b \in \{G\}$$

is called abelian or commutative. Groups not satisfying this property are said to be nonabelian or non-commutative.

2.1.1 Cyclic Groups

A **Cyclic Group** is a group that can be generated by a single element often called as group generator). Cyclic groups are always Abelian. A cyclic group of finite group order n is denoted G_n . The genralized generation rule can be specified as :

$$X^n = I(X \in G)$$

In the simple sense, it means identity element can be generated by any single element by repeated application of group operations. And since the identity element can be realized this way, all the elements of group can be realized too.

2.1.2 Permutation groups

A **Permutation Group** is a group G whose elements are permutations of a given set M and whose group operation is the composition of permutations in G (which are thought of as bijective functions from the set M to itself). The group of all permutations of a set M is the symmetric group of M, often written as Sym(M).[1] The term permutation group thus means a subgroup of the symmetric group. If M = 1,2,...,n then, Sym(M), the symmetric group on n letters is usually denoted by Sn.

2.1.3 Group Actions

an action of a group is a formal way of interpreting the manner in which the elements of the group correspond to transformations of some space in a way that preserves the structure of that space.

. For other groups, an interpretation of the group in terms of an action may have to be specified, either because the group does not act canonically on any space or because the canonical action is not the action of interest. For example, we can specify an action of the two-element cyclic group

$$\mathrm{C}_2 = \{0, 1\}$$

on the finite set

$$\{a, b, c\}$$

by specifying that 0 (the identity element) sends

$$a \mapsto a, b \mapsto b, c \mapsto c$$

, and that 1 sends

$$a \mapsto b, b \mapsto a, c \mapsto c$$

. This action is not canonical.

2.2 Field Theory

In mathematics, a field is a set on which addition, subtraction, multiplication, and division are defined, and behave as the corresponding operations on rational and real numbers do. A field is thus a fundamental algebraic structure, which is widely used in algebra, number theory and many other areas of mathematics. The best known fields are the field of rational numbers, the field of real numbers and the field of complex numbers. Many other fields, such as fields of rational functions, algebraic function fields, algebraic number fields, and p-adic fields are commonly used and studied in mathematics, particularly in number theory and algebraic geometry. Most cryptographic protocols rely on finite fields, i.e., fields with finitely many elements.
The relation of two fields is expressed by the notion of a field extension. Galois theory, initiated by variste Galois in the 1830s, is devoted to understanding the symmetries of field extensions. Among other results, this theory shows that angle trisection and squaring the circle can not be done with a compass and straightedge. Moreover, it shows that quintic equations are algebraically unsolvable.
Fields serve as foundational notions in several mathematical domains. This includes different branches

of analysis, which are based on fields with additional structure. Basic theorems in analysis hinge on the structural properties of the field of real numbers. Most importantly for algebraic purposes, any field may be used as the scalars for a vector space, which is the standard general context for linear algebra. Number fields, the siblings of the field of rational numbers, are studied in depth in number theory. Function fields can help describe properties of geometric objects.

2.3 Rings Theory

In mathematics, a ring is one of the fundamental algebraic structures used in abstract algebra. It consists of a set equipped with two binary operations that generalize the arithmetic operations of addition and multiplication. Through this generalization, theorems from arithmetic are extended to non-numerical objects such as polynomials, series, matrices and functions.

Whether a ring is commutative or not (i.e., whether the order in which two elements are multiplied changes the result or not) has profound implications on its behavior as an abstract object. As a result, commutative ring theory, commonly known as commutative algebra, is a key topic in ring theory. Its development has been greatly influenced by problems and ideas occurring naturally in algebraic number theory and algebraic geometry. Examples of commutative rings include the set of integers equipped with the addition and multiplication operations, the set of polynomials equipped with their addition and multiplication, the coordinate ring of an affine algebraic variety, and the ring of integers of a number field. Examples of noncommutative rings include the ring of n n real square matrices with n 2, group rings in representation theory, operator algebras in functional analysis, rings of differential operators in the theory of differential operators, and the cohomology ringof a topological space in topology.

2.3.1 Polynomial Rings

In mathematics, especially in the field of abstract algebra, a polynomial ring or polynomial algebra is a ring (which is also a commutative algebra) formed from the set of polynomials in one or more indeterminates (traditionally also called variables) with coefficients in another ring, often a field. Polynomial rings have influenced much of mathematics, from the Hilbert basis theorem, to the construction of splitting fields, and to the understanding of a linear operator. Many important conjectures involving polynomial rings, such as Serre's problem, have influenced the study of other rings, and have influenced even the definition of other rings, such as group rings and rings of formal power series.

A closely related notion is that of the ring of polynomial functions on a vector space.

2.4 Iso-morphisms

In abstract algebra, a group isomorphism is a function between two groups that sets up a one-to-one correspondence between the elements of the groups in a way that respects the given group operations. If there exists an isomorphism between two groups, then the groups are called isomorphic. From the standpoint of group theory, isomorphic groups have the same properties and need not be distinguished.

2.5 Homo-morphisms

In algebra, a homomorphism is a structure-preserving map between two algebraic structures of the same type (such as two groups, two rings, or two vector spaces). The word homomorphism comes from the ancient Greek language: (homos) meaning "same" and (morphe) meaning "form" or "shape". However, the word was apparently introduced to mathematics due to a (mis)translation of German hnlich meaning "similar" to meaning "same".[1]

Homomorphisms of vector spaces are also called linear maps, and their study is the object of linear algebra.

The concept of homomorphism has been generalized, under the name of morphism, to many other structures that either do not have an underlying set, or are not algebraic. This generalization is the starting point of category theory.

A homomorphism may also be an isomorphism, an endomorphism, an automorphism, etc. (see below). Each of those can be defined in a way that may be generalized to any class of morphisms.

2.6 Matrix Groups

In mathematics, a matrix group is a group G consisting of invertible matrices over a specified field K, with the operation of matrix multiplication, and a linear group is an abstract group that is isomorphic to a matrix group over a field K, in other words, admitting a faithful, finite-dimensional representation over K.

Any finite group is linear, because it can be realized by permutation matrices using Cayley's theorem. Among infinite groups, linear groups form an interesting and tractable class. Examples of groups that are not linear include groups which are "too big" (for example, the group of permutations of an infinite set), or which exhibit some pathological behaviour (for example finitely generated infinite torsion groups).

2.7 Category Theory

Algebraic structures, with their associated homomorphisms, form mathematical categories. Category theory is a formalism that allows a unified way for expressing properties and constructions that are similar for various structures.

The language of category theory is used to express and study relationships between different classes of algebraic and non-algebraic objects. This is because it is sometimes possible to find strong connections between some classes of objects, sometimes of different kinds. For example, Galois theory establishes a connection between certain fields and groups: two algebraic structures of different kinds.

Chapter 3

Computer Architecture

3.1 Instruction Set

3.2 Microarchitecture Design

3.3 Logic Synthesis

3.4 Implementation

Chapter 4

Compilers

4.1 Preprocessing

4.2 Lexical Analysis

4.3 Parsing

4.4 Symantic Analysis

4.5 Optimization

4.6 Code Generation

4.7 Interpreter

Chapter 5

Type Systems

5.1 Dependent Types

5.2 Flow-Sensitive

5.2.1 Gradual

5.3 Latent Type-ing

5.4 Refinement

5.5 Substructural

5.6 Unique

5.7 Type Safety & Memory Safety

5.8 Static vs Dynamic

5.9 Nominal vs Structural

5.10 Duck Type-ing

5.11 Weakly vs Strongly Typed

5.12 Unified Type System

5.13 Applicative equivalence

Chapter 6

Proving

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 7

Proof Assistants

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 8

Programming Languages

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 9

Programming Language Verification

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 10

Complete Stack Verification

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 11

Quantum Computing Overview

11.1 Qubits

11.1.1 Representation

11.1.2 States and Entanglement

11.2 Quantum Gates

Chapter 12

Quantum Computing Programming Languages

12.1 Quantum Computing SDKs

12.2 Imperative QC Languages

12.3 Functional QC Languages

12.4 Ongoing Development

Chapter 13

Interplay between Quantum Computing and Type Theory

13.1 Procedural Implementations

13.2 Descriptive Implementations

13.3 Phantom Types

13.4 Conclusion

Chapter 14

Applications in Machine Learning

14.1 Automated Theorem Provers

14.2 Teaching AI to Prove Theorems

14.3 Automated Reasoning for formalization

Bibliography

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.