Verified Computer Programs and Type Theory - A short report

Satyendra Kumar Banjare (16115102)

October 8, 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 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.

Categories

[**Operating Systems**]: ReliabilityVerification, [**Software Engineering**]: Software/Program Verification

Key Words

Verification, Design, COQ, PLT (Programming Language Theory)

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

$$C_2 = \{0, 1\}$$

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.

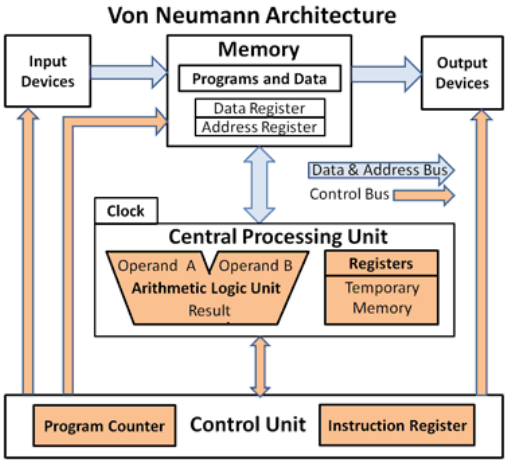
Chapter 3

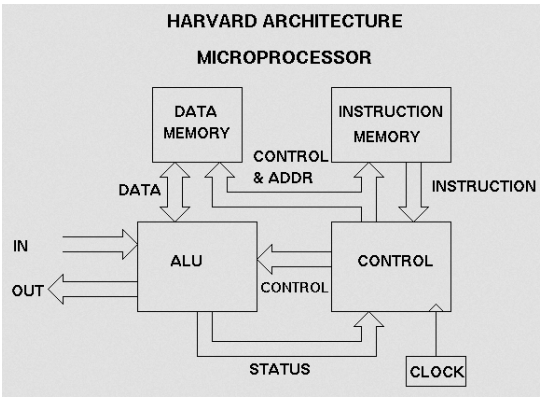
Computer Architecture

Computer architecture in layman terms is all about building a computer machine with some or total specification and detailing of how a set of software and hardware technologies interact with each other as to form a computer system.It refers to how a computer system is designed and what technologies it is compatible with.

One of the most followed Logical model of computer architecture is **Von Neumann's Architecture**. This was proposed by the mathematician *John von Neumann* in 1945. It describes the design of an electronic computer with its CPU, which includes the arithmetic logic unit, control unit, registers, memory for data and instructions, an input/output interface and external storage functions. It follows that an instruction fetch and data operation cannot be performed at the same time. Another used architecture is **Harvard's Architecture** which is also similar to Neuman's Architecture except that it uses different physical paths or data buses for carrying data and instruction. This idea has lead to creation of parallelism and this distinction has evolved over to reason for the different Cache mechanisms used for a *Random Accessed Memory (RAM)* and *Read Write Memory (RWM)* type memory storage.

Let's now look at the basic parts of computer architecture.





3.1 Instruction Set Architecture

Instruction Set Architecture (ISA) serves as the interface between computer software and hardware. It is a set of rules or instructions that define the working of a computer. These have evolved overtime due to numerous optimizations implemented for the same hardware. Such improvements have led to more abstractness. For example, most of present ISA deal with pointers rather than directly referring to registers in the machine byte code implementation. ISA define some of the very fundamental data types that correspond to some fixed bit size storage in memory. It handles very low-level control flow and Logic operations. Its very important to understand many compiler-implementations can produce same Instruction Set. Instruction Set is defined only for a hardware and assembly-level machine codes should follow these IS specifications only.

An ISA may be classified by their implementation complexity. A **Complex Instruction Set Computer (CISC)** has many special instructions but can be realized with other basic instructions too thus a **Reduced Instruction Set Computer (RISC)** is used that increases efficiency by implementing only those instructions that are frequently used in programs, and also some special instructions if required and made possible by an appropriate memory-time tradeoff.

Other types include **Very Long Instruction Word** (VLIW) architectures, and the closely related **Long Instruction Word (LIW)** and **Explicitly Parallel Instruction Computing** (EPIC) architectures. These architectures seek to exploit instruction-level parallelism with less hardware than RISC and CISC by making the compiler responsible for instruction issue and scheduling.

3.2 Microarchitecture Design

Microarchitecture design also called computer organization desing deals with the ways a given instruction set architecture (ISA), is implemented in a particular processor. The microarchitecture is usually represented as diagrams that describe the interconnections of the various microarchitectural electronic hardware elements of the machine which includes components from basic registers to complete ALU. These diagrams generally separate the datapath and instruction control path. Some important terms related to the microarchitecture design are given here under.

The **Instruction Cycle** is the basic operational process of a computer system. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction describes, and then carries out those actions. **Instruction Pipelining** is a technique for implementing instruction-level parallelism within a single processor. **Cache** is simply very fast memory. It can be accessed in a few cycles as opposed to many-step access or other type of memory. **Multiprocessing** is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor or the ability to allocate tasks between them. **Multithreading** or **Concurrency** is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to execute multiple processes or threads concurrently, supported by the operating system.

3.3 Logic Synthesis and Implementation

Finally when both the ISA and Microarchitecture design are available, the next thing to do is finally implement it.This includes all the logic implementation (the gate logic), Circuit Implementation and desingn validation through FPGAs. The way it works is making complete circuitary on a FPGA board and then performing error reductions to improve efficiency. Else one may simulate the proposed μ-architecture on an existing machine or computer system.

Here comes the intersting part that follows from the **Church-Turing Hypothesis** that says any first order recursive and computable function can be well simulated in a **Turing-Complete machine**. All classical machines are however not perfectly turing complete and some functions can't be simulated on them. Here on **Destusch** added his hypothesis claming the *Quantum Parallelism* couldn't be sufficiently simulated because of much large randomness in the resluting values.

Chapter 4

Compilers

Compilers are that piece of software that *translates* the programs in one language to other languages. They are different from **Assembler**. An assembler converts the assebly codes to final machines' instruction codes according to the ISA of the target computer hardware. Compilers are generally thought to convert higher level programs to low level programs. They are of different types like Cross compilers, Bootstrap Compilers (it comes under cross compilers) , Transilers and De-compilers. The basic operations carried by most of the compilers are :

4.1 Lexical Analysis

This is the first part of any compiler that takes in the program spurce code and does word-substitutions, Macro expansions, generating *Tokens* and cleaning up extra white spaces. All this work is done by using regular expression to increase compactness of input code. The tokens when combined in a special way form what is known as **Abstract Syantax Trees (AST)**. The regular expressions are interpreted as *Non-Deterministic Finite Automata (NDFA)* since the states or the inner expression in between any two parts of a Regex can't be fixed since it will vary with the input used. Thus the NDFA is converted to *Deterministic Finite Automata (DFA)* which will be parsed to AST using the technique of ϵ-*Transformation*.

4.2 Syntax Analysis or Parsing

This is the process of recombining the context-free grammatical tokens into a data structure, syntax tree. It is important that any sort of syntax error should be reported at this point of compilation. Context-free grammar is a recursive notation for describing sets of strings and imposing a structure on each such string without consiering the context in which the syntax is generated.

4.2.1 Abstract Syntax Trees and Abstract Binding Trees

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. It is "Abstract" in the sense that it does not represent every detail appearing in the real syntax of the programming language used, but rather just the structural and content-related details.

An abstract syntax tree is an ordered tree whose leaves are variables and whose interior nodes are operators whose arguments are its children. Asts are classified into a variety of sorts corresponding to different forms of syntax that divide ASTs into syntactic categories. For example, common programming languages often have a syntactic distinction between commands and an expression. These are two type of sorts for ASTs.

Abstract Binding Trees (ABT) enrich the meaning of AST with the methods to introduce new variables and symbols called *bindings* to an existing AST. It should be noted that it is a theoretical concept and only AST is used in compilers while parsing. The additions are performed within a given range of significance of the added terms called its *scope*. In desinging type systems, we therefore deal with these ABT to construct new language.

Writing a new Programming language grammar basically involves writing syntactic sugar for *Expressions, Statements* (often called commands) and *Declaration* like header declarations and variable assisgnments. There are some **rules** defined and *Judgements and derivations* are performed on them. The derivations are mostly inductive and also covers the finite aspect of machine. This can be understood in a way that DFA is mantained and machine never goes to a NDFA state.

For example here is a derivation of successor of a natural number, [succ(succ(succ(zero)))nat] :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{zero \quad nat}}{succ(zero) \quad nat}}{succ(succ(zero)) \quad nat}}{succ(succ(succ(zero))) \quad nat}$$

.

The denominator in the expression is an inductive derivation from the numberator conditions.

4.3 Type Checking

This is one of the most important part where data consistency is checked of input source code. The Compiler may exploit or depend on type information, which makes it natural to combine calculation of types with the actual translation of source code. A language is considered to be type-safe if does not allow violation of basic data types. The varying level of strictness leads to formation of weak and strong type safe languages.

It shows a diagram of the design space of static vs. dynamic and weak vs. strong typing, placing some well-known programming languages in this design space.

4.4 Optimization

This part deals with converting the present code as AST into more efficient using a series of transformations that makes the code use lesser sources but produces same output and variable scope is maintained throughout. Some of the common optimization techniques are :

Peephole optimizations replaces a complex multi-step instruction by an instruction of less-number of steps preferably a single step, for example multiplication by 2 is equivalent ot rotating bits to left by 1. **Data Flow Optimizations** convert expressions into simple to reduce number of duplicate expressions.

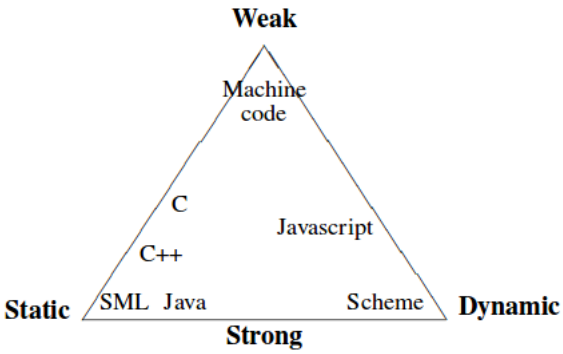


Figure 4.1: Design Space of Types

Static Single Assignment (SSA) optimization involves assigning variables only a single value. If variables have same values then they are considered as one variable only. Scope Tables are used in these cases.

4.5 Code Generation

This part converts the optimized AST into machine codes of a hardware compatible ISA representation. It involves expoliting complex instructions and carefully placing jump and branch instructions for the machine code program. This part is also to be optimized and texhiques of *Register allocation, Instruction selection, Instruction scheduling* and *Rematerialization*.

4.6 Interpreter

Interpreters are different from the compilers only in the aspect it performs operations as the code is produced without having a previously compiled code and then executing it. It works similar to a compiler in terms of basic steps but uses an **Interediate Representation (IR)** of AST before producing machine code and executing it too.

Low Level Virtual Machine (LLVM) is a popular compiler technology that takes into account the benefits of IR in compilation. It is used all over now and has increased portability of softwares a lot. Recently the *Polyhedral Loop Optimizations (Polly)* in the LLVM IR has gained popularity and they have been used in all possible places.

Chapter 5

Type Systems

A **Type System** for any programming language is a set of rules that associate a property called " *Type*" with the basic grammatical constructs of the language like variables, constants, statements and commands. It is the computer programming equivalent of type theory which uses type safety and type validation to prevent bugs, compiler runtime errors and compiler optimizations. It is very basic for any programming language to have types to actually perform any of Higher Order Logic on the host machine. Note that a type system can work only if the program is run. In other words it is absolutely insignificant to check a program if it is not run or atleast compiled to be run.

5.1 Dependent Types

This is the most important kind of Types whose definition depends on a value. It is the conceptual overlap between computer's Type System and Type theory as it allows evaluation by induction. In functional Programming languages, dependent types increases expressivity of a defined type. It's examples include dependent functions and dependent pairs. Introduction of new variable or new values may completely change the overall behaviour. Thus the Structual Type checking would be used and not nominal (defined below). It is considered to be of two types from mathematical point of view :

- **5.1.1 Π-Type**

  It is called **Dependent Product Type** where given a type space Ω, we define a family of types B:A→ Ω that says every term a:A has a type B(a):Ω. It explains that there is not a fixed co-domain for functions of this type. The name 'pi-type' comes from the idea that these may be viewed as a Cartesian product of types. Pi-types can also be understood as models of universal quantifiers. For an example let's say a function gives tuple of order n for a given natural number input n. Clearly the terms in that tuple does not depend on the value of n. Here also, the Dependent product nature is preserved.

- **5.1.2 Σ-Type**

  It is called **Dependent Sum Type** and is categorically dual to dependent product type where given a type space Ω, if we have a family of types B:A→ Ω then there must exist a dependent pair type.

It explains that there should be fixed codomain for any function.

They use mathematical operators ($\rightarrow$) and binders ($\forall$) to produce complex types. The higher order **Dependently Typed Polymorphic lambda calculus** also called **Calculus of Inductive Constructions** is the foundational basis of all Proof Assistants.

5.2 Flow-Sensitive Types

Such Type systems determine the type of a variable by controlflow. The type of a variable may change in any of the program's method. The type inference (defined below) is used for checking these type systems.

5.3 Latent Types

Latent typing refers to a type system where types are associated with values and not variables.

5.4 Refinement Types

These are a set of some preconditions that define the behaviour of the function or variable. These can be understood as the return types of most of functions which may dpend on some 'if/else' conditions.

5.5 Substructural Types

This is the type system defined depending on the use of a variable. For exmaple, a variable may be used once or many times. It has **Linear**, **Affine**, **Relevant** and **Ordered** as subcategories. The Linear system allows the use of object exactly once, Affine system allows use atmost once, Relevant system allows usage any number of times and Order system allows using objects only once in an order.

5.6 Unique Types

A Unique type guarantees that an object is used in a single-threaded way only with maximum one reference to it. It mantains the referntial transparency and improves the efficieny of functional languages.

5.7 Type Safety & Memory Safety

Type safety refers to type equivalence and behavioural equivalence of all language constructs for a given program written in any programming language. Memory safety on the other hand refers to leakage and corruption of memory by any program. example : stack overflow.

Some Type checking techniques to ensure type safety are :

- ### Static vs Dynamic

  A **Static** type system refers to compilation time type checking. The source code's type correctness is verified first and then the executable is generated. A **Dynamic** type system checks type correctness

during the runtime. This is the case when bindings and linking are created so that data passed on from one of the program to other is correct type-wise. Together both create polymorphic type checking where a single interface or a single Type check-pass is capable of doing both kind of type checks and is very portable to use.

- **Nominal vs Structural**

  In a **Nominal** or **Nominative** type system, the equivalence of data types is determined by explicit declarations or name of types. For example, in C programming language, two "*struct* types with different names in the same translation unit are never considered compatible, even if they have identical field declarations. In **Structural** type system, the inner structure of a data type are evaluated to show data equivalence and type safety.

- **Manifest vs Inferred**

  Manifest Typing is explicit identification by the software programmer of the type of each variable being declared. For example: if variable X is going to store integers then its type must be declared as integer. Type **Inference** refers to the automatic detection of the data type of an expression in a programming language. For example, user may write a float point decimal without having declared that variable as a float point. The compiler technology should be able to detect what program is trying to do and perform further actions accordingly. This is also referred to as **Typeless Typeing** where the type is later introduced depending on data passed.

- **Duck Type-ing**

  This type system got it's name from on of the most common computer engineering practices which says if something behaves like a duck, quacks like aduck then it must be duck. In other words the behaviour determines the data type. For example, adding a float type integer to a double type integer, the answer should have a double type because it is surely to have characteristics of a double type integer.

5.8 Unified Type System

In the object oriented programming (OOP) context, the abstraction and scope of methods deals with it's derivation from a base class. similar logic can be extended to class objects whose properties and methods depends on a base class. A **Unified Type System** states that all the Types of objects are derived from a single root type. Thus that object is capable of performing operations approved by the root type. Example of such a programming language is C# developed by Microsoft. In C# the concepts of encapsulation and methods have been decoupled from the reference requirement so that a type can support methods and encapsulation without being a **Reference type**. A reference type is also an OOP concept that says all instances of class objects are by reference.

Chapter 6

Proving

In this and next chapter I will discuss how proving is done on a computer machine. Similarity of mathematical type structures and a programming language type system is to be understood thus explaining the theoretical correctness of afore mentioned equivalence.

6.1 Hilbert-style deduction systems

This is a kind of logic system. The idea of a formal deduction says that there are mathematical formulaes or functions (for Input-Output generalization) which are either the base cases often called Axioms or are derived from some axioms by applying some hypotheses. Let Γ be a set of pressumed contions and base cases (Axioms) and ϕ be deduction made using Γ then we can write this as $\Gamma \vdash \phi$ meaning ϕ can be duduced and hence can be proved from Γ.

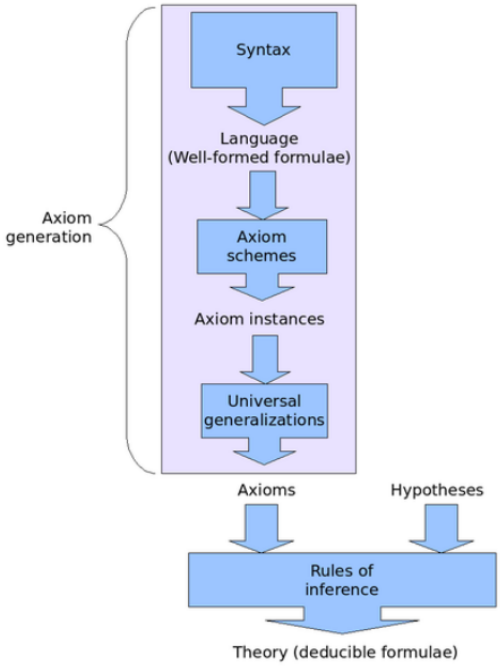


Figure 6.1: A Basic Deduction System

The Deduction system is well characterized by use of numerous Logical Axioms called an **Axiom Scheme**. It gives us a iterative and inductive nature of axiom deduction so that it can be extended as many step as one wish. The terms called *Combinators* (like $\rightarrow$) that interweave the results of a set of a given Axiom Scheme and *Quantifiers* (like $\forall$) generalize the results of axiom schemes using some quanitfiable data like the initial value of a mathematical function.

Using Combinators and Quantifiers one may easily reduce the deductions into less number of steps hence forming metatheorems. A metatheorem inmathematics is the one that can be achieved using some other very basic mathematical theorems. Thus the this gives us idea of correctness of untyped λ -calculus. We can thus extend this to other objects also. Below I have tried to explain formation of a Hilbert-style deduction system for Abstract Binding Tree.

Professor **Robert Harper** in his book *Practical Foundations of Programming Languages* defines the term *Judgement* 'J' defined on a set of rules 'R' defined for a type that satisfies a property 'P' (hence forming an Abstract Binding Tree ABT) and it follows an inductive definition

$$\frac{J_1 J_2 J_3 \cdots J_k}{J}$$

where each J_i is some sort of pre-defined abstract judgements or base cases. The set R is a collection of all such J_is. R is also referred to as an *Axiom* if k=0 i.e it is just first judgement to be made. for example :

$$\frac{}{zero \quad nat} \qquad and \qquad \frac{a \quad nat}{succ(a) \quad nat}$$

He explains *Derivabilty* Judjement $J_1 ... J_k \vdash_R K$ for a given set of rules 'R', with J_i and K as basic judgements to show that we may derive K from the expansion of $R \cup \{J_1 ... J_k\}$ of the rules with axioms

$$\frac{}{J_1} \quad \frac{}{J_2} \quad \frac{}{J_3} \quad \cdots \quad \frac{}{J_k}$$

$\Gamma \vdash_R K$ means that a Judgement K is derivable from rules $R \cup \Gamma$.

He defines *Admissibility* Judgement written as $\Gamma \models_R J$, that if there exists any judgement like $\vdash_R \Gamma$ then it implies $\vdash_R J$. It means that if all the assumptions (base cases) are derivable from R then J is derivable from R.

The mathemetical formalization of static part and dynamic part of a typed programming language is expressed in the form of the judgements.

The static feature of a language imposes constraints on the formation of the phases that are sensitive to the context in which thay occur. It follows an induction definiton like this

$$\chi | \Gamma \vdash e : \tau$$

where χ is a finite set of variables and Γ is a typing context consisting of hypothesis of the form $x{:}\tau$, one for each $x \in \chi$. The Dynamics feature basically means the transition of states expressed in form of *Transition* Judgement having a transition $s \longmapsto^n s'$ (for n^{th} order transition) as

$$\frac{}{s \longmapsto^n s'} \qquad and \qquad \frac{s \longmapsto s' \quad s' \longmapsto^* s"}{s \longmapsto^* s"}$$

It is important to note that all this operations are performed on pressumed abstract mathematical objects, Abstract Binding Trees (ABTs) which are also the most important fundamental part of any type system. This theoretical equivalence thus allows us to construct the static and dynamic rules and some derived Judgements or abstract functions (with type correctness) for a programming language.

6.2 Curry-Howard Correspondence

In programming language theory the **CurryHoward correspondence** (also known as the CurryHoward isomorphism) says that mathematical proofs and computer programs are equivalent. This equivalence depends on *Axiom Schemes* with a Type as a combinator. It also follows the *Combinatory Logic System* that allows removing a mentioned variable using some 'ways'. These ways also called functions and they use some combinators to define a result from given arguments. Then it assumes the correctness of Typed-λ Calculus.

Let's assume a function F that has a Type defined by set of rules R and some set of initial conditions J_i (i=$1 \cdots n$) then we can write :

$$F \quad \equiv \quad J_{1 \cdots n} \vdash_R \lambda \quad (\lambda \text{ is the result of function that also has a Type defined by R.})$$

A computer program is a set of such functions and a computer aided proof is a computer program. Thus it can be derived that "*A computer Proof is a program with the thesis being proved is type for that program*" . Thus this forms the basis of proof assistants and Functional Programming.

6.3 Hoare Logic

6.4 Separation Logic

Chapter 7

Proof Assistants

Having discussed some basic ideas of working of a Proof Assistant and theoretical soundness of proving things, let's now get to the programming and implementation. Proof Assistants as stated earlier is a software tool do help in developing formal proofs. As soon as the formal logic system was developed, a very basic theorem prover called *Logic for Computable Functions* (LCF) was developed along with a basic theoretical programming language *Programming Computable Functions* (PCF) was developed. These were some of the initial works in the field of Proof Assistants. The successors include *Isabelle* and *COQ*. Also the theory advanced to reason for working at higher order of complexity called *Higher Order Logics* (HOL).

7.1 LCF - Logic for Computable Functions

This was the first interactive Automated theorem prover that also introduced a new *Domanin Specific Language* (DSL) called ML. It allowed users to generate new tactics (disscussed below) to improve proving. The type system of ML ensures that theorems are derived completely by the given inference rules.

LCF shaped the thought process of how to write a type of a function to finally verify it by using terms like *Inclusive* rules, *Conditional* rules, λ rules, *Truth* rules. It also introduced the idea of tactic logics and put forward a *Read Evaluate Print Loop* (REPL) type approach of proof assistant. It worked by breaking a given hypotheses into smaller forms. The smaller forms often are Lemmas on which the proof depends.

The LCF-style inference kernel used in many proof assistants consists of two layers: proof objects and theorems. Proof objects are represented as concrete datatypes. Depending on system parameters, a varying amount of information is maintained here. In any case, there is a complete record of oracles used in the proof. An explicit λ-term representation of the proof is also possible, but requires significantly more resources. Theorems are elements of an abstract datatype thm.

According to the original LCF tradition, theorems are proven propositions that have been certified by the kernel module, which implements primitive inferences as abstract datatype constructors. Any operation on theorems has to go through that kernel, so any value of type thm is *correct by construction*.

7.2 Tactics

The forward reasoning of proving a hypothesis consisting of first proving sub-hypotheses say A and B and finally concluding that if A is proved and B is proved we have proved $A\hat{B}$ (our main hypothesis). The

backward reasoing says to prove $A\hat{B}$, we need to prove A and B. **Tactics** basically run inference rules forward or backwards to help break down proving things. It may also apply a predefined lemma ($\simeq$ function application), split up a lemma about some inductive type (in that particular step of proving) into a case for each constructor, and so on.

Basic tactics may succeed or fail depending upon the context in which they are applied. More advanced tactics are like little functional programs that run the basic tactics, perform pattern matching over the terms in the goal and/or assumptions, make choices based on the success or failure of tactics, and so forth. More advanced tactics deal with arithmetic and other specific kinds of theories. The key paper on Coq's tactic language is the following:

7.3 De Bruijn criterion

One may ask how to veify the verifying system being used. **De Bruijn criterion** answers this by ensuring that proof assistants create an independently checkable proof object while the user is interactively proving a theorem. These proof objects should be checkable by a program that a skeptic user could easily write him/herself. De Bruijns Automath systems were the first to specifically focus on this aspect and therefore this property wascoined De Bruijn criterion by Barendregt (Barendregt & Geuvers 2001). In De Bruijns systems, the proof objects are basically encodings of natural deduction derivations that can be checked by a type checking algorithm.

Chapter 8

Programming Language Verification

Any computer programming language consists of type system to its core supported by data structures and control flow techniques. Any Programming language finally can be broken down to machine codes resulting in perfoming specified actions. The assembly level machine codes should theoretically represent the same semantics as implied by the higher order programming language. **Programming Language Verification** in the broader sense is checking that programs in one programming language is equivalent to the one in other. Generally this equaivalence is tested between a high order language and a small low level C language.

The basic understanding of how the verification work can be thought as, say there are two programs e_1 and e_2 then to prove they are equivalent the must meet the following criteria :

- Input Parameters as well as the context in which the program is being run is same. Input parameters can be thougt as some initial values to a program. Input parameters should necessarily be of same type. The context refers to same harware used as well as the scope of the program in consideration is maintained same. Context can then again be broken as a *Global* context and a *Local* context.

- The result should be same and should be of the same type. It is necessary that any program should not violate any of the bonding rules imposed by context.

How do we check this ?

Well the method used is called **Step-Indexing**. In this method we look for holes in a program. These holes represented by C_i can be thought as a program's loop structure or the way conditional arguments have been put up. In any step of a program thus if we are able to show that at that point, $C_2[e_1]$ termminates and $C_1[e_2]$ termminates also producing same results then we have shown the programs are equal. The cardinality issues related to number of such possible holes in programs increases the complexity of verification proof. This method therefore says that we should unfold the program upto required N steps and consider the equivalence then.

The areas where this method stucks is where we are unable to predict N. This is the case of IO type functions. The work on this is going on and improved proofs are being generated.

8.1 Verified Software Toolchain

This project led by Princeton University deals with verifying software programs and compilers. It is an advancement to Compcert. It uses multi layered kernels to securely verify the programs.

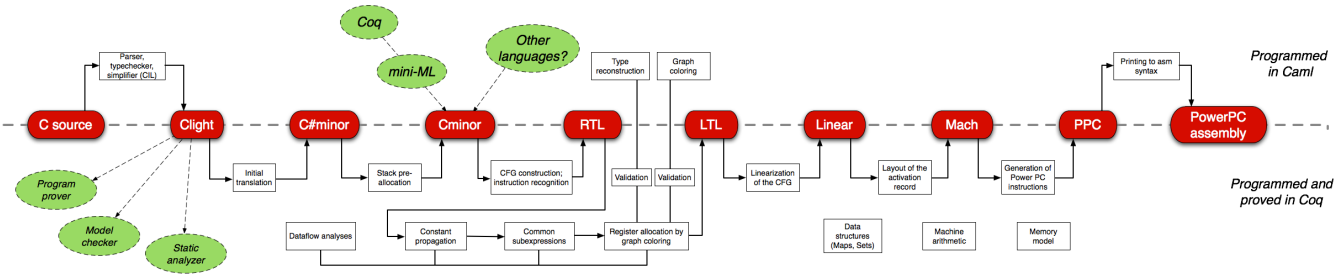


Figure 8.1: Flow diagram of CompCert C

Chapter 9

Complete Programming Stack Verification

A computing programming stack being referred here is basically classical toolchain that includes all the tools from low level to high level. It includes compilers, linkers, debuggers, Database Management System (DBMSS) and interfacing kernels. The idea of stack verification is to check the correctness of all these technologies. Assuming the logical model of each of the software we can use the reasoning of Curry-Howard correspondence and Hoare Logic to construct a verifiable proof for each one.

9.1 DeepSpec Project

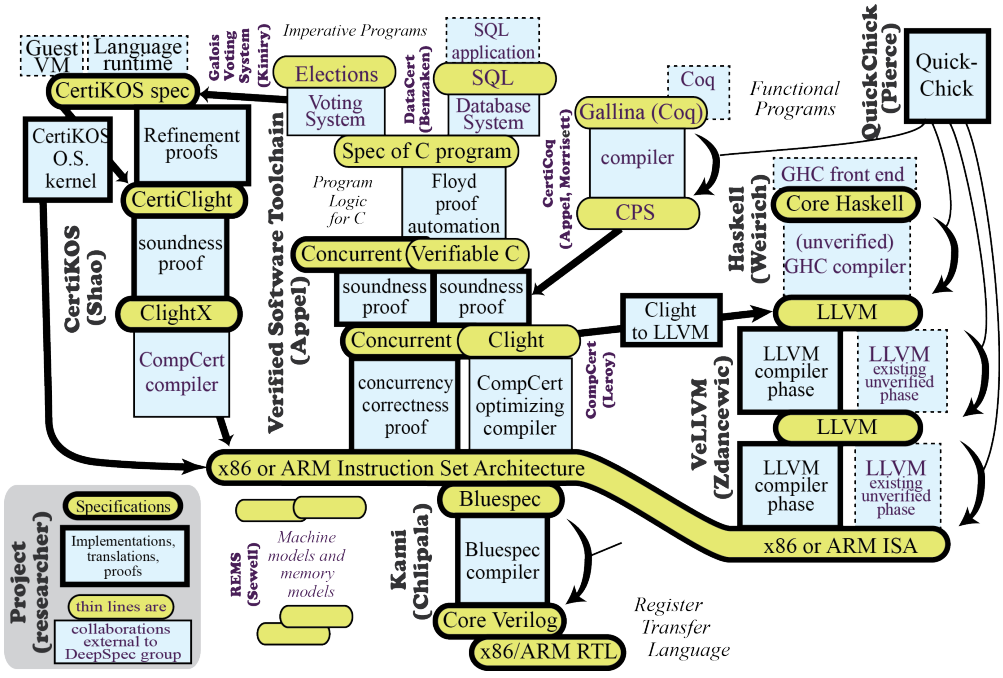


Figure 9.1: The DeepSpec Project

This project is umbrella project of many important projects like CertiCoq, VST, VELLVM, CertiKOS

in the field of software verification. The subprojects deal with checking different software tools and also the operating system semantics. Thus the project DeepSpec overall verifies the computer stack.

A very basic explaination reads as provided in the manual, :

At the top of the diagram are specifications of application-level interfaces such as the semantics of the SQL query language or the rules governing fair Elections. At the bottom is the semantics of the RTL (register-transfer language) implementation of some specific microprocessor, say x86. A voting or database system is implemented as a C program with a specification Spec of C program, written in a program logic called Verifiable C. We prove (in Coq) that the C program correctly implements SQL by applying the program logic to the program, with the assistance of the Floyd proof automation system. Verifiable-C is a set of reasoning rules for program correctness; whatever properties are proved about a program will actually hold when the program runs; this is formalized in Coq as the soundness proof connecting Verifiable C to the operational semantics of C-light. The CompCert optimizing compiler compiles C light to machine language in ARM, PowerPC, or x86-ISA. The correctness of CompCert's translation is proved in Coq. Finally, the hardware is correct: the instruction set architecture x86-ISA is specified in the Bluespec hardware-description language, which itself has a deep specification defining its precise semantics. The Kami synthesizer translates this to register transfers written in Verilog (which, again, itself has a deep specification).

It would be a major breakthrough to completely build a coherent system of specified/verified components whose specification deals only with the semanics of some very basic technologies like for example Verilog and SQL.

Chapter 10

Quantum Computing Overview

10.1 Qubits

10.1.1 Representation

10.1.2 States and Entanglement

10.2 Quantum Gates

Chapter 11

Quantum Computing Programming Languages

Chapter 12

Interplay between Quantum Computing and Type Theory

Chapter 13

Applications in Machine Learning

13.1 Automated Theorem Provers

13.2 Teaching AI to Prove Theorems

13.3 Automated Reasoning for formalization

Chapter 14

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.