Introduction to Quantum Computing

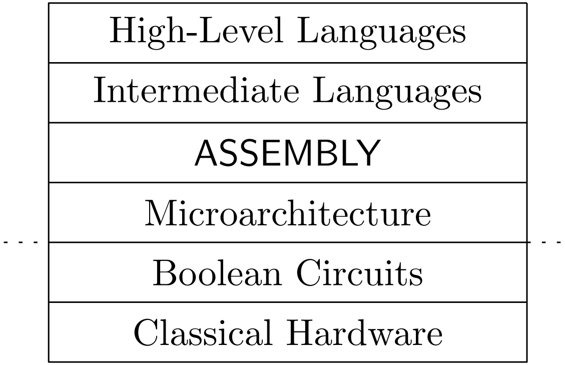
Module 2 — Part IV

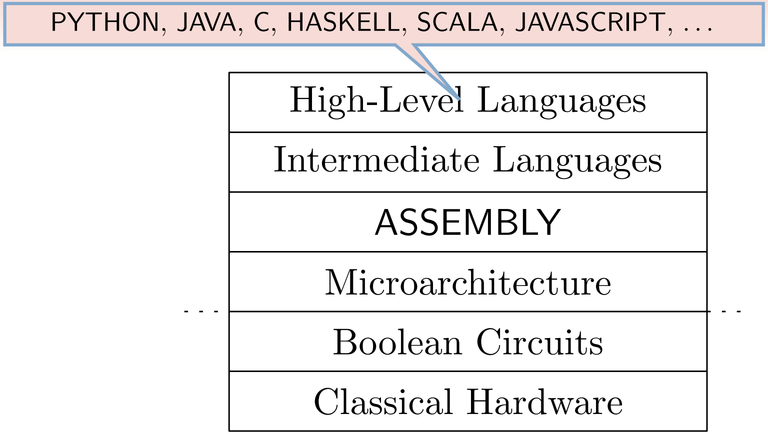
Ugo Dal Lago

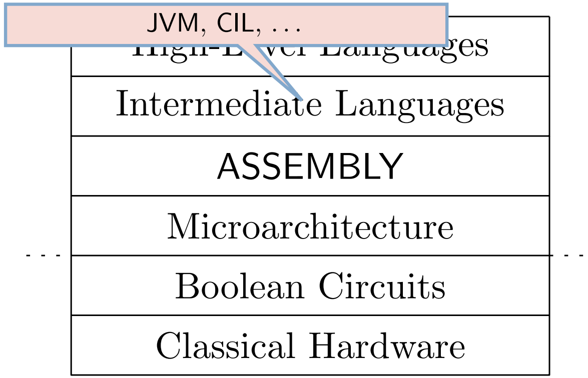
Academic Year 2024/2025

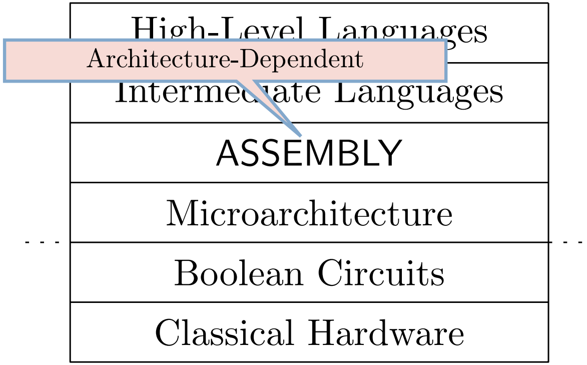
Part I

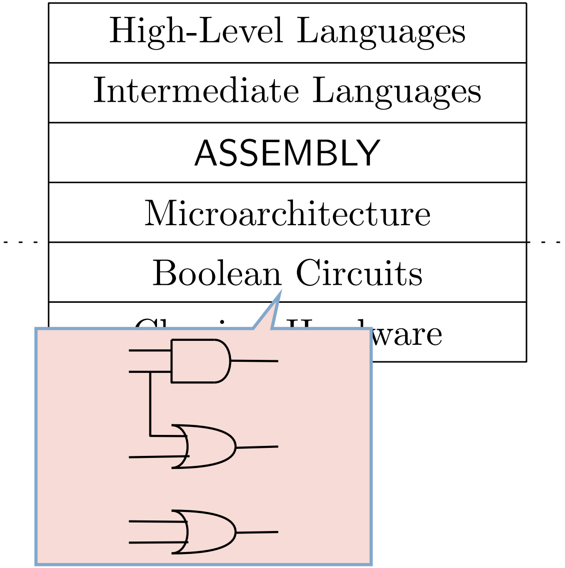
Quantum Programming Languages

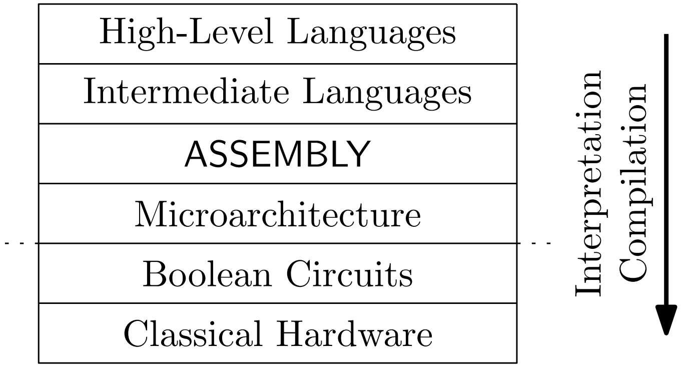

High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware

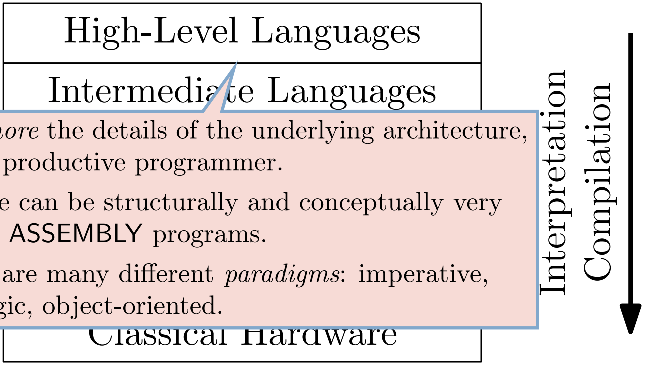

PYTHON, JAVA, C, HASKELL, SCALA, JAVASCRIPT, . . .
High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware


JVM, CIL, . . .
High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware

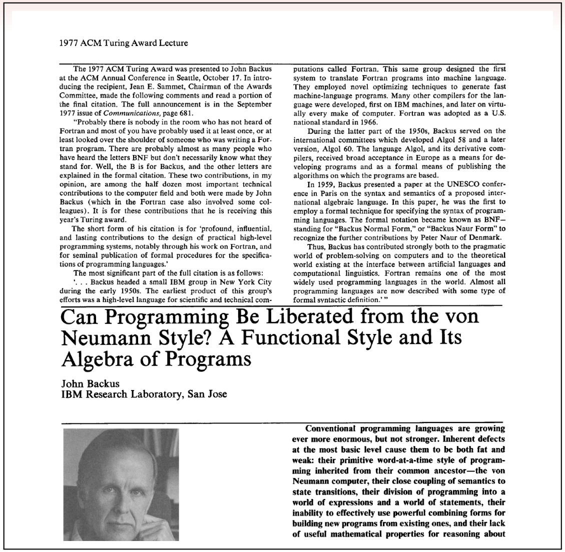

High-Level Languages
Architecture-Dependent
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware


High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits


High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware
Interpretation
Compilation



- One could *ignore* the details of the underlying architecture, and still be a productive programmer.
- Programs here can be structurally and conceptually very different from ASSEMBLY programs.
- In fact, there are many different *paradigms*: imperative, functional, logic, object-oriented.



The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications*, page 681.

"Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for 'profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages.'

The most significant part of the full citation is as follows:

'. . . Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computations called Fortran. This same group designed the first system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. national standard in 1966.

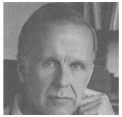
During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 60. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form" to recognize the further contributions by Peter Naur of Denmark.

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational linguistics. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.' "

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

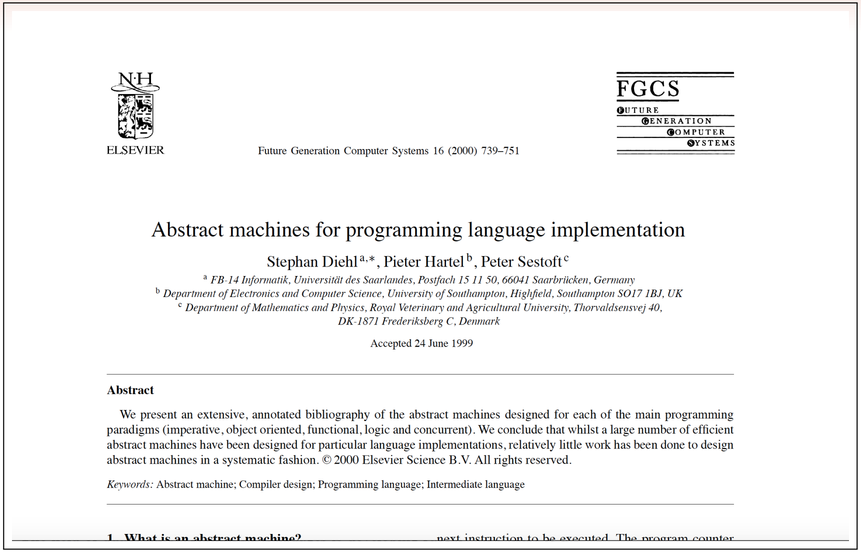
John Backus
IBM Research Laboratory, San Jose



Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about

JOHN BACKUS

ABSTRACT MACHINES



ELSEVIER

Future Generation Computer Systems 16 (2000) 739–751

FGCS
FUTURE
GENERATION
COMPUTER
SYSTEMS

Abstract machines for programming language implementation

Stephan Diehl[a,*], Pieter Hartel[b], Peter Sestoft[c]

[a] *FB-14 Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany*
[b] *Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, UK*
[c] *Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark*

Accepted 24 June 1999

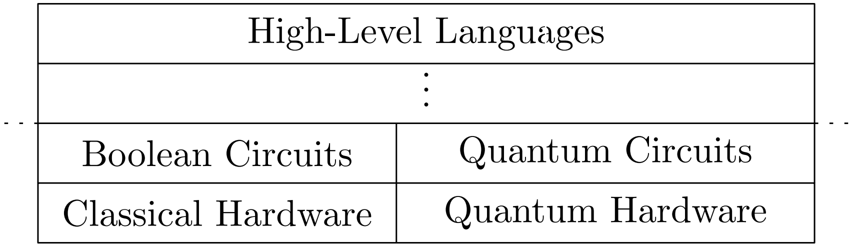
Abstract

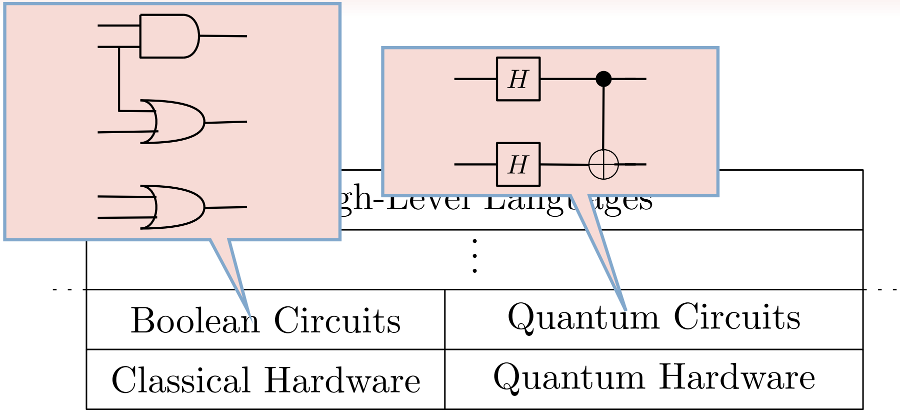
We present an extensive, annotated bibliography of the abstract machines designed for each of the main programming paradigms (imperative, object oriented, functional, logic and concurrent). We conclude that whilst a large number of efficient abstract machines have been designed for particular language implementations, relatively little work has been done to design abstract machines in a systematic fashion. © 2000 Elsevier Science B.V. All rights reserved.

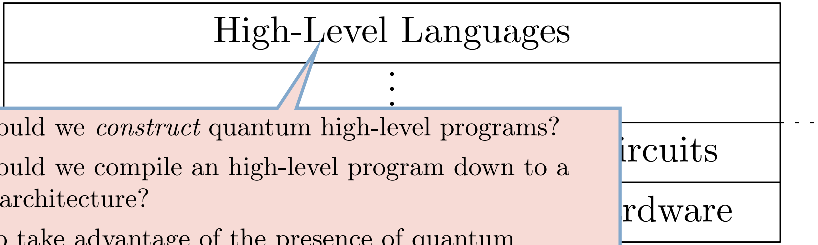
Keywords: Abstract machine; Compiler design; Programming language; Intermediate language

1. What is an abstract machine?

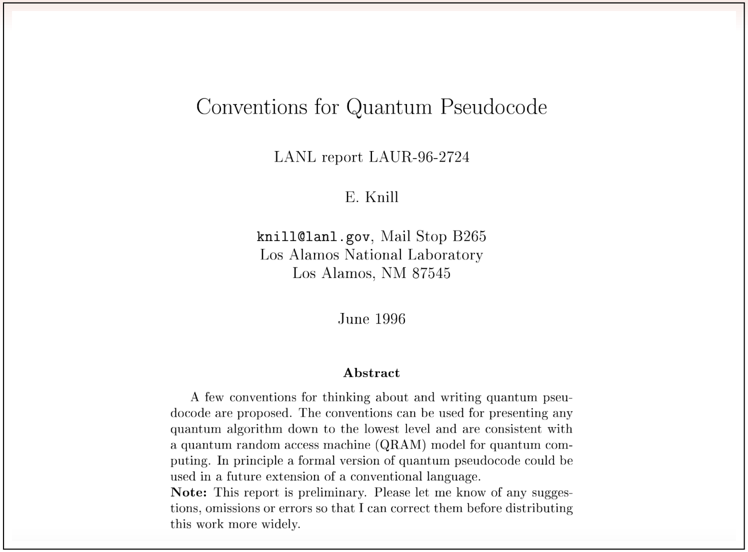
next instruction to be executed. The program counter


High-Level Languages
⋮
Boolean Circuits
Quantum Circuits
Classical Hardware
Quantum Hardware


H
H
gh-Level Lang ges
⋮
Boolean Circuits
Quantum Circuits
Classical Hardware
Quantum Hardware



- How could we *construct* quantum high-level programs?
- How could we compile an high-level program down to a *mixed* architecture?
- How to take advantage of the presence of quantum circuits, and of the computation power they provide?



Conventions for Quantum Pseudocode

LANL report LAUR-96-2724

E. Knill

knill@lanl.gov, Mail Stop B265
Los Alamos National Laboratory
Los Alamos, NM 87545

June 1996

Abstract

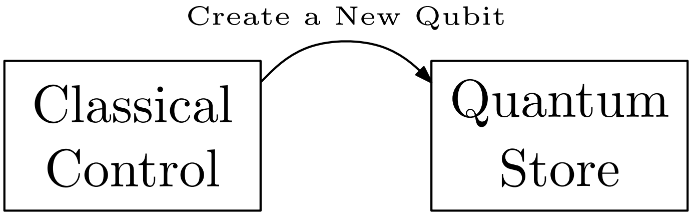
A few conventions for thinking about and writing quantum pseudocode are proposed. The conventions can be used for presenting any quantum algorithm down to the lowest level and are consistent with a quantum random access machine (QRAM) model for quantum computing. In principle a formal version of quantum pseudocode could be used in a future extension of a conventional language.
Note: This report is preliminary. Please let me know of any suggestions, omissions or errors so that I can correct them before distributing this work more widely.

THE QRAM MODEL

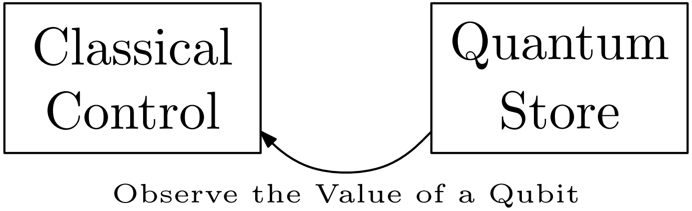
Quantum Data and Classical Control

Classical Control
Quantum Store

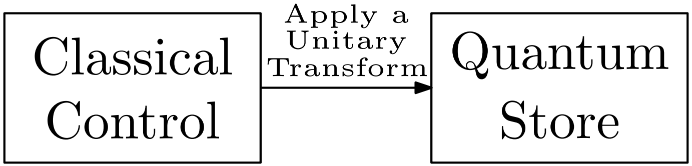
Quantum Data and Classical Control



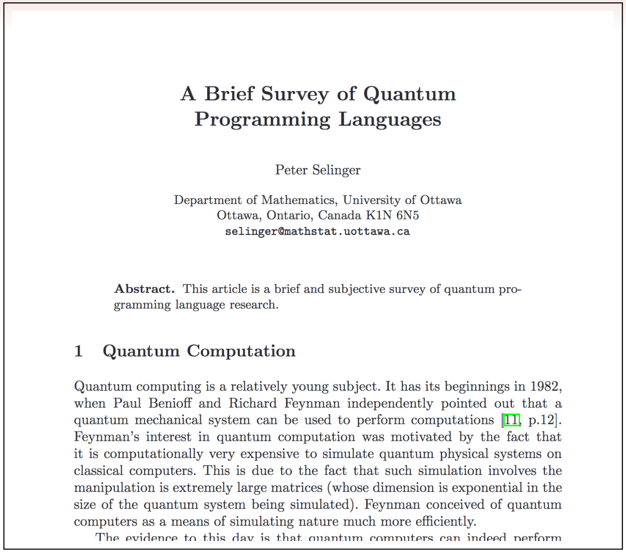
Quantum Data and Classical Control



Quantum Data and Classical Control



A SURVEY



A Brief Survey of Quantum Programming Languages

Peter Selinger

Department of Mathematics, University of Ottawa
Ottawa, Ontario, Canada K1N 6N5
`selinger@mathstat.uottawa.ca`

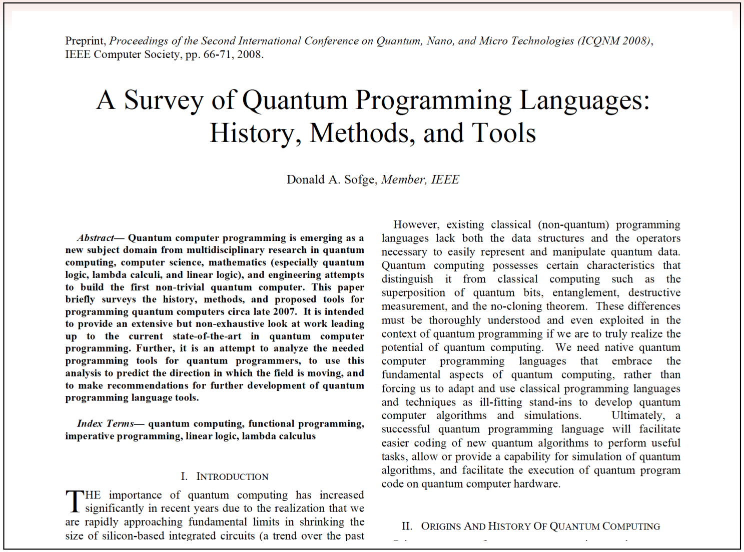
Abstract. This article is a brief and subjective survey of quantum programming language research.

1 Quantum Computation

Quantum computing is a relatively young subject. It has its beginnings in 1982, when Paul Benioff and Richard Feynman independently pointed out that a quantum mechanical system can be used to perform computations [11, p.12]. Feynman's interest in quantum computation was motivated by the fact that it is computationally very expensive to simulate quantum physical systems on classical computers. This is due to the fact that such simulation involves the manipulation is extremely large matrices (whose dimension is exponential in the size of the quantum system being simulated). Feynman conceived of quantum computers as a means of simulating nature much more efficiently.

The evidence to this day is that quantum computers can indeed perform

ANOTHER SURVEY



Preprint, *Proceedings of the Second International Conference on Quantum, Nano, and Micro Technologies (ICQNM 2008)*, IEEE Computer Society, pp. 66-71, 2008.

A Survey of Quantum Programming Languages: History, Methods, and Tools

Donald A. Sofge, *Member, IEEE*

***Abstract*— Quantum computer programming is emerging as a new subject domain from multidisciplinary research in quantum computing, computer science, mathematics (especially quantum logic, lambda calculi, and linear logic), and engineering attempts to build the first non-trivial quantum computer. This paper briefly surveys the history, methods, and proposed tools for programming quantum computers circa late 2007. It is intended to provide an extensive but non-exhaustive look at work leading up to the current state-of-the-art in quantum computer programming. Further, it is an attempt to analyze the needed programming tools for quantum programmers, to use this analysis to predict the direction in which the field is moving, and to make recommendations for further development of quantum programming language tools.**

***Index Terms*— quantum computing, functional programming, imperative programming, linear logic, lambda calculus**

I. Introduction

THE importance of quantum computing has increased significantly in recent years due to the realization that we are rapidly approaching fundamental limits in shrinking the size of silicon-based integrated circuits (a trend over the past

However, existing classical (non-quantum) programming languages lack both the data structures and the operators necessary to easily represent and manipulate quantum data. Quantum computing possesses certain characteristics that distinguish it from classical computing such as the superposition of quantum bits, entanglement, destructive measurement, and the no-cloning theorem. These differences must be thoroughly understood and even exploited in the context of quantum programming if we are to truly realize the potential of quantum computing. We need native quantum computer programming languages that embrace the fundamental aspects of quantum computing, rather than forcing us to adapt and use classical programming languages and techniques as ill-fitting stand-ins to develop quantum computer algorithms and simulations. Ultimately, a successful quantum programming language will facilitate easier coding of new quantum algorithms to perform useful tasks, allow or provide a capability for simulation of quantum algorithms, and facilitate the execution of quantum program code on quantum computer hardware.

II. Origins And History Of Quantum Computing

A Multitude of Idioms

- **Functional Languages**
- **Imperative Languages**
- **Logic Programming Languages**
- **Quantum Constraint Programming Languages**
- ...

Quantum Circuit Description Languages

- Over the past decade, we have witnessed the introduction (and use) of programming languages in which programs are not designed to execute (quantum) instructions, but to rather **build quantum circuits**, to be then executed by quantum hardware architectures or simulators.

Quantum Circuit Description Languages

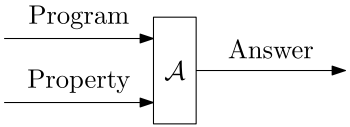
- Over the past decade, we have witnessed the introduction (and use) of programming languages in which programs are not designed to execute (quantum) instructions, but to rather **build quantum circuits**, to be then executed by quantum hardware architectures or simulators.
- It is clear that in this way **any classical programming language** can become a quantum language, simply through the implementation of libraries designed for the manipulation of circuits.
  - This is the case of `Qiskit`, `Cirq`, `Quipper`, . . .

Part II

Quantum Program and System Verification

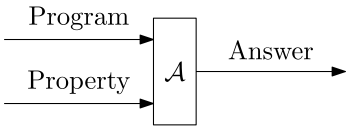
Classic Program Verification

- In **classic** program verification, one is interested in checking whether a program satisfies a property:



Classic Program Verification

- In **classic** program verification, one is interested in checking whether a program satisfies a property:

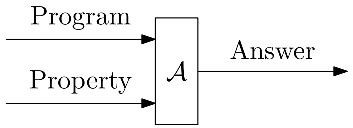


. . . where:

- The property specifies the expected behaviour of the program, in the form of *functional* or *non-functional* specifications.
- The answer, typically, is either `YES`, `NO`, or `UNKNOWN`.
- $\mathcal{A}$ can be an algorithm, or some kind of analytic methodology.

Classic Program Verification

- In **classic** program verification, one is interested in checking whether a program satisfies a property:



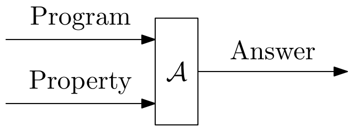
Whether the program guarantees a certain **“quality of service”**, or not.

Whether the program does **what** it is supposed to do (or not).

  - The property specifies the expected behaviour of the program, in the form of *functional* or *non-functional* specifications.
  - The answer, typically, is either `YES`, `NO`, or `UNKNOWN`.
  - $\mathcal{A}$ can be an algorithm, or some kind of analytic methodology.

Classic Program Verification

- In **classic** program verification, one is interested in checking whether a program satisfies a property:



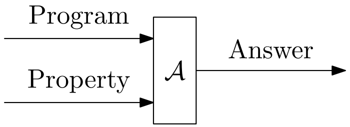
…where:

- The property specifies t rogram, in the form of *functional* or *non-functional* specifications.
- The answer, typically, is either `YES`, `NO`, or `UNKNOWN`.
- $\mathcal{A}$ can be an algorithm, or some kind of analytic methodology.

A precise analysis is in most cases impractical.

Classic Program Verification

- In **classic** program verification, one is interested in checking whether a program satisfies a property:



...where:

  - The property specifies the expected behaviour of the program, in the form of *functional* or *non-functional* specifications.
  - The answer, typically, is either `YES`, `NO`, or `UNKNOWN`.
  - $\mathcal{A}$ can be an algorithm, or some kind of analytic methodology.

- This can be done in high-level languages **only if** the property is preserved through compilation or interpretation.
  - Most functional properties are preserved.
  - Many non-functional properties (e.g. performances) requires more care.

Examples of Program Verification Techniques

- **Program Logics**
  - Very popular for imperative programs.
  - One prove, deductively, the validty of triples in the form $\vdash \{\Phi\}\mathtt{P}\{\Psi\}$, meaning that the program `P` when run on an input state satisfying Φ, ends up in a state satisfying Ψ.
  - Particularly suited for proving functional properties, but adapted to non-functional ones.

Examples of Program Verification Techniques

- **Program Logics**
  - Very popular for imperative programs.
  - One prove, deductively, the validty of triples in the form $\vdash \{\Phi\}\mathtt{P}\{\Psi\}$, meaning that the program $\mathtt{P}$ when run on an input state satisfying Φ, ends up in a state satisfying Ψ.
  - Particularly suited for proving functional properties, but adapted to non-functional ones.
- **Model Checking**
  - A system or program $\mathtt{S}$ is seen as a structure interpreting a formula A written in modal logics, e.g., temporal logics.
  - One then checks whether $\mathtt{S} \models \mathsf{A}$ in an exhaustive way, i.e., by checking in which states of $\mathtt{S}$ the formula A (and its subformulas) hold, and then checking whether the initial states of S are among those.

Examples of Program Verification Techniques

- **Program Logics**
  - Very popular for imperative programs.
  - One prove, deductively, the validty of triples in the form $\vdash \{\Phi\}\mathtt{P}\{\Psi\}$, meaning that the program `P` when run on an input state satisfying Φ, ends up in a state satisfying Ψ.
  - Particularly suited for proving functional properties, but adapted to non-functional ones.
- **Model Checking**
  - A system or program `S` is seen as a structure interpreting a formula A written in modal logics, e.g., temporal logics.
  - One then checks whether $\mathtt{S} \models \mathsf{A}$ in an exhaustive way, i.e., by checking in which states of `S` the formula A (and its subformulas) hold, and then checking whether the initial states of S are among those.
- **Type Systems**
  - Supported by most functional programming languages
  - one derives judgments in the form $\Gamma \vdash \mathtt{P} : A$ where A is a type and Γ attributes a type to each of the variables which `P` refers to.
  - Originally meant to guarantee safety ("Well-typed programs cannot go wrong"), but then generalized to other functional and non-functional properties

Quantum Program Verification: Challenges

- Verifying quantum programs and system can be **rewarding**:
  - Testing them can be very expensive!
  - Certain properties are arguably of paramount importance, like resource consumption.

Quantum Program Verification: Challenges

- Verifying quantum programs and system can be **rewarding**:
  - Testing them can be very expensive!
  - Certain properties are arguably of paramount importance, like resource consumption.
- On the other hand, verifying quantum programs against functional or nonfunctional properties is **challenging** for a number of reasons:
  - The underlying computational model is simply different, and more complicated.
  - Besides the usual exponential blowup (there are $\Theta(2^n)$ states of size n), there is another one coming from superposition.
  - Most quantum algorithms are only approximately correct.