UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Evandro Chagas Ribeiro da Rosa

Ket Quantum Programming

Florianópolis
2021

Evandro Chagas Ribeiro da Rosa

Ket Quantum Programming

Dissertation submitted to the Programa de Pós-Graduação em Ciência da Computação of the Universidade Federal de Santa Catarina to obtain the title of Master in Computer Science.
Supervisor: Prof. Rafael de Santiago, Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Ribeiro da Rosa, Evandro Chagas
   Ket Quantum Programming / Evandro Chagas Ribeiro da
Rosa ; orientador, Rafael de Santiago, 2021.
   109 p.

   Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico, Programa de Pós-Graduação em
Ciência da Computação, Florianópolis, 2021.

   Inclui referências.

   1. Ciência da Computação. 2. Quantum Computation. 3.
Quantum Programming. 4. Quantum Simulation. I. de
Santiago, Rafael. II. Universidade Federal de Santa
Catarina. Programa de Pós-Graduação em Ciência da Computação.
III. Título.

Evandro Chagas Ribeiro da Rosa

Ket Quantum Programming

The present work at master's level was evaluated and approved by an examining board composed of the following members:

We certify that this is the **original and final** version of the conclusion work that was considered adequate to obtain the title of Master in Computer Science.

Coordination of the
Graduate Program

Prof. Rafael de Santiago, Dr.
Supervisor

Florianópolis, 2021.

ACKNOWLEDGEMENTS

Agradeço aos meus pais, Maria Joana Chagas e Vanderlei Ribeiro da Rosa, por sempre me darem suporte e incentivo aos meus estudos. Seu apoio foi fundamental para que eu pudesse começar e concluir este trabalho.

Agradeço ao meu orientador, Professor Rafael de Santiago, por acreditar no meu projeto e pela amizade que construímos durante o desenvolvimento deste trabalho.

Agradeço aos integrantes do Grupo de Computação Quântica – UFSC pela amizade e pelo espaço aberto para que eu pudesse apresentar e melhorar este trabalho.

Agradeço ao Otto Menegasso Pires por testar as ferramentas que desenvolvi neste trabalho, me ajudando a corrigir problemas e aperfeiçoá-las.

Agradeço aos professores que compuseram a banca de avaliação deste trabalho, Juliana Kaizer Vizzotto, Samuel da Silva Feitosa e Ricardo Azambuja Silveira, pelas valiosas sugestões e contribuições.

Agradeço ao meu amor, Patricia Corrêa Candido, pela companhia durante todos os dias desta jornada. Te amo.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior- Brasil (CAPES) – Código de Financiamento 001.

ABSTRACT

Quantum programming languages fill the gap between quantum mechanics and classical programming constructions to simplify the development of quantum applications. However, most quantum programming languages only address the inherent quantum programming constraints without observing the construction restrictions of quantum computers. Due to decoherence, cloud-based quantum computers must run as fast as possible, which leads to batch processing, limiting the interaction between classical and quantum computers. In this work, we present Ket, a Python-embedded quantum programming language for hybrid classical-quantum programming that mitigates this interaction limitation with a runtime architecture suitable for cloud-based quantum computers. As the core of our proposed runtime architecture, we have the C++ runtime library Libket, which features runtime quantum code generation to enable generic quantum programming with dynamic quantum execution while keeping the quantum computation as specific as possible. Libket also introduces the `future` variables to delay the quantum execution, which Ket used to mitigate the interaction limitation between classical and quantum computers. Ket, Libket, and Ket Bitwise (quantum computer) Simulator (KBW) constitute the Ket Quantum Programming framework. With KBW, we improve over the Bitwise representation, associating the simulation time to the amount of superposition and entanglement in the quantum system, not the number of qubits.

Keywords: Quantum Computation. Quantum Programming. Quantum Simulation.

RESUMO

As linguagens de programação quântica preenchem a lacuna entre a mecânica quântica e as construções clássicas de programação para simplificar o desenvolvimento de aplicações quânticas. No entanto, a maioria das linguagens de programação quântica abordam apenas as restrições intrínsecas à programação quântica sem observar as restrições advindas da construção dos computadores quânticos. Devido à decoerência, computadores quânticos em nuvem devem executar o mais rápido possível, possibilitando apenas o processamento em lote que limita a interação entre computadores clássicos e quânticos. Neste trabalho, apresentamos o Ket, uma linguagem de programação quântica embarcada em Python para programação híbrida clássica-quântica que mitiga essa limitação de interação com uma arquitetura de tempo de execução adequada para computadores quânticos em nuvem. Como componente central da arquitetura de tempo de execução proposta, apresentamos a biblioteca C++ Libket, que introduz geração de código de tempo de execução para possibilitar a programação quântica genérica com execução dinâmica, enquanto mantém a computação quântica o mais específica possível. O Libket também introduz as variáveis do tipo `future` para atrasar a execução quântica. Estas variáveis, por sua vez, são utilizadas pelo Ket para mitigar a limitação de interação entre computadores clássico e quântico. Ket, Libket e o Ket Bitwise Simulator (KBW) compõem o framework de programação quântica Ket. Com o KBW, melhoramos a representação Bitwise, tornando o tempo de simulação não dependente do número de qubits, mas sim da quantidade de superposição e emaranhamento do sistema.

Palavras-chave: Computação Quântica. Programação Quântica. Simulação Quântica.

RESUMO EXPANDIDO

Introdução

Motivado pela dificuldade em simular a evolução de sistemas quânticos, Feynman (1982) conjecturou que um computador que use fenômenos da computação quântica, tais como superposição e entrelaçamento, poderia resolver alguns problemas mais rápido do que computadores convencionais. Chamamos tal computador de computador quântico. Em contraste, chamamos de computadores clássicos aqueles que não dependem do modelo quântico. Os trabalhos primordiais de Shor (1997) e Grover (1997) confirmaram essa vantagem dos computadores quânticos sobre os clássicos. O algoritmo de Shor usa uma sub-rotina quântica para resolver o problema da fatoração em tempo polinomial. Um problema sem algoritmo polinomial clássico conhecido. O algoritmo de Grover, por suas vez, é um algoritmo com tempo $O(\sqrt{n})$ para o problema da busca em base de dados desordenada, conhecido por ter um limite inferior clássico de $O(n)$. Hoje, o site Quantum Algorithm Zoo (JORDAN, 2021) lista mais de 60 algoritmos quânticos para álgebra, simulação, otimização, aprendizado de máquina e mais, com um ganho de até superpolinomial quando comparado com sua contraparte clássica.

Hoje, estamos na era NISQ (do inglês *Noisy Intermediate-Scale Quantum*) (PRESKILL, 2018), com computadores quânticos com apenas algumas dezenas de qubits e operando com portas quânticas de baixa fidelidade. Nesta era, o número de qubits e portas quânticas limitam a execução de diversos algoritmos quânticos. Por exemplo, até mesmo instâncias pequenas dos algoritmos de Shor e Grover, que caberiam em um computador quântico de hoje em dia, tem dificuldades com a decoerência, perdendo as informações dos qubits para o ruído devido a baixa fidelidade das portas quânticas. No entanto, já passamos o marco da vantagem quântica (ARUTE et al., 2019), onde um computador quântico consegue superar a performance de um supercomputador com vantagem exponencial para resolver algum problema (não necessariamente útil) (PRESKILL, 2012).

A maioria das realizações físicas de computadores quânticos requer condições específicas para funcionar corretamente, como por exemplo, deve estar isolado de quase qualquer ruído, o que eleva o custo de infraestrutura e manutenção a computadores quânticos. Essas condições tornam computadores quânticos adequados para a computação em nuvem, onde empresas e instituições podem usar de computação quântica sob demanda, cortando custos de infraestrutura e manutenção. Este modelo é implementado hoje, com provedores de computação quântica em nuvem, tais como a Amazon Braket e a Microsoft Azure Quantum, representando a maioria dos provedores de hardware quântico. No entanto, como o tempo de coerência de um computador quântico em nuvem é normalmente menor que a latência entre este computador com um computador clássico, execuções quânticas são escalonadas em lote, proibindo a interação entre computadores durante a computação quântica.

Objetivos

O objetivo geral deste trabalho é desenvolver uma nova linguagem e ambiente para programação híbrida clássica-quântica, que mitigue a limitação de interação entre computadores classico e quântico em nuvem. Adicionalmente, a nova linguagem de

programação deve garantir a não violação de aspectos intrínsecos da computação quântica, tais como o teorema da não-clonagem (WOOTTERS; ZUREK, 1982) e a computação quântica reversível. Para alcançar esses objetivos, listamos quatro objetivos específicos:

- O1 No contexto de programação híbrida clássica-quântica, com um computador quântico processando em lote, o primeiro objetivo específico deste trabalho é desenvolver um modelo para mitigar a limitação de interação entre computadores clássico e quântico em nuvem;
- O2 Com um computador quântico tendo que executar o mais rápido o possível para diminuir o efeito da decoerência, nosso segundo objetivo específico é desenvolver um modelo para permitir programação quântica genérica com execução dinâmica enquanto mantém a computação quântica o mais específica o possível, evitando enviar para o computador quântico instruções que possam ser executadas no computador clássico;
- O3 Para testar e validar nossa proposta, o terceiro objetivo específico deste trabalho é implementar a linguagem de programação quântica proposta;
- O4 Para compor o ambiente de execução quântica e para validar nossa proposta, o último objetivo específico deste trabalho é desenvolver um simulador de computação quântica com a mesma limitação de interação de computadores quânticos em nuvem.

Metodologia
Nossa pesquisa é qualitativa, uma vez que seu foco é encontrar novas maneiras para facilitar a programação de aplicações clássica-quântica através da abstração de peculiaridades e restrições da computação quântica para construções mais familiares de programação clássica. Para alcançar os objetivos deste trabalho, usamos os seguinte métodos:

- Com o nosso cenário e perguntas de pesquisa definidas, nós buscamos, na literatura, por linguagens de programação quântica que possuíssem uma implementação. Então, verificamos como essas linguagem se encaixam no nosso cenário e se elas respondem as questões de pesquisa. No entanto, nenhuma das linguagens pesquisadas atingiu totalmente os requisitos.
- Para responder nossas questões de pesquisa, primeiramente, definimos uma arquitetura de tempo de execução e, inspirado em programação concorrente, introduzimos o tipo `future` para gerenciar medidas quântica e informações clássicas no computador quântico.
- Para testar nossa proposta, desenvolvemos os componentes essenciais da arquitetura de tempo de execução: a biblioteca de tempo de execução (Libket) e um simulador quântico (KBW) implementando as mesmas restrições de um computador quântico em nuvem.
- Primeiramente, nossa intenção era usar o Libket como a biblioteca de tempo de execução de uma nova linguagem de programação quântica desenvolvida do zero, totalmente integrada com as variáveis do tipo `future`. No entanto, decidimos implementar a nova linguagem de programação Ket baseado na linguagem Python,

uma vez que conseguimos integrar as variáveis do tipo `future` com a maior parte das construções da linguagem. Desta forma, incluímos todos os benefícios do Python no Ket, deixando a linguagem compatível com bibliotecas consolidadas como NumPy (HARRIS et al., 2020) e SciPy (VIRTANEN et al., 2020).

- Com o Ket, nós podemos efetivamente testar e validar nossa proposta, implementando diversos algoritmos quânticos, como o teletransporte quântico, o algoritmo de Shor, o algoritmo de Grover, QAOA, e mais. Também implementamos parte do Microsoft Q# Coding Contest para comparar o Ket com a linguagem Q# (SVORE, K. et al., 2018). Essas implementações também ajudaram a depurar e melhorar o Ket, Libket, e o simulador KBW.
- A implementação do simulador quântico (KBW) foi baseada na representação Bitwise (ROSA; TAKETANI, 2020), com a adição de uma otimização inspirada pelo simulador Qrack (STRANO et al., 2020). Avaliamos a performance do KBW em relação a diversos simuladores do estado da arte.
- Desenvolvemos a biblioteca Libket em C++ 17 usando a biblioteca Boost, e para a interface com Python, nós usamos o software SWIG (BEAZLEY, 1996). Para o Ket, nós usamos Python com linguagem base, construindo muito das suas funcionalidades em cima da interface produzida pelo SWIG. Implementamos transformações sobre a árvore de sintaxe abstrata do Python para integrar o tipo `future` com as expressões `if` e `while` do Python. O Libket se comunica com o KBW através de uma API REST. Usamos Python com Flask para desenvolver o servidor do KBW e ANTLR 4 (PARR, 2013) em C++ para criar o analisador sintático e executar o código quântico. Nós também usamos a biblioteca C++ Boost para implementar a representação Bitwise no KBW.

Resultados e Discussão

Um produto derivado desta pesquisa é o framework de programação quântica Ket, um projeto de código aberto composto pela linguagem de programação clássica-quântica embarcada em Python Ket, a biblioteca C++ de tempo de execução Libket, e o simulador de computação quântica Ket Bitwise Simulator (KBW). Tornamos todos os código fonte disponivel em `https://gitlab.com/quantum-ket`, incluindo documentação em `https://quantum-ket.gitlab.io`. A linguagem e o simulador estão em uso no Grupo de Computação Quântica da Universidade Federal de Santa Catarina, e já há trabalhos publicados usando o framework.

Nossa pesquisa também deu suporte à pesquisa científica de Pires et al. (2021). A qual usa o algoritmo QAOA para encontrar uma solução aproximada para o problema da alocação de horários (*school timetabling problem*) através de uma redução para o problema de coloração de grafos. Este trabalho implementa o algoritmo usando a linguagem de programação Ket, e usa o Simulador KBW para executar um experimento usando 42 qubits. Pires et al. (2021) tem um artigo intitulado "*Two Stage Quantum Optimization for the School Timetabling Problem*" publicado na conferência 2021 IEEE Congress on Evolutionary Computation (CEC).

Este trabalho também gerou uma publicação intitulada "*Ket Quantum Programming*". aceita para ser publicada na revista científica ACM Journal on Emerging Technologies in Computing Systems (JETC) na edição especial Design Automation for Quantum Computing. Nossas principais contribuições técnicas e científicas são:

- O projeto de uma arquitetura de tempo de execução adequada para programação híbrida clássica-quântica com computadores quânticos em nuvem, implementado no framework de programação Ket.
- O projeto e a implementação da linguagem de programação clássica-quântica Ket, a qual consegue gerar códigos clássico e quântico bem separados e introduz interação dinâmica entre computadores clássicos e quânticos em nuvem, um problema que não é totalmente abordado pelos trabalhos relacionados.
- O uso de variáveis do tipo `futuro` como o retorno de medidas quânticas para atrasar a execução quântica, e a integração de maneira transparente dessas variáveis com as construções da linguagem de programação, a fim de controlar a execução quântica.
- A melhoria na representação Bitwise, deixando o tempo de simulação não dependente do número de qubits, mas sim, dependente da quantidade de superposição e entrelaçamento do sistema.

Considerações Finais
Nós alcançamos os dois primeiros objetivos específicos deste trabalho com a biblioteca de tempo de execução Libket, e melhoramos esta solução com a linguagem de programação Ket. Através da integração das variáveis do tipo `future` com as construções `if-then-else` e `while` do Python, o Ket consegue mover o fluxo de controle clássico para o computador quântico de maneira transparente para o programador, mitigando, assim, a limitação de interação entre computadores quântico e clássico. Essa integração também permite a programação de código que pode executar tanto no computador clássico quanto no computador quântico, a depender de onde os valores estão disponíveis. Com a geração de código quântico em tempo de execução possibilitada pelo Libket, é possível limitar a execução quântica em apenas instruções que não podem ser resolvidas ou avaliadas pelo computador clássico. Permitindo, deste modo, que o Ket use qualquer construção do Python para programação clássica-quântica genérica. A implementação do Ket satisfaz o terceiro objetivo específico deste trabalho, validando os conceitos da linguagem de programação quântica proposta. O Libket é escrito apenas em C++, independe do Python e do Ket, podendo ser usado na implementação de outros softwares ou linguagem de programação quântica.

Para implementar o suporte total da linguagem Ket em computadores quânticos, é necessário que eles suportem fluxo de controle clássico. No entanto, isso ainda não é possível em computadores NISQ, mas acreditamos que computadores quânticos irão começar a ter um suporte inicial a fluxo de controle clássico em breve. Eliminando o fluxo de controle em computadores quânticos, a linguagem Ket e sua arquitetura de tempo de execução são adequadas para a execução em computadores NISQ.

Nós acreditamos que o Ket é uma maneira conveniente para programadores Python adentrarem na computação quântica, levando a dinâmica de programação do Python para a manipulação de bits quânticos. Apesar do Ket implementar manipulação de

qubits em baixo nível, assim como a linguagem Q#, é possível usar classes para implementar construções de alto nível como a descomputação da linguagem Silq (BICHSEL et al., 2020). Em execuções quânticas simuladas, o Ket pode usar as variáveis do tipo `dump` como uma ferramenta para facilitar a depuração e o estudo de algoritmos quânticos. Apesar da visualização do código quântico como um circuito quântico ainda não ter sido implementada, o Libket possibilita a extração para inspeção do código quântico gerardo.

Como trabalhos futuros, pretendemos fazer a especificação formal da linguagem Ket, apresentando as extensões que fizemos na linguagem Python, e implementar tratamento de erros semânticos e erros que possam ocorrer durante uma execução quântica. Na arquitetura de tempo de execução proposta nós não tratamos propriamente do problema da depuração de programas quânticos e há uma falta de precisão na descrição da execução em hardware quântico, problemas que também queremos tratar em trabalhos futuros.

Palavras-chave: Computação Quântica. Programação Quântica. Simulação Quântica.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Motivated by the nature of simulating the evolution of quantum states, Feynman (1982) conjectured that a computer using quantum mechanics phenomena, such as superposition and entanglement, could solve some problems faster than conventional computers. We call such a computer of a quantum computer, and in contrast, we call classical computers the ones that do not rely on the quantum model. The seminal works of Shor (1997) and Grover (1997) confirmed this advantage of quantum computers over classical ones. Shor's algorithm uses a quantum subroutine to solve the Integer Factorization problem in polynomial-time. A problem without a known classical polynomial algorithm. Grover's algorithm, on the other hand, is an $O(\sqrt{n})$ time algorithm for the unordered databases search problem, known for having an $O(n)$ classical lower bound. Today, the Quantum Algorithm Zoo (JORDAN, 2021) lists more than 60 quantum algorithms for algebra, simulation, optimization, machine learning, and more, with up to superpolynomial speedup compared with their classical counterparties.

Today, we are in the Noisy Intermediate-Scale Quantum (NISQ) (PRESKILL, 2018) era with quantum computers featuring a few dozen noisy qubits operated by low-fidelity quantum gates. In this era, the number of qubits and quantum gates restrict the execution of several quantum algorithms. For example, even small instances of Shor's and Grover's algorithms that would fit in the number of qubits of today's quantum computers struggle with decoherence, losing the quantum bits' information to noise because of low fidelity gates. However, we have passed the quantum advantage milestone (ARUTE et al., 2019), where a quantum computer outperforms a supercomputer solving a problem (not necessarily a useful one) with exponential speedup (PRESKILL, 2012).

Most physical realizations of quantum computers require specific conditions, isolated from almost any kind of noise, to work correctly, adding an infrastructure and maintenance cost to the quantum computer. Those requirements make quantum computers suitable for cloud-based computation, where companies and institutions can use quantum computation on-demand, cutting the costs of infrastructure and maintenance. This mode is implemented today, with cloud-based quantum computation providers, such as Amazon Braket and Microsoft Azure Quantum, represent most quantum computers vendors. However, as the decoherence time of a cloud-based quantum computer may be shorter than the latency between it and a classical computer, quantum executions are scheduled in batch, forbidding the interaction between the computers during the quantum computation.

1.1 MOTIVATION

The challenge of developing a quantum programming language is long known. As said by Deutsch (1985),

> “Quantum computers raise interesting problems for the design of programming languages [...]” (DEUTSCH, 1985).

However, it was only with the development of the first quantum computers in the last decade that the research in quantum programming languages takes off, with only a few examples dating before 2010. Today, manipulation of quantum superposition and entanglement in programming languages is an open problem, with most languages relying on low-level quantum instructions like quantum gates.

Along with the intrinsic quantum programming limitation, *e.g.*, the no-cloning (WOOTTERS; ZUREK, 1982) theorem and the reversible computation, quantum programmers also need to consider the constraints of the quantum computer construction when developing quantum applications. However, most high-level quantum programming languages focus only on the intrinsic limitations of quantum computation, disregarding quantum hardware limitations, like the interact limitation of cloud-based quantum computers, the focus of this work. Also, with classical-quantum algorithms like Shor’s algorithm and QAOA (FARHI et al., 2014) and protocols like the Quantum Teleportation (BENNETT et al., 1993) that require interactive quantum computation, quantum programming languages need to be open to hybrid classical-quantum programming.

1.2 RESEARCH PROBLEM

When classical and quantum computers cannot communicate fast enough for the quantum computer does not get idle, the quantum computer needs to run in batch to reduce decoherence. Cloud-based quantum computation fits this scenario. Considering that a quantum computer implements classical control flow, it can use classical information from measurements to control `if-then-else` and `while` statements. This scenario raises our first research question: in the context of hybrid classical-quantum programming, with a quantum computer processing in batch, how to implement an interactive classical-quantum application?

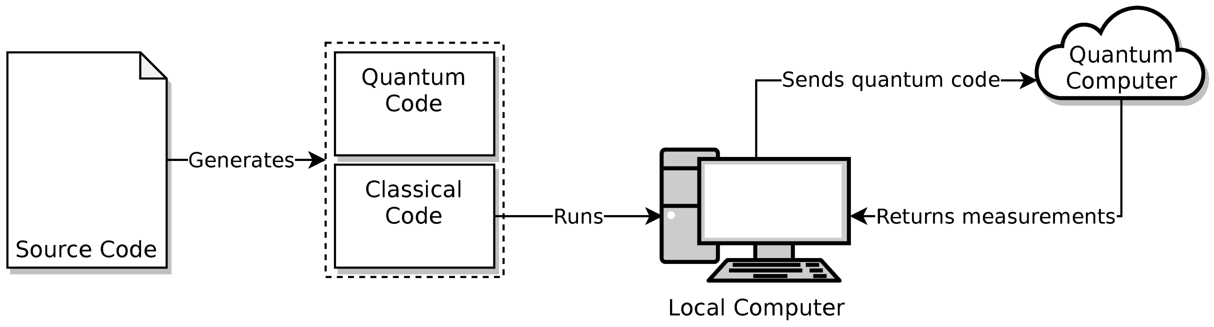
Due to decoherence, fast execution is crucial for accurate quantum computation. Therefore, the quantum computer must execute as few instructions as possible, particularly auxiliary classical operations, which imitates the quantum computer’s ability to run generic quantum applications developed for an unknown classical input. In the same classical-quantum programming scenario presented before, the second research question of this work is: how to implement a generic quantum application with dynamic execution while keeping the quantum execution as specific as possible to minimize the number of classical control statements in the quantum computer?

Adding to those questions is the restriction that a single source code needs to generate a well-separated classical and quantum part for each one to run efficiently on its respective architecture. This additional constraint makes it easier to code a classical-quantum application since there is no need to interface two different programming languages.

1.3 PROGRAMMING AND EXECUTION SCENARIO

We summarized the quantum programming and execution scenario of this work in Figure 1. In this scenario, a compiler or interpreter inputs a single source code and outputs both the classical and quantum codes. The local computer runs its corresponding code, coordinating the execution of the quantum code execution in a quantum computer available in the cloud. The quantum computer only returns measurement results and does not allow communication during the quantum execution, in other words, processing in batch, which complies with the quantum computing service provided by IBM Quantum and Amazon Braket.

Figure 1 – Our quantum programming and execution scenario.



Quantum computers also have severe time constraints due to decoherence and gate fidelity, which impose restrictions on the complexity of the quantum execution, limiting the number of operations that a quantum execution can perform before the qubits lose information. This way, any quantum code needs to be as specific as possible, making it hard to program generic and dynamic quantum applications.

One of the problems that we solved in this work is how to program an interactive classical-quantum application, *e.g.*, a quantum teleportation. Other quantum programming languages (BICHSEL et al., 2020; SMITH et al., 2017) also address this problem but not considering the presented scenario of cloud-based quantum computation. This scenario is equivalent whenever classical and quantum computers cannot communicate fast enough that the quantum computer does not get idle. The cloud is one possible and likely situation.

Another constraint not addressed by the related works is for a single source code to generate well-separated classical and quantum codes. You can use embedded languages like Quipper (GREEN et al., 2013) to implement a whole classical-quantum application, but you cannot separate the quantum and classical execution to run on their respective computers. On the other hand, Q# (SVORE, K. et al., 2018) has a well-separated quantum code. However, a general-purpose programming language is required to coordinate the quantum execution and implement the classical one. We consider that a single source code for a whole classical-quantum application facilitates its development since there is no need to interface two programming languages.

We consider that quantum computers can execute simple classical expressions and branch instructions to support the execution of loops and classical controls statements, *e.g.*, `while` and `if-then-else`. Although today's quantum computers are limited in terms of classical control flow, which impacts the classical-quantum interaction, there are experimental results of quantum computers with full dynamic execution capacity (FU et al., 2019).

1.4 OBJECTIVES

With the research problems of this work defined, we set the **primary objective** of this work as the development of a new language and environment for hybrid classical-quantum programming that mitigates the interaction limitation of cloud-based quantum computers. In addition, the new programming language must guarantee the non-violation of any aspect of quantum computation, like the no-cloning theorem (WOOTTERS; ZUREK, 1982) and reversible computation. To meet this objective, we list four **secondary objectives**:

O1 To answer the first research question, the first secondary objective of this work is to develop a model to mitigate the interaction limitation between classical and quantum computers imposed by the batch processing of cloud-based quantum computers;

O2 To answer the second research question, the second secondary objective of this work is to develop a model to enable generic quantum programming with dynamic execution while keeping the quantum computation as specific as possible, avoiding passing instructions, which the classical computer can execute, to the quantum computer;

O3 To test and validate our proposal, the third secondary objective of this work is the implementation of the proposed quantum programming language;

O4 As part of the quantum programming environment, also to test and validate our proposal, the last secondary objective of this work is to develop a quantum computer simulator with the same interaction restrictions of cloud-based quantum computers.

1.5 WORK DELIMITATION

In this work, we focus on gate-based quantum computers, not discussing the impact of our proposal in another quantum computational model like adiabatic quantum computing. We also do not approach the quantum debugging problem or any specifics for quantum hardware execution, besides the limitation implied by the cloud-based model of quantum computation.

We only test Ket on simulated quantum execution. However, we argue that it was enough to validate our proposal since we implemented the simulator with the same restrictions of cloud-based quantum computers in our scenario. We also emphasize that noise quantum simulation is out of the scope of this work.

Quantum code optimization during or afterward the quantum code generation is out of the scope of this work. Despite not being available in today's quantum computers, we assume that quantum computers can execute classical instruction from quantum code, like binary operations and branches. We expect initial support for these features soon (FU et al., 2019).

1.6 CONTRIBUTIONS

One product of this research is the Ket Quantum Programming framework, an open-source project that features the Python-embedded classical-quantum programming language Ket, the C++ runtime library Libket, and the quantum simulator Ket Bitwise Simulator (KBW). We made all the source code is available at `https://gitlab.com/quantum-ket`, including documentation at `https://quantum-ket.gitlab.io`. The language and simulator are in use in the Grupo de Computação Quântica (Quantum Computation Group) of the Universidade Federal de Santa Catarina, with already a published research using the framework.

Our work is supporting the scientific research of Pires et al. (2021), which uses the QAOA algorithm to find an approximation to the school timetabling problem, reducing it to the graph coloring problem. Their work implemented the algorithm in the Ket language, using KBW to run an execution with 42 qubits. Pires et al. (2021) have a paper titled “Two Stage Quantum Optimization for the School Timetabling Problem” published in the 2021 IEEE Congress on Evolutionary Computation (CEC).

This research also generated a paper title “Ket Quantum Programming”. accepted in the ACM Journal on Emerging Technologies in Computing Systems (JETC) for the special issue on Design Automation for Quantum Computing. This paper describes our main scientific and technical contributions:

- Design of a runtime architecture suitable for hybrid classical-quantum programming with cloud-based quantum computers, implemented in the Ket Quantum Programming framework;
- Design and implementation of the classical-quantum programming language Ket, which generates a well-separated classical and quantum code and has dynamic interaction between classical and quantum computers, problems not fully addressed by the related works;
- The use of `future` variables as the return of quantum measurements to delay the quantum execution and seamless integration of those variables with the programming language constructions to control the quantum execution, even without the measurement result;
- The improvement of the Bitwise representation, making the simulation time independent of the number of qubits but dependent on the amount of superposition and entanglement of the system.

1.7 METHODOLOGY

Our research is qualitative as it aims to find new ways to ease the programming of classical-quantum applications by abstracting the quantum peculiarities and constraints into more familiar classical programming constructions. To reach the primary objective of this work, we use the following method.

- With the scenario and research problems defined, we survey the literature for quantum programming languages with available implementation. We check how they fit in the scenario and if they answered the research questions. However, none surveyed quantum programming language fully satisfied the requirements.
- To answer the research problems, we first designed a runtime architecture and, inspired by concurrent programming, introduced the `future` construct to handle quantum measurements and classical information on the quantum computer.
- To test our proposal, we developed the essential components of the runtime architecture, the runtime library (Libket), and a quantum simulator (KBW), implementing the same restriction of cloud-based quantum computers.
- At first, we intended to use Libket as the runtime library of a new quantum programming language developed from the ground up, fully integrating the `future` variables in the language structures. However, we decided to implement the new language Ket based on Python, since it was possible to seamlessly integrate the `future` variables in most of the language's constructions. Also, including all

the benefits of Python in Ket, making the language compatible with widespread Python libraries like NumPy (HARRIS et al., 2020) and SciPy (VIRTANEN et al., 2020).

- With Ket, we could effectively test and validate our proposal by implementing several quantum applications like quantum teleportation, Shor's algorithm, Grover's algorithm, QAOA, and more. We also implemented part of the Microsoft Q# Coding Contest to compare Ket with Q#. Those implementations also helped us debug and improve Ket, Libket, and KBW.
- We have based the implementation of the quantum simulator (KBW) on the Bitwise representation (ROSA; TAKETANI, 2020), adding an optimization inspired by the simulator Qrack (STRANO et al., 2020). We evaluated the KBW performance benchmarking it against other state of the art quantum simulators.
- We developed Libket in C++ 17 using the Boost library, and for its Python wrapper, we use SWIG (BEAZLEY, 1996). For Ket, we used Python as the base language, constructing most of its functionality on top of the SWIG wrapper. We implement transformations over the Python AST to integrate the `future` type with the Python `if` and `while` statements. Libket communicates with KBW throw an API REST. We used Python with Flask for the KBW server and ANTLR 4 (PARR, 2013) in C++ the parse and execute the quantum code. We also used the C++ Boost library in KBW to implement the Bitwise representation.

1.8 DISSERTATION STRUCTURE

We present the basics of quantum computation and the postulates of quantum mechanics in Chapter 2 and introduce quantum programming particularities in Chapter 3. Still, in chapter 3, we present our related works. Our contributions are in chapters 4 and 5. First, we introduce the proposed runtime architecture and its main components, including Libket and KBW, in Chapter 4; and the new quantum programming language Ket in Chapter 5. We end in Chapter 6 with the conclusions.

This work also has two appendices. In Appendix A, we present the Python AST transformation necessary to integrate the `future` variables with the Python `if-then-else` and `while` statements. And in Appendix B, we show the implementation of some problems of the Microsoft Q# Coding Contest in Ket.

2 QUANTUM COMPUTATION

In this chapter, we introduce some relevant quantum mechanics characteristics in the context of quantum computation, highlighting their implications, benefits, and limitations. We start with an overview of quantum computation in Section 2.1, presenting the concept of quantum bit (qubit) and how to manipulate it to compute. And Section 2.2, we introduction the four postulates of quantum mechanics enumerated by Nielsen and Chuang (2010a), which gives a mathematical formalism for quantum computation.

2.1 QUANTUM BITS IN A NUTSHELL

A quantum bit or a **qubit** is the basic unit of computation of a quantum computer. Similar to a bit, a qubit has two possible states, 0 and 1, or $|0\rangle$ and $|1\rangle$ following the Dirac (1939) notation. However, different from a bit, a qubit can be at both states 0 and 1 at the same time, in what we call a **superposition**. It enables a sequence of qubits to store an exponential amount of information. For example, while a sequence with four classical bits can store an unsigned integer number between 0 to 15, four qubits can store all integers from 0 up to 5. In general, while n qubits can store 2^n bits of information, n bits only store a linear amount of information, meaning n bits. In Table 1, we show this exponential increase in the qubits storage capacity.

Table 1 – Number of bits necessary to store the information of n qubits.

n qubits	Equivalent number of bits
1	2
2	4
4	16
8	256
16	65536
32	4294967296
64	18446744073709551616
128	340282366920938463463374607431768211456
256	115792089237316195423570985008687907853269984665640564039457584007913129639936
512	13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096
1024	179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216

There is no polynomial-time method to evaluate a quantum superposition, in other words, to identify all the states (numbers) of a qubit sequence. So in quantum computation, the only viable way to collect classical information out of a quantum superposition is by performing a **measurement**. A measurement will return information about one state of a qubit superposition and collapse it. The collapse will destroy the superposition and leave the qubit in a quantum state relative to the measurement result. For example, consider a sequence of four qubits being at in a superposition of the states $|0010\rangle$, $|0100\rangle$, and $|0110\rangle$; suppose that the measurement of those qubits returns 0100, the following measurement results will always be 0100 because the qubits has collapsed to $|0100\rangle$.

A qubit in a superposition is represented by $\alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex number and $|\alpha|^2 + |\beta|^2 = 1$. The number α and β are **probability amplitudes** that ponders the random result of a measurement. For example, the measurement of the qubit $\alpha\,|0\rangle + \beta\,|1\rangle$ has probability $|\alpha|^2$ of return 0 and $|\beta|^2$ of return 1. In general terms, the measurement of a n qubit sequence $\sum_{k=0}^{2^n-1} \alpha_k\,|k\rangle$, where $\sum_{k=0}^{2^n} |\alpha_k|^2 = 1$, has probability $|\alpha_k|^2$ of return k.

In the **quantum entanglement** phenomena, two or more qubits can be in an entangled state where a single qubit cannot fully describe a part of the whole collection. By consequence, a change on one qubit affects all qubits of an entangled set. For example, taking the pair of entangled qubits $|\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, if we measure a single qubit, we know the state of the other without measuring it. The measurement of any qubit of $|\beta_{00}\rangle$ is equal to $b \in \{0,1\}$, and the collapsed state is $|bb\rangle$.

To manipulate qubits, which includes creating and destroying superposition and entanglement, or in other words, to tate a quantum bit from state $|\psi_A\rangle$ to $|\psi_B\rangle$, we use unitary operations called **quantum gates**. Examples of quantum gates are the Hadamard gate

$$\begin{aligned} H\,|0\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \\ H\,|1\rangle &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \end{aligned} \tag{1}$$

and the CNOT gate

$$\text{CNOT}\,|a\,b\rangle = |a\,(a \oplus b)\rangle\,. \tag{2}$$

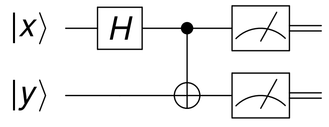
Quantum computation is **reversible** because quantum gates are time-reversible. So, we can undo any quantum computation until a measurement. Also, quantum gates cannot copy the state of a qubit to another. Because it would violate the **no-cloning theorem** (WOOTTERS; ZUREK, 1982).

A **quantum circuit**, usually represented by a diagram, describes a sequence of quantum gates and measurements. Figure 2 is an example of a quantum circuit. The lines represent qubits and the double lines classical bits (measurement result). Quantum gates are usually represented by a box that may wrap more than one qubit. We can also add control qubits to any quantum gate. For instance, the CNOT gate (Eq. (2)) is a Not gate (X gate, Eq. (3)) with a control qubit.

$$\begin{aligned} X\,|0\rangle &= |1\rangle \\ X\,|1\rangle &= |0\rangle \end{aligned} \tag{3}$$

The circuit of Figure 2 with qubits $|xy\rangle$ has the following gate order $H\,|x\rangle$ (equation (4)), CNOT $|x\,y\rangle$ (equation (2)), and measures x and y (equation (6)). With this circuit,

Figure 2 – Example of quantum circuit.



if $x = 0$ and $y = 0$, the evolution will be

$$H_x |00\rangle = \frac{|00\rangle + |10\rangle}{\sqrt{2}}, \tag{4}$$

$$\text{CNOT}\frac{|00\rangle + |10\rangle}{\sqrt{2}} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}, \tag{5}$$

$$M\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{cases} 50\% \text{ probability of measuring } 00 \\ 50\% \text{ probability of measuring } 11 \end{cases}. \tag{6}$$

2.2 THE POSTULATES OF QUANTUM MECHANICS

In this section, we explain the basics of quantum computation using the four postulates of quantum mechanics enumerated by Nielsen and Chuang (2010a). They are an attempt to represent the quantum world formally using math, more specifically, using linear algebra. The postulates define how to describe: qubits (Subsection 2.2.1); functions that changes the qubits state (Subsection 2.2.2); the probability and effect of a measure (Subsection 2.2.3); and, how to extrapolate it for multiple qubits (Subsection 2.2.4). Quantum computation uses the Dirac/bra-ket notation to represent vectors. As it is not usual outside quantum mechanics, we summarize this notation in Table 2.

Table 2 – Summary of the Dirac/bra-ket notation.

Notation	Description
z^*	Conjugate of $z \in \mathbb{C}$
A^T	Transpose of matrix A
A^*	Conjugate matrix A
$A^\dagger$	conjugate transpose of matrix A, $A^\dagger = (A^T)^*$
$\|\psi\rangle$	Column vector; pronounced “ket ψ”
$\langle\psi\|$	Dual vector of $\|\psi\rangle$; pronounced “bra ψ”; $\langle\psi\| = \|\psi\rangle^\dagger$
$\langle\varphi\|\psi\rangle$	Inner product between $\|\varphi\rangle$ and $\|\psi\rangle$
$\|\varphi\rangle \otimes \|\psi\rangle$	Tensor product between $\|\varphi\rangle$ and $\|\psi\rangle$
$\|\varphi\rangle \|\psi\rangle$	Tensor product between $\|\varphi\rangle$ and $\|\psi\rangle$
$\|\varphi\psi\rangle$	Tensor product between $\|\varphi\rangle$ and $\|\psi\rangle$

2.2.1 State Space

Postulate 1: Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space) known as the *state space* of the system. The system is completely described by its *state vector*, which is a unit vector in the system’s state space. (NIELSEN; CHUANG, 2010a, p. 80)

Based on the first postulate, we can represent *n* qubits by a vector $\in \mathbb{C}^{2^n}$ with norm 1. As mentioned in the last section, qubits can store exponentially more data than bits, which also implies an exponential scale in memory and time to simulate quantum states.

We can represent qubits by linear combinations of base vectors, usually the computational base. For example, we can express the qubit $|\psi\rangle = [\alpha\ \beta]^T$ (where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$) using the one-qubit computation base, formed by $|0\rangle = [1\ 0]^T$ and $|1\rangle = [0\ 1]^T$, like $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$.

2.2.2 System Evolution

Postulate 2: The evolution of a *closed* quantum system is described by a *unitary* transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi'\rangle$ of the system at time t_2 by a unitary operator U which depends only on the times t_1 and t_2,

$$|\psi'\rangle = U\,|\psi\rangle\,. \tag{7}$$

(NIELSEN; CHUANG, 2010a, p. 81)

By the second postulate, we can describe a quantum computation in discrete steps, each one represented by a unitary matrix with dimension 2^n, where n is the number of qubits. Again, we have an exponential speed-up in quantum computation that requires exponential resources to simulate. And, since evolution is unitary, we have the limitation that every step in a quantum computation needs to be time-reversible.

We present the matrix representation of the Hadamard (Equation (1)) and CNOT (Equation (2)) gates along with other traditional quantum gates in Table 3. We can rewrite equations

$$\begin{aligned} H\,|0\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \\ H\,|1\rangle &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \end{aligned} \tag{8}$$

and

$$\text{CNOT}\,|a\,b\rangle = |a\,(a \oplus b)\rangle\,. \tag{9}$$

as

$$\begin{aligned} \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} &= \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \\ \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} &= \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} \end{aligned} \tag{10}$$

and

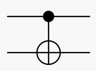
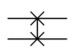
$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}a_0 b_0\\ a_0 b_1\\ a_1 b_0\\ a_1 b_1\end{bmatrix} = \begin{bmatrix}a_0 b_0\\ a_0 b_1\\ a_1 b_1\\ a_1 b_0\end{bmatrix}. \tag{11}$$

Note the matrix dimension must match the number of qubits.

2.2.3 Measurement

Postulate 3: Quantum measurements are described by a collection $\{M_m\}$ of *measurement operators*. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes

Table 3 – Traditional quantum gates.

Gate	Matrix	Effect	Circuit
Pauli X	$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$	$-\boxed{X}-$
Pauli Y	$\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$	$Y\|0\rangle = -i\|1\rangle$ $Y\|1\rangle = i\|0\rangle$	$-\boxed{Y}-$
Pauli Z	$\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$	$-\boxed{Z}-$
Hadamard	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$	$H\|0\rangle = \frac{1}{\sqrt{2}}\|0\rangle + \frac{1}{\sqrt{2}}\|1\rangle$ $H\|1\rangle = \frac{1}{\sqrt{2}}\|0\rangle - \frac{1}{\sqrt{2}}\|1\rangle$	$-\boxed{H}-$
S	$\begin{bmatrix}1 & 0\\0 & i\end{bmatrix}$	$S\|0\rangle = \|0\rangle$ $S\|1\rangle = i\|1\rangle$	$-\boxed{S}-$
$S^\dagger$	$\begin{bmatrix}1 & 0\\0 & -i\end{bmatrix}$	$S^\dagger\|0\rangle = \|0\rangle$ $S^\dagger\|1\rangle = -i\|1\rangle$	$-\boxed{S^\dagger}-$
T	$\begin{bmatrix}1 & 0\\0 & e^{i\pi/4}\end{bmatrix}$	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = \frac{1+i}{\sqrt{2}}\|1\rangle$	$-\boxed{T}-$
$T^\dagger$	$\begin{bmatrix}1 & 0\\0 & e^{-i\pi/4}\end{bmatrix}$	$T^\dagger\|0\rangle = \|0\rangle$ $T^\dagger\|1\rangle = \frac{1-i}{\sqrt{2}}\|1\rangle$	$-\boxed{T^\dagger}-$
Phase	$\begin{bmatrix}1 & 0\\0 & e^{i\lambda}\end{bmatrix}$	$P\|0\rangle = \|0\rangle$ $P\|1\rangle = e^{i\lambda}\|1\rangle$	$-\boxed{e^{i\lambda}}-$
RX	$\begin{bmatrix}\cos\frac{\theta}{2} & -i\sin\frac{\theta}{2}\\-i\sin\frac{\theta}{2} & \cos\frac{\theta}{2}\end{bmatrix}$	$RX\|0\rangle = \cos\frac{\theta}{2}\|0\rangle \; -i\sin\frac{\theta}{2}\|1\rangle$ $RX\|1\rangle = -i\sin\frac{\theta}{2}\|0\rangle \; +\cos\frac{\theta}{2}\|1\rangle$	$-\boxed{RX_\theta}-$
RY	$\begin{bmatrix}\cos\frac{\theta}{2} & -\sin\frac{\theta}{2}\\\sin\frac{\theta}{2} & \cos\frac{\theta}{2}\end{bmatrix}$	$RY\|0\rangle = \cos\frac{\theta}{2}\|0\rangle \; -i\sin\frac{\theta}{2}\|1\rangle$ $RY\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle \; +\cos\frac{\theta}{2}\|1\rangle$	$-\boxed{RY_\theta}-$
RZ	$\begin{bmatrix}e^{-i\theta/2} & 0\\0 & e^{i\theta/2}\end{bmatrix}$	$RZ\|0\rangle = e^{-i\theta/2}\|0\rangle$ $RZ\|1\rangle = e^{i\theta/2}\|1\rangle$	$-\boxed{RX_\theta}-$
CNOT	$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$	$\text{CNOT}\|a\,b\rangle = \|a\,(a \oplus b)\rangle$	
SWAP	$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix}$	$\text{SWAP}\|a\,b\rangle = \|b\,a\rangle$	

that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result m occurs is given by

$$p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle\,, \tag{12}$$

and the state of the system after the measurement is

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi| M_m^\dagger M_m |\psi\rangle}}. \tag{13}$$

The measurement operators satisfy the *completeness equation*,

$$\sum_m M_m^\dagger M_m = I. \tag{14}$$

(NIELSEN; CHUANG, 2010a, p. 84)

The third postulate refers to the probability and consequences of measurement. Note that different from a quantum gate, a quantum measurement is not a unitary operation. It lost information in the process, so it is not time-reversible.

The measurement operators of the one-qubit computational base (Pauli Z) are

$$M_0 = |0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \text{ and } M_1 = |1\rangle\langle 1| = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}. \tag{15}$$

Replacing M_m with M_0 and M_1 on Equation (12) for $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ we see that $p(0) = |\alpha|^2$, and $p(1) = |\beta|^2$, respectively. And, in Equation (13), if we measure zero, the state will be $\frac{\alpha|0\rangle}{|\alpha|}$, and if we measure one, the state will be $\frac{\beta|1\rangle}{|\beta|}$.

2.2.4 Composed System

Postulate 4: The state space of a composite physical system is the *tensor product* of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$. (NIELSEN; CHUANG, 2010a, p. 92)

The last postulate extrapolates the others for multiples qubits. It relies on the tensor product to concatenate qubits and operations. Equation (16) generalize this operation for matrixes, where A is $m \times n$ and B is $p \times q$.

$$A \otimes B = \begin{bmatrix} A_{11}B & A_{12}B \cdots\cdots & A_{1n}B \\ A_{11}B & A_{12}B \cdots\cdots & A_{1n}B \\ A_{21}B & A_{22}B \cdots\cdots & A_{2n}B \\ \vdots & \vdots \ddots & \vdots \\ A_{m1}B & A_{m2}B \cdots\cdots & A_{mn}B \end{bmatrix}_{mp \times nq} \tag{16}$$

Now, we can rewrite Equation (4) as matrix multiplication. With the initial state

$$|xy\rangle = |00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{17}$$

we need to concatenate the Hadamard gate with the Identity

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{18}$$

to take the matrix to a compatible dimension and operate only with the qubit $|x\rangle$

$$H_x |00\rangle = (H \otimes I) |00\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{|00\rangle + |10\rangle}{\sqrt{2}}. \qquad (19)$$

The measurement operators for the two-qubits computational base are $\{M_a \otimes M_b \mid M_a, M_b \in \{|0\rangle\langle 0|, |1\rangle\langle 1|\}\}$, and we can generalize for n-qubits as $\{|k\rangle\langle k| \mid 0 \leq k < 2^n\}$.

3 QUANTUM PROGRAMMING

Quantum programming is a relatively new paradigm that has particularities that do not exist in classical programming. We start this chapter by discussing those quantum programming features and limitations (Section 3.1). For our related works, we present an overview of quantum programming languages (Section 3.2), classifying than in Quantum Assembly Languages, Quantum Circuit Circuit Description Languages, and Classical-Quantum Programming Languages (Section 3.3).

3.1 PROGRAMMING PARTICULARITIES

Quantum computers have features and constraints that are not present in classical computers, therefore, not addressed by classical programming languages. In this section, we introduce some aspects of quantum computation, highlighting its impact on quantum programming. We divide those characteristics into Features, Limitations, and Construction Constraints. In Features (Subsection 3.1.1), we present quantum computations advantages that can speed up computation, and in Limitations (Subsection 3.1.2), we introduce restrictions that get in the way of quantum programming. For Construction Constraints (Subsection 3.1.3), we discuss quantum programming limitations that are not intrinsic of quantum computation but reflections from the NISQ computers.

3.1.1 Features

Superposition and **entanglement** are the quantum mechanics phenomena that give quantum computers their power (NIELSEN; CHUANG, 2010a). As these features are not available in classical computers, and to simulate them takes exponential time and space, classical programming languages had no reason to address superposition and entanglement. Only quantum computers have turned those characteristics relevant. As quantum programming is a relatively new paradigm, we do not have a standard or consolidated method to create and represent superposition and entanglement.

Non-determinism and parallelism are not good analogies for superposition, despite commonly used. A quantum register has the potential of having more than one associated value. For instance, while a 32-bits integer stores a single number, a 32-quantum-bits integer stores up to 2^{32} numbers. With superposition, we have exponential storage capacity, where n-quantum-bits equals 2^n-bits of information. Also, the process that creates superposition is deterministic and not understood as branching in computation.

Unlike the branches of a non-deterministic execution, states in superposition interfere with each other. And for parallelism, even if we had enough memory to store quantum bits equivalents, the parallel execution is limited by the number of threads.

Also, in parallel execution, we need to coordinate memory access to avoid a race condition, which is not the case for superposition since it is a single execution branch.

As both superposition and entanglement are simultaneously necessary for a quantum computer to express its power, it is hard to unlink these two features. A usual approach for quantum programming languages to store quantum information is to use the standard quantum bit, or qubit for short. Indeed, all programming languages for digital quantum computation of our related works (Section 3.2) use the qubit representation. We discuss how to manipulate the quantum state to generate superposition and entanglement in the next subsection.

Quantum computation is **reversible in time**, meaning that we can undo any quantum computation before a measurement. Reversibility is an intrinsic characteristic of quantum computation used by several quantum algorithms. We continue to discuss reversibility in the next subsection.

3.1.2 Limitations

As we mentioned before, a quantum state can store an exponential amount of information. However, we cannot know the state in polynomial time. Therefore, in quantum computation, **measurement** is the only viable way to gather classical information from a quantum superposition. A measurement returns one base state of a superposition at random, but in the process, it collapses the superposition to the measured state. For example, the measurement of qubit $|0\rangle + |1\rangle$ can return either 0 or 1, collapsing the qubit to $|0\rangle$ or $|1\rangle$, respectively.

Measurement is the only operation that simultaneously changes the quantum state and returns classical information. Also, it is the only non-reversible quantum operation. In quantum programming languages, measurement is an explicit statement, usually in the **computational base**.

A quantum state is equivalent to a unit vector, and except for measurement, quantum operations are **unitary transformations**, meaning that there is no loss of information during computation. This characteristic makes reversibility an intrinsic feature of quantum computation. However, since reversibility is not a concern for classical programming languages, they do not provide ways to prevent irreversible constructions. However, quantum programming languages must provide means to restrict or warn the use of irreversible quantum operations.

A possible strategy prevent irreversible quantum operations is to only use **quantum gates** to manipulate the quantum state. As quantum gates are reversible, any combination of gates is also reversible. The downside of quantum gates is that they are not so intuitive as the usual classical operations. For example, the sum of two qubits-registers requires a polynomial number of gates instead of a single sum operation. On the other hand, as most quantum computers use the quantum gate model of computation, compiling from a quantum programming language build on quantum gates is straightforward. Also, high-level quantum programming languages provide ways to wrap quantum gates in subroutines to implement more complex operations.

The use of quantum gates also prevents the **no-cloning theorem** (WOOTTERS; ZUREK, 1982) violation, but that is not enough in itself. We cannot copy a quantum state from a qubit to another, a usual and necessary classical programming operation. Most quantum programming languages use linear-type system (BICHSEL et al., 2020; PAYKIN et al., 2017; GREEN et al., 2013) or opaque reference (SVORE, K. et al., 2018; CROSS et al., 2017) to avoid quantum state copies.

In a **linear type system**, we can use a variable once only. This way, we cannot pass a variable that stores a quantum state to more than one function/gate or assign it to another variable and keep using the original. Quantum programming languages can enforce the use of linear type only on quantum variables (BICHSEL et al., 2020; SINGH et al., 2017).

Qubit data types can store an **opaque reference** instead of a quantum state, so any explicit or implicit copy operation only copies the reference. This strategy makes it easy to translate from a high-level quantum programming language to a low-level one since most quantum assembly languages use opaque qubits reference. Considering that only quantum gates can change the quantum state, any other operation on a qubit reference has no quantum side effect.

3.1.3 Construction Constraints

The main limitations of NISQ computers are the **number of qubits**, **quantum gate fidelity**, and **decoherence time** (PRESKILL, 2018). Although those metrics have increased significantly in recent years, we do not expect to get rid of those implementation constraints soon.

The number of qubits limits the input size of a quantum application. For example, in Shor's factorization algorithm (GIDNEY; EKERÅ, 2021), the larger the number you want to factorize, the more qubits you need. On the other hand, the quantum gate fidelity limits the complexity of a quantum application. Since every operation introduces a small error to the quantum state, it can accumulate, invalidating the quantum computation. Another factor that limits a quantum application complexity is the decoherence time (DEVITT et al., 2013), which tells how long a quantum computer can hold a quantum state coherently, independently of the computation.

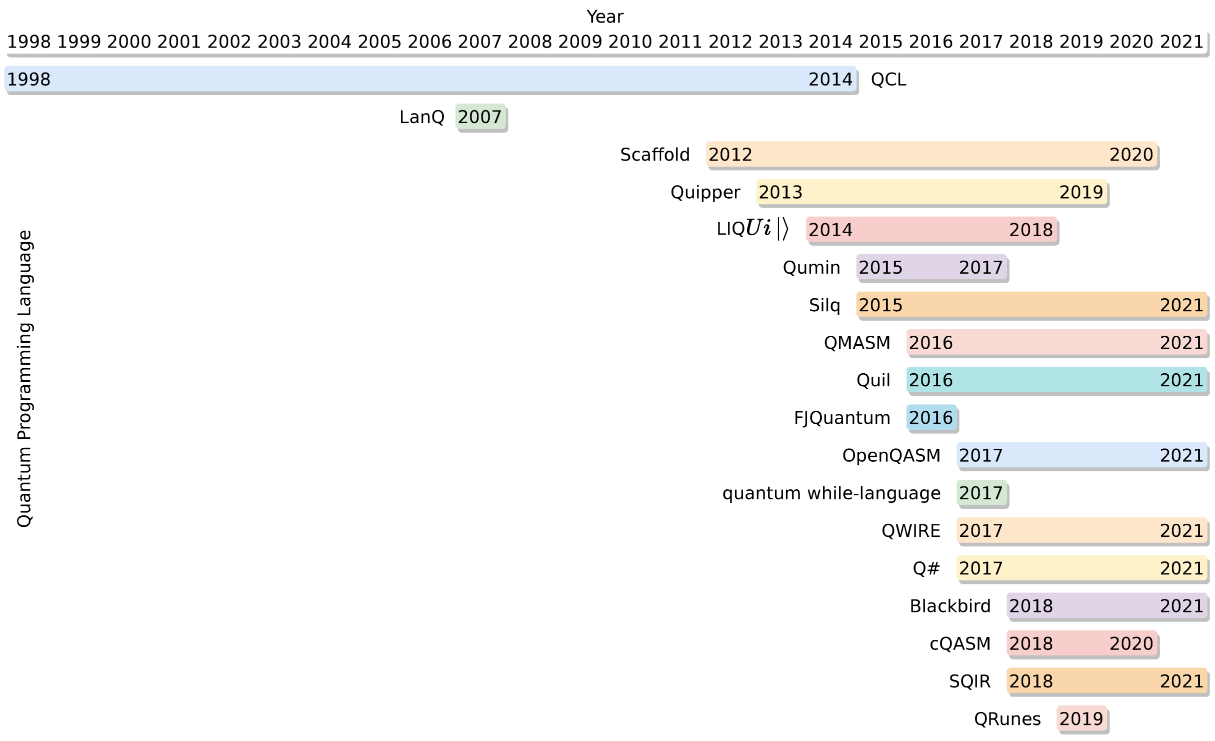
The execution of **classical instruction** and **classical control flow** on today's quantum computers are limited, although improving (FU et al., 2019). Considering that a quantum computer is a coprocessor or a cloud service, quantum programming languages need to separate and handle classical instructions and information on quantum computers.

When the decoherence time of a quantum computer is shorter than the latency between it and a classical computer, interactive quantum execution is not an option. For example, if a quantum computer in the cloud waits for instructions from a classical computer, at the time it receives, the quantum information may be lost to decoherence. That is why NISQ computers are **batch processors**. A quantum computer processing in batch imposes limitations for hybrid classical-quantum programming.

3.2 RELATED WORKS: QUANTUM PROGRAMMING LANGUAGES

Quantum programming is a relatively new paradigm that growth with the development of NISQ computers in the last ten years. Quantum Computation Language (QCL) by Ömmer is the first quantum programming language implementation dating back to 1998. However, it is the Scaffold and Quipper languages that started the quantum programming language boom in 2012. We survey 19 quantum programming languages in this work, with 17 of them developed after 2012. We summarized the development of quantum programming languages through the years in Figure 3, where we list the year of the first and last release. We consider the first release as the first public release, article publication, or git repository creation, and as the last release, we recognized the latest public release or git commit.

Figure 3 – Quantum Programming Languages through the yeas.



We survey only languages with a reference implementation that we can use in some quantum programming process, been quantum hardware execution, simulation, resource estimation, or validation. In Section 3.3, we classify quantum programming languages in Quantum Assembly Language, Quantum Circuit Circuit Description Language, and Classical-Quantum Programming Language. In the remainder of this section, we present an overview of each known quantum programming language.

Blackbird (KILLORAN et al., 2019) is a quantum assembly language for continuous variable (CV) quantum computation that targets Xanadu's photonic quantum information processors and the Strawberry Fields simulator. The Strawberry Fields provides a Python-embedded implementation of Blackbird, which uses the same syntax as the standalone language with the extension of Python constructions. The CV quantum computation model uses a different set of quantum gates that act on qumodes, CV equivalent for qubits. The computational power of the Quantum Gate and CV models is equivalent. Blackbird is the only language in the survey that targets the CV model.

FJQuantum (FEITOSA et al., 2016) is an extension of the Featherweight Java language that handles quantum data through a monadic approach. Featherweight Java is a subset of Java, focusing on a functional perspective. The language does not provide quantum gate primitives. Instead, it provides features like conditional control, monadic sum, and scalar product that we can use to implement quantum gates. Although it is possible to implement quantum gates in FJQuantum, we cannot classify the language in the Quantum Gate model of computation. FJQuantum has an interpreter with a built-in quantum simulator available.

LIQ*Ui* |⟩ (WECKER; SVORE, K. M., 2014) is an F#-embedded quantum programming language developed by the Quantum Architectures and Computation team of Microsoft Research. The programming language is hardware-independent, but it supports architecture-specific timing and layout constraints. It also provides tools to study quantum noise and quantum error correction code, simulating thousands of qubits with a stabilizer simulator. In addition to simulation, LIQ*Ui* |⟩ also supports quantum circuit rendering.

LanQ (MLNARIK, 2007) is a high-level quantum programming language with C-like syntax, which implements classical control and quantum data stored in qubits. It also provides means for parallel process execution and communication, with channels sharing classical and quantum information in shared memory. The language uses the Quantum Gate model, and its reference implementation provides a quantum computer simulator.

OpenQASM (CROSS et al., 2017) is one of the most used quantum assembly languages. Specified by the IBM Quantum Computing group, the language is used in the IBM Quantum Experience and Qiskit platforms (ANIS et al., 2021) and as an intermediary representation for some quantum applications (JAVADIABHARI et al., 2014; KISSINGER; WETERING, 2020; BERGHOLM et al., 2020). On its current version, OpenQASM 2.0 is equivalent to the Quantum Circuit model of computation, featuring classical and quantum bit registers, a two-qubits gate (CNOT), the single-qubit rotation U gate,

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta) & e^{-i\lambda}\sin(\theta) \\ e^{i\phi}\sin(\theta) & e^{i(\phi+\lambda)}\cos(\theta) \end{bmatrix} \tag{20}$$

Pauli Z measurement, and `if-then` statement. The language's next iteration, OpenQASM 3.0, which is under development, plans to add classical control flow, instructions, and types to enable the interactive quantum execution.

QCL (ÖMER, 2005) is the first implementation of a quantum programming language developed to fill the gap between quantum mathematical formalism and the classical programming constructions well-known by computer scientists. Build on top of the Quantum Gate model, the programming language has a C-like syntax, and its implementation features a quantum simulator.

QMASM (PAKIN, 2016) is a quantum assembly language for quantum annealing that provides low-level instructions and hardware-independent abstraction. It is suitable to be used as a programming language itself or as a compilation target for higher-level programming language. The QMASM implementation uses the D-Wave Ocean (D-WAVE SYSTEMS INC., 2021), which targets D-Wave's hardware and simulator. Quantum annealing is a weaker model of computation than Quantum Gate, focused on solving optimization problems. QMASM is the only language targeting quantum annealers in this survey. The assembly language was first called QASM and later renamed to avoid ambiguity.

QRunes (CHEN; GUO, 2019) is a high-level imperative quantum programming language designed to be transpiled to C++ and Python, using, respectively, the libraries QPanda and pyQPanda. Those libraries feature CPU, GPU, and cloud-based quantum simulators of up to 32 qubits.

QWIRE (PAYKIN et al., 2017) is a language for quantum circuit descriptions, which uses a linear type system to ensures the non-violation of the no-cloning theorem. Intended to be an embedded programming language, but without a specific host language, QWIRE has an implementation embedded in the Coq proof assistant that compiles for OpenQASM.

Q# (SVORE, K. et al., 2018) is a domain-specific quantum programming language part of the Microsoft Quantum Development Kit (QDK). It features statements similar to C# and F#, asserts for quantum debugging, and specific quantum statements, like the repeat-until-success comparable to the C do-while statement with a fixup code that runs after an unsuccessful loop. The language has a vast standard library with methods for quantum chemistry, machine learning, quantum error correction, and more. The language implementation features a full state simulation similar to LIQ$Ui|\rangle$ and a limited Toffoli simulator for millions of qubits that only operate with Not and multi-controlled Not gates. With a quantum resource estimator, Q# can also provide metrics such as the number of qubits and quantum gates need for a given quantum program. Q# programs can run standalone, executing in command line or Jupyter Notebook, or integrated with Python, C#, or F#.

Quantum while-language (LIU et al., 2017) is a quantum programming language embedded in C#. It is part of the Q$|$SI$\rangle$ quantum programming environment that features quantum simulation, optimization, analyzing, and verification. The programming language extends C# with qubit datatype, quantum gates, quantum measurement, and if and while statements based on measurement results.

Quil (SMITH et al., 2017) is an instruction set or assembly language for hybrid classical-quantum computers. The language divides into classical and quantum states, with classical bits and a program counter (PC) forming the classical state and qubits composing the quantum state. Jump instructions can change the PC to implement control flow. Quantum gates and measurement are the only instructions that can change the quantum state, with measurement being the only operation that affects both classical and quantum states. Although Quil uses the same gate-based model of computation as cQASM and OpenQASM, the ability to implement control flow makes Quil more expressive, meaning that it can describe a more comprehensive set of algorithms and methods. The Forest SDK provides means to generate Quil code with the pyQuil, and execute then on Rigetti's quantum computers and simulators. However, Rigetti's computers support only a subset of Quil that does not allow control flow.

Quipper (GREEN et al., 2013) is a Haskell-embedded quantum programming language that allows the scalable description of large and parameterized quantum circuits. Its primary objective is to estimate and reduce resources for quantum algorithm execution on quantum hardware. Therefore the language provides a comprehensive quantum computation library and robust operators for quantum circuit assembling. It also implements a quantum circuit simulator and rendering.

Qumin (SINGH et al., 2017) is a minimalist quantum programming language composed of two sub-languages defined in lambda calculus. An untyped language handles the classical operation and controls the quantum operations, while a language with a linear type system manipulates the qubits applying quantum gates and measurement. This construction imposes restrictions to enforce the no-cloning theorem (WOOTTERS; ZUREK, 1982) only on the quantum part of the programming language, without affecting the classical one. An experimental implementation of an interpreter is available for the Qumin.

SQIR (HIETALA et al., 2021) is a quantum intermediary representation language embedded in Coq for the VOQC verified optimizer. VOQC can convert SQIR to OpenQASM and vice versa, allowing the integration with several quantum programming software.

Scaffold (JAVADIABHARI et al., 2014) is a C-like quantum programming language for static quantum circuit description. The language implements the novel feature Classical-To-Quantum-Gate that creates quantum circuits from classical ones generated from the language's statements and expressions. As a Scaffold program statically describes quantum circuits, the Scaffold compiler (ScaffCC) performs loop-unrolling during the compilation process to implement the 'for' statement. ScaffCC compiles to OpenQASM, among other compilations targets.

Silq (BICHSEL et al., 2020) is a high-level quantum programming language that uses inverse quantum computation to free temporary variables. Releasing temporary qubits without removing their entanglement with the rest of the system has unwanted side effects. Silq performs uncomputation to untangle and release qubits, allowing it to have a syntax closer to classical programming, mitigating the reverse quantum computation problem. The Silq's interpreter features a built-in quantum simulator.

cQASM (KHAMMASSI et al., 2018) is a technology-independent quantum assembly language for quantum circuit description. The language objective is to present a common quantum assembly language to interface with different tools, unifying the diverse QASM dialects. As DQCsim frontend, the cQASM can target the Quantumsim and QX simulators. Overall, the assembly language has classical and quantum bits registers, several gates of one, two, and three qubits, measurement in Pauli Z, X, and Y, control statement, and static loops.

3.3 CLASSIFICATION

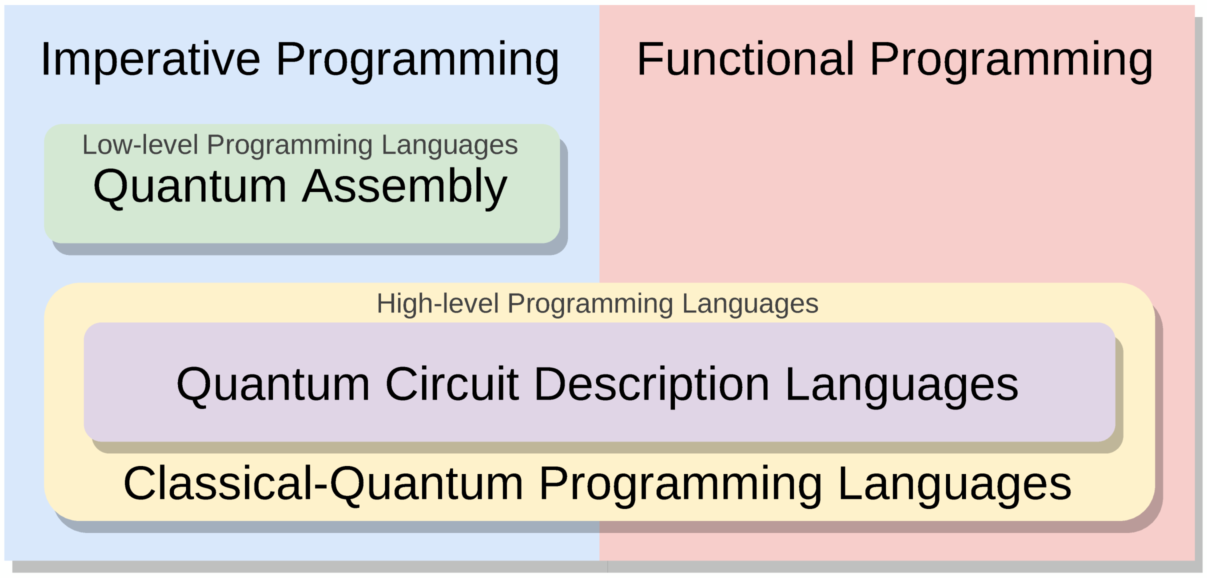
We divide the quantum programming languages (QPL) into three groups based on their abstraction and expressivity power. The low-level QPLs, Quantum Assemble languages; and the high-level QPLs, Quantum Circuit Description Languages and Classical-Quantum Programming languages. Also, we can divide the high-level QPLs into functional and imperative based on their programming paradigm. Figure 4 illustrates this division, and Table 4 summarizes the classification. We can also classify Quantum Programming Languages into Functional and Imperative as in the previous surveys (GAY, 2006; GARHWAL et al., 2021; SELINGER, 2004; SOFGE, 2008a).

In the next subsections, we discuss the classification of QPLs in Quantum Assembly languages, Quantum Circuit Description Languages, and Classical-Quantum Programming Languages, emphasizing the particularities of each group and its main QPLs.

Table 4 – Quantum Programming Languages

Programming Language	Programming Paradigm
Quantum Assembly Language	
Blackbird	Imperative
cQASM	Imperative
OpenQASM	Imperative
QMASM	Imperative
Quil	Imperative
Quantum Circuit Description Language	
LIQ$Ui\,|\rangle$	Functional
Quipper	Functional
QWIRE	Functional
Scaffold	Imperative
SQIR	Functional
Classical-Quantum Programming Language	
FJQuantum	Obj. Oriented/ Functional
LanQ	Imperative
QCL	Imperative
QRunes	Imperative
Q#	Imperative
Qumin	Functional
Silq	Imperative
quantum while-language	Imperative

Figure 4 – Quantum programming languages classification.


Imperative Programming
Functional Programming
Low-level Programming Languages
Quantum Assembly
High-level Programming Languages
Quantum Circuit Description Languages
Classical-Quantum Programming Languages

3.3.1 Quantum Assembly Languages

Quantum Assembly (QASM) languages describe simple quantum and classical instructions for quantum computers. Those instructions have a straightforward translation to quantum computer's control signals and pulses. Higher-level quantum programming languages often use a QASM as a compilation target, and quantum programming software often uses them as an intermediary representation (JAVADIABHARI et al., 2014; KISSINGER; WETERING, 2020; ANIS et al., 2021; BERGHOLM et al., 2020).

Today, the only languages officially supported by quantum computer vendors for quantum hardware execution are QASMs. Microsoft plans to run Q# (SVORE, K. et al., 2018) in its quantum computers, but it is not yet available. Table 5 shows the QASMs and programming platforms used to program quantum hardware. The only examples of quantum programming languages that use a quantum computation model other than quantum gates are the QASMs Blackbird (KILLORAN et al., 2019) and QMASM (PAKIN, 2016). With Blackbird being used to program Xanadu photonic quantum computers in the continuous-variable quantum computation model (KILLORAN et al., 2019); and QMASM to program D-Wave adiabatic quantum computers used to perform Quantum Annealing (PAKIN, 2016).

Table 5 – QASM languages for quantum hardware execution.

QASM Language	Programming Platform	QC Vendor
Blackbird	Strawberry Fields	Xanadu
OpenQASM	Qiskit	IBM
QMASM[1]	Ocean	D-Wave
Quil	Forest SDK	Rigetti

For quantum hardware execution using the quantum gate model of quantum computation, the available QASM are OpenQASM (CROSS et al., 2017) and Quil (SMITH et al., 2017), used to program the IBM and Rigetti quantum computers, respectively. An OpenQASM program describes a quantum circuit with quantum gates and if statements. Similarly, Quil manipulates the quantum state with quantum gates, but for classical control, it uses an approach more similar to the classical assembly. Quil allows better control of the program counter than OpenQASM by implementing labels and jump instructions. This characteristic makes Quil a more expressive language than OpenQASM, allowing it to implement, among other things, the semantics of the statements `if-then-else`, `for`, and `while`. However, today, neither IBM nor Rigetti quantum computers support interactive quantum execution. The OpenQASM v3, which is under development, plans to add classical control flow and instructions.

Naming a quantum assembly language of QASM is commonplace. For example, QMASM (PAKIN, 2016) was initially published as QASM and later renamed to avoid conflict with MIT's QASM[2] and IBM's QASM (OpenQASM). MIT's QASM was the first QASM that inspired other similar languages like OpenQASM and cQASM. Released in 2005, MIT's QASM is the language supported by qasm-tools, an open-source software package for studying fault-tolerance quantum circuits (SOFGE, 2008b; ALIFERIS et al., 2005). The cQASM (KHAMMASSI et al., 2018) language was born with the initiative to create a common quantum assembly language, opposing the diverse MIT's QASM dialects.

3.3.2 Quantum Circuit Description Languages

Quantum Circuit Description Languages (QCDL) are higher-level quantum programming languages with semantics limited to the quantum circuit model of quantum computation (NIELSEN; CHUANG, 2010b). This limitation impacts the classical control of a quantum execution, shortening how dynamic and generic a quantum program can be. For example, a quantum program in a QCDL cannot dynamically execute `for` statements. However, as implemented in ScaffCC (JAVADIABHARI et al., 2014) (the Scaffold optimizing compiler), this does not prohibit the compiler from performing a loop-unrolling during the translation process. It is worth mentioning that the quantum circuit model of computation is Turing-complete (MOLINA; WATROUS, 2019), meaning that it can describe any computable problem.

Scaffold (JAVADIABHARI et al., 2014) and Quipper (GREEN et al., 2013) mark the resurgence of quantum programming languages. Both of them focus on quantum circuit synthesis but from different perspectives. The Scaffold language is a C-like language that can compile classical operation into a quantum circuit; Quipper is a

[1] QMASM uses Ocean SDK, although it is not part of the D-Wave's plataform.
[2] `https://www.media.mit.edu/quanta/quanta-web/projects/qasm-tools/`

Haskell-embedded language for the scalable description of potentially long quantum circuits. Scaffold and Quipper, respectively released in 2012 and 2013, are the first relevant quantum programming language after the pioneer QCL language (ÖMER, 2005), published in 1998 (ÖMER, 1998).

Quantum circuit is a widespread quantum computing model used in most quantum computers, and therefore the more accessible paradigm for quantum execution. QCDLs can also benefit from the vast literature for quantum circuit compilation and optimization (SIRAICHI et al., 2018; KISSINGER; WETERING, 2020; BHATTACHARJEE et al., 2019).

Since quantum programs written in a QCDL have the size and classical values fixed, they can be strongly optimized and their execution highly parallelable (JAVADIABHARI et al., 2014). Also, there is no need for runtime optimizations, such as branching prediction and runtime qubit allocation (JAVADIABHARI et al., 2014). However, the limited expressiveness of QCDLs also limits classical-quantum interactions.

3.3.3 Classical-Quantum Programming Languages

Classical-Quantum Programming Languages (CQPL) are high-level programming languages that allow any combination of classical control and quantum operations, including dynamic loops controlled by quantum measurements. Examples of quantum constructions enabled by control flow are: qubits reuse through runtime qubit allocation and deallocation (RISTÈ et al., 2012); implementation of quantum communication processes like quantum teleportation (BENNETT et al., 1993); use of interactive quantum applications like non-deterministic quantum gate decomposition (BOCHAROV et al., 2015) and iterative quantum phase estimation (KITAEV, 1995).

The expressivity gain of CQPLs over QCDLs makes it easier to integrate classical and quantum algorithms, which can boost the development of hybrid classical-quantum applications. However, it comes at the cost of optimization. Classical control flow splits a quantum program into basic blocks, making it difficult to optimize a quantum application as a whole like a quantum circuit. For in-block optimization, we can use the same optimization techniques as quantum circuit optimization.

For languages like Q# (SVORE, K. et al., 2018) that allow dynamic allocation of qubits, mapping a logical qubit to a physical one at runtime is another challenge. Quantum computer realization, such as superconductor qubits, organize qubits in a topology where qubits do not directly communicate with all other qubits, which makes mapping of logical qubits to physical ones a not straightforward process.

Today's quantum computer has limited support to control flow which limits the full implementation of CQPLs. Also, many CQPLs consider that a quantum computer is a coprocessor with real-time communication, but it is not the case for NISQ computers that are cloud-based batch processors.

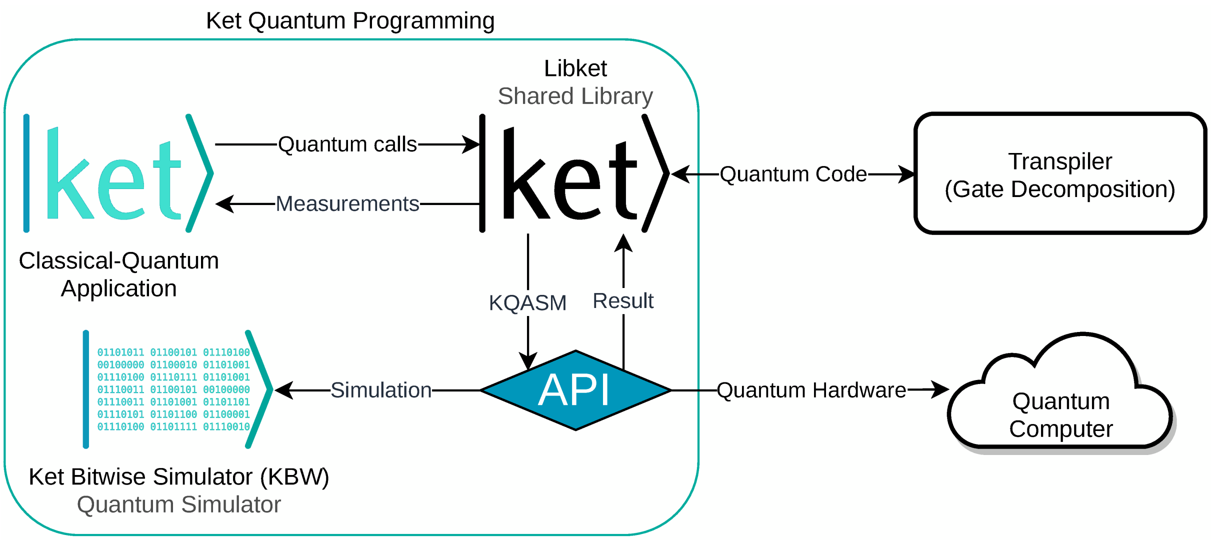
The language Q# introduces the quantum-specific construction repeat-until-success (BOCHAROV et al., 2015). This loop statement has a body scope that executes every iteration, a stop condition, and a fixup code that runs whenever the stop test fails. We cannot implement this construction in QCDLs. The within-apply is another Q# quantum-specific construction. This statement is composed of a `within` and a `apply` scopes, which execute as follows: `within` $\rightarrow$ `apply` $\rightarrow$ inverse `within`.

FJQuantum (FEITOSA et al., 2016) is a quantum programming language that escapes from the quantum gate model. Instead, it uses conditional construction to implement quantum operations. However, FJQuantum makes it hard to compile a code to a gate-base quantum computer. The only other quantum programming language that uses quantum operations other than quantum gates is the Silq (BICHSEL et al., 2020) language, which uses constructions that implicitly create auxiliary qubits and automatically use inverse quantum computation to release them. However, Silq uses quantum gates to define this behavior, making it easy to compile the language to a gate-based quantum computer.

4 PROPOSED RUNTIME ARCHITECTURE

Ket is a Classical-Quantum Programming Language designed for the scenario presented in Section 1.3. Using a single source code to program both the classical and cloud-based quantum computers, Ket mitigates the interaction limitation between the two computers and enables quantum applications with dynamic execution while keeping the quantum code as specific as possible. In this chapter, before presenting further details of Ket in Chapter 5, we introduce its runtime architecture, summarized in Figure 5, responsible for enabling the core features of Ket.

Figure 5 – Ket's runtime architecture



The runtime architecture is composed of several componentes with the following workflow. A classical-quantum application calls the shared library, implemented in Libket (Section 4.1), which handles the quantum operations. Then Libket generates Ket QASM code at runtime and sends it for quantum execution, performed by the Ket Bitwise Simulator (Section 4.3), in this case. With the results of the quantum execution, the shared library returns to the classical-quantum application. Despite not yet implemented, before the quantum execution, optionally, the shared library can send the quantum code to a transpiler for quantum gate decomposition (Section 4.2), which is necessary for quantum hardware execution (Section 4.4). The following subsections detail every component of the runtime architecture as well as their interactions and intermediaries.

4.1 LIBKET: SHARED LIBRARY

The shared library is an intermediary for other components of the runtime architecture, and its implementation, Libket, is the core of the Ket Quantum Programming framework. The library handles all quantum operations of the classical-quantum application, including qubit allocation, quantum gate application, including inverted and controlled, measurement, and quantum computer's control flow. Libket is a C++ library that can be used standalone to implement classical-quantum applications and tools. For instance, we implement a Python wrapper of Libket for the Ket interpreter. So the Ket Program can be either written in C++ and linked with Libket or written directly in the Ket language.

Libket uses the calls performed by the classical-quantum application to generate a quantum code in the Ket Quantum Assembly (KQASM) language, which is an intermediary representation, which is not necessarily equivalent to a quantum circuit. As the quantum code is independent of the quantum execution target, KQASM has no preset limit of qubits and connectivity layout, acting on logical qubits. We designed KQASM to take advantage of Ket Bitwise Simulator (Section 4.3), supporting adding an arbitrary number of control qubits to any single-qubit quantum gate and using labels and branch instructions, similar to Quil, to implement control flow.

KQASM has a basic block structure similar to the LLVM IR (LLVM PROJECT, 2021), where every block starts with a label and ends with a jump or branch instruction, except the last block that ends in the end-of-file. We present the context-free grammar of KQASM in Code 4.1.1. We can see that KQASM, and so Libket, has native support for all single-qubits gates of Table 3, including their controlled version.

Libket has three main classes, `process`, `quant`, and `future`. The `process` class handles every quantum operation and the interaction with others components of the runtime architecture. However, we did not plan for a programmer to create and access directly the `process` variables. Instead, it must be handled by other classes and functions. The `quant` class handles qubit allocation and deallocation, storing and managing a list of qubit opaque references. The `future` class stores a reference for a classical 64-bits signed integer located in the quantum computer. A quantum computer can generate a classical integer in three ways: quantum measurement, constant value in the quantum code, or classical binary operation. The integer value of a `future` variable is only available for the classical computer after the quantum execution.

Libket can produce multiple unrelated quantum codes concurrently. This feature can help quantum programming dynamism and quantum execution performance. For example, considering that we want to use an actual random number generated by a quantum computer as input of a quantum algorithm, we can initialize a second quantum code independent of the main one to produce the random value. This approach reduces the number of qubits and (consequently) gates required to run the quantum application by splitting it into several quantum executions. To implement this behavior, Libket uses a global stack of `process` variables. Every Libket class and function always communicates with the top `process` variable. And unlike a programming language scope, a `process` variable does not have access to the `process` variable below in the stack. To stack

Code 4.1.1 – Ket Quantum Assembly language ANTLR grammar.

```
grammar kqasm;

start       : block* end_block;
block       : label (instruction ENDL+)* end_inst ENDL+;
end_block   : label (instruction ENDL+)*;
label       : 'LABEL' LABEL ENDL+;
instruction : ctrl? gate_name arg_list? QBIT                # gate
            | ctrl? 'PLUGIN' '!'? name=STR qubits_list ARGS # plugin
            | 'ALLOC' 'DIRTY'? QBIT                         # alloc
            | 'FREE' 'DIRTY'? QBIT                          # free
            | 'INT' result=INT left=INT bin_op right=INT    # bin
            | 'INT' INT '-'? UINT                           # const
            | 'SET' target=INT from=INT                     # set
            | 'MEASURE' INT qubits_list                     # measure
            | 'DUMP' qubits_list                            # dump
            ;
end_inst    : 'BR' INT then=LABEL otherwise=LABEL           # branch
            | 'JUMP' LABEL                                  # jump
            ;
ctrl        : 'CTRL' qubits_list ',';
qubits_list : '[' QBIT (',' QBIT)* ']';
gate_name   : 'X'|'Y'|'Z' |'H'|'S'|'SD'|'T'|'TD'|'P'|'RZ'|'RX'|'RY';
arg_list    : '(' DOUBLE (',' DOUBLE)* ')';
bin_op      : '=='|'!='|'>' |'>='|'<'  |'<=' |'+' |'-'
            | '*' |'/' |'<<'|'>>'|'and'|'xor'|'or'
            ;

ADJ    : '!';                        ARGS : '\"'~["]+'\"';
DIRTY  : 'DIRTY';                    UINT : [0-9]+;
QBIT   : 'q'UINT;                    INT  : 'i'UINT;
DOUBLE : '-'?[0-9]+'.'[0-9]*;        ENDL : '\r''\n'?|'\n';
LABEL  : '@'STR;                     SIG  : '-';
STR    : [a-zA-Z]+[._0-9a-zA-Z]*; WS   : [ \t]+ -> skip;
```

up and pop a `process` variable, Libket uses the `process_begin()` and `process_end()` functions, respectively.

The main features of the shared library are the ability to generate the quantum code at runtime and that measurements return a *future* variable that represents a promise for the results. Those two features are essential for Ket's ability to execute generic quantum applications with the dynamic interaction between classical and quantum computers.

4.1.1 Runtime Quantum Code Generation

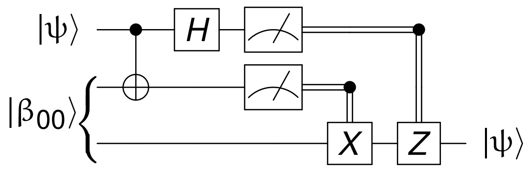
Due to decoherence, every quantum operation needs to be highly optimized. Therefore, it is discouraged to send to a quantum computer a parameterized quantum execution, where the classical computer can define the input. For example, Shor's algorithm needs to perform modular exponentiation on a superposition taking a state $|x\rangle|0\rangle$ into $|x\rangle|a^x \mod N\rangle$, where a and N are classical parameters. We can optimize this operation for every classical parameter if we statically define the values, but this strategy does not allow interactive execution. Libket's approach for this problem is to delay the quantum code generation for the classical runtime where every parameter is available. This way, we can define a generic Ket program using any classical control needed, and the KQASM code will have only the necessary operations.

Besides enabling the description of parameterized quantum applications, as described above, the generation of quantum code at runtime also permits programming responsive quantum applications, which change their execution based on user input, sensor measurement, or quantum execution. For example, a program that returns the prime factors for a user inputted number and a quantum code that uses a measurement result as input.

4.1.2 Delayed Execution

Some quantum applications, *e.g.*, the quantum teleportation of Figure 6, uses measurement results to control the application of quantum gates. Considering that the quantum computer's decoherence time may be shorter than the response time between the classical and the quantum computers, it is up to the quantum computer to handle this control instead of the classical computer. It is in this context that measurement results as a `future` variable takes place.

Figure 6 – The quantum teleportation circuit, where $|\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$. The last two quantum gates Z and X are controlled by the measurement result of the top two qubits.



Inspired by concurrent programming, when a program calls for a measurement, the shared library returns a `future` variable instead of the measurement result. A `future` variable holds a promise for a measurement result that is fulfilled by Libket when the classical computer needs the value. However, this promise accomplishment implies the

Code 4.1.2 – An `if` statement that runs on the quantum computer, implemented in Ket (left) and C++ using Libket (center), and the generated KQASM (right).

Ket
```
q = quant(2)
H(q[0])
if measure(q):
    X(q[1])
```

Libket (C++)
```
quant q{2};
H(q(0));
label then{"then"};
label end{"end"};
auto test = measure(q);
branch(test, then, end);
then.begin();
X(q(1));
jump(end);
end.begin();
```

KQASM
```
LABEL @entry
  ALLOC   q0
  ALLOC   q1
  H       q0
  MEASURE i0  [q0]
  BR  i0  @then0 @end1
LABEL @then0
  X       q1
  JUMP    @end1
LABEL @end1
```

execution of the quantum code. With `future` variables, it is possible to use measurement results in control flow since Libket can place them in the quantum code for the quantum computer to execute. In code 4.1.2, we present an `if` statement that runs in the quantum computer. In the Ket language, we integrate the `future` class with the Python `if` declaration. Using Libket with C++, we can end and begin a basic block with the functions `branch()` and `jump()`, and the method `begin` of the `label` class, respectively. A `future` variable can hold other information that is not necessarily a measurement result but is only available in the quantum computer, *e.g.*, the result of expressions with measurement results and loop control variables.

The quantum code execution is the last action of a `process`. When requested by a `future` variable, Libket sends the generated KQASM file to Ket Bitwise Simulator, which returns the value of the integers, so that the `process` can fulfill the `future` variables.

4.1.3 Inverse and Controlled Operations

As quantum computation is inherently reversible, Libket can generate the adjunct of a quantum operation inverting the order of the quantum gates and change it for its inverse. This functionality is only available for quantum gates. To construct KQASM with the inverse operation, Libket uses a stack of stacks of gates as following. A call to begin an inverse section (`adj_begin()`) pushes an empty stack of gates. The call to end an inverse segment (`adj_end()`, Code 4.1.3) has two possibilities. First, if the outer stack has one element (lines 8-11), the inner stack pops the gates into the KQASM file (line 10); else (lines 12-15), the inner stack pops the gates into the stack below (line 14). Inside of an inverse section, the library places the quantum gates into the top stack. If the outer stack has an odd number of elements, the library pushes the adjunct quantum gate instead.

Libket allows adding qubits of control to the application of any quantum gate. For example, a CNOT gate, with qubit `q0` as control and `q1` as the target, in KQASM is “`CTRL`

Code 4.1.3 – Libket implementation of `adj_end`.

```cpp
// Save the inner stack in a queue
std::queue<std::string> tmp;
while (not adj_stack.top().empty()) {
    tmp.push(adj_stack.top().top());
    adj_stack.top().pop();
}
adj_stack.pop(); // Pop the inner stack
if (adj_stack.empty()) while (not tmp.empty()) {
    // If the outer stack is empty
    kqasm << tmp.front(); // Push into the KQASM file
    tmp.pop();
} else while (not tmp.empty()) {
    // If the outer stack is not empty
    adj_stack.top().push(tmp.front()); // Push into the stack below
    tmp.pop();
}
```

`[q0], X q1`”, and a Toffoli gate, with qubits `q0` and `q1` as control and `q2` as the target, is “`CTRL [q0, q1], X q2`”. To apply a controlled operation, first, Libket needs to open a controlled scope pushing control qubits to a stack using the `ctrl_begin()` function, then call the quantum operation. It is possible to stack more control qubits inside the scope with successive `ctrl_begin()` calls. The `ctrl_end()` function ends the last open controlled section popping its control qubits. It is possible to open a controlled scope inside an inverse section, and vice versa.

4.1.4 Dump & Metrics

Although it is not possible in quantum hardware execution, only on simulation, Libket allows returning the quantum state of qubits with the `dump` class. As the integer of `future` variables, the value of a `dump` is available only after the quantum execution. When simulating a quantum execution, a `dump` instance can implement fake measurements that do not collapse the quantum state or emulate infinite measures. However, a `dump` cannot influence a quantum execution since it has no side effects. TODO DEBUG

The function `report()` returns a `metrics` instance with information on the current state of the KQASM on the actual process. The metrics include the number of qubits allocated/deallocated/measured, single-qubit/controlled gates used, and Ket Bitwise plugins applied. Libket can only return the KQASM metrics before the quantum execution because, after that, Libket destroys the process and, consequently, the KQASM file.

4.1.5 Libket CLI

Every application linked with Libket, including the Ket language interpreter, provides the command line interface of Figure 7. Libket will show every quantum code sent to the execution into a file when set by the `--out` flag. Alternatively, with the `--no-execute` flag, Libket will not send the KQASM code execution, setting every `future` variable to 0.

```
Ket program options:
    -h [ --help ]               Show this information.
    -o [ --out ]                KQASM output file.
    -s [ --kbw ]  (=127.0.0.1)  Quantum execution (KBW) address.
    -p [ --port ]  (=4242)      Quantum execution (KBW) port.
    --seed                      Set RNG seed for quantum execution.
    --api-args                  Additional parameters for quantum
                                execution.
    --no-execute                Does not execute KQASM, measurements
                                return 0.
    --dump-to-fs                Use the filesystem to transfer dump
                                data.
```

Figure 7 – Libket Command Line Interface

To execute the quantum code, Libket sends an HTML GET with the KQASM encoded in `application/x-www-form-urlencoded` on the request's body. Then the Ket Bitwise Simulator server runs the KQASM code and returns a JSON file with dump states if any, and the integer values to fulfill the `future` variables. Unless the flags `--kbw` and `--port` are set, Libket sends the request to `localhost` port `4242`. By default, KBW sends the dump states encoded in Base64. With the flag `--dump-to-fs`, KBW will dump the quantum state to a temporary file and send the path to Libket.

When Libket sends a seed to KBW with the `--seed` flag, the quantum execution is deterministic, making the execution of two equal Ket programs return the same measurement results. Although it has no use in the Ket Bitwise Simulator server, the flag `--api-args` append URL parameter to HTML request.

4.2 QUANTUM GATE DECOMPOSITION

The transpiler performs source-to-source translation with quantum gate decomposition (ITEN et al., 2016; VARTIAINEN et al., 2004) and architecture-independent optimizations (KISSINGER; WETERING, 2020; NAM et al., 2018) to prepare the quantum code for quantum hardware execution or to estimate the resources needed for the quantum execution. The input and output languages can be the same, with just the decomposition of the multi-controlled quantum gates.

The Ket Quantum Assembly (KQASM) language is more expressive than the available quantum computers, limiting the execution of a Ket application. However, we can easily translate a subset of KQASM to OpenQASM and Quil for quantum hardware execution.

4.3 KET BITWISE SIMULATOR: QUANTUM SIMULATOR

Although some small quantum computers are freely accessible through the Internet, simulating one is helpful to test and debug quantum applications, even with the exponential complexity of the problem. Besides the need to wait in a queue, a quantum hardware execution is subject to noise that can invalidate the computation, interfering even with short instances of some quantum algorithms. On the other hand, simulated quantum executions do not need to comply with some quantum limitations, like the impossibility to check into the superposition and qubits connectivity (a construction constraint). Today, simulated execution can be better than quantum hardware execution to validate the concept of a quantum algorithm due to decoherence. We argue that even with large-scale fault-tolerant quantum computers, there will be a need for quantum simulators for quantum debugging and learning purposes.

Ket Bitwise Simulator (KBW) uses the Bitwise representation. In contrast to the usual approach that uses matrix multiplication and linear algebra to handle quantum numeric simulations, the Bitwise representation uses a hashmap to store the quantum state and bitwise operations to apply quantum gates. In summary, KBW represents a qubit $|\psi\rangle$ in the hashmap `qubits` with the following equivalence

$$|\psi\rangle = \sum_k \alpha_k |k\rangle \equiv \sum_k \texttt{qubits[}k\texttt{]} |k\rangle . \tag{21}$$

For the evolution of the quantum state, KBW implements a specific function for every quantum gate.Those implementations are independent of the number of qubits, unlike gates stored in matrices that scale exponentially with the number of qubits. In the Bitwise representation, KBW can apply controlled gates as fast as without control. So quantum gate decomposition, needed for a quantum hardware execution, increases simulation time.

The computational time of the Bitwise representation grows exponentially with the amount of superposition in the quantum system and not with the number of qubits as usual. However, our implementation also limits this exponential scaling by the amount of entanglement in the quantum system, meaning that simulating a quantum system with all qubits in superposition with sets of few entangled qubits can have up to an exponential speedup comparing with other simulators. Inspired by the Qrack simulator (STRANO et al., 2020), we arrange this improvement by keeping the qubits in separate hashmaps when they are not entangled. As KBW only implements single-qubit gates, besides plugins (Subsection 4.3.1), the only way to generate entanglement is by adding control qubits to the quantum gate. However, KBW only considers that the target and the control qubits are entangled if the control is under superposition.

4.3.1 Ket Bitwise Plugins

KBW implements all quantum gates supported by KQASM and allows user-defined quantum gates through plugins, using the C++ Ket Bitwise API that exposes the hashmap of the quantum simulation. In contrast to the gate decomposition of complex quantum operations, plugins are easy and fast to implement. For example, the modular exponentiation required by Shor's algorithm (SHOR, 1997) has a complexity of $O(N^3)$ (where N is the number to be factored) in terms of quantum gates (VAN METER; ITOH, 2005), requiring to perform the decomposition for every possible value of the parameters N and a. On the other hand, with a Ket Bitwise plugin, you can efficiently implement this operation for any classical parameter using familiar constructions.

Code 4.3.1 presents the core of the modular exponentiation plugin implementation (`ket_pown.so`) required by Shor's factorization algorithm. The plugin takes the numbers a and N as input parameters, constructing the resulting quantum state in `new_qubits` (lines 7-10), iterating over the base states of `qubits` (line 3). The base states of the second register are the result of binary exponentiation of a, x, and N (lines 4-6).

Code 4.3.1 – Core of `libket_pown.so` plugin. $|x\rangle|0\rangle \rightarrow |x\rangle|a^x \mod N\rangle$.

```
1  // L = $\lceil \log_2 N \rceil$;  size = $3 \times L + 1$
2  map new_map;
3  for (auto &i : qubits) { // qubits = $\sum_x \alpha_x |x\rangle \otimes |0\rangle^{\otimes L}$
4    auto reg1_reg2 = i.first[0] & ((1ul << size)-1); // = $|x\rangle|0\rangle$
5    auto reg1 = reg1_reg2 >> L;    // = $|x\rangle$
6    auto reg2 = pown(x, reg1, N); // = $|a^x \mod N\rangle$
7    reg1_reg2 |= reg2;
8    auto idx = i.first;
9    idx[0] = reg1_reg2;
10   new_qubits[idx] = i.second; // new_qubits += $\alpha_x |x\rangle|a^x \mod N\rangle$
11 }
12 qbits.swap(new_map); // qubits = $\sum_x \alpha_x |x\rangle|a^x \mod N\rangle$
```

4.3.2 Benchmark

As the simulation time depends on the amount of superposition and entanglement of the quantum system, we propose four benchmarks to test the KBW performance. We run the benchmarks against the quantum simulators of Table 6, with the

Table 6 – Quantum simulators used in the Benchmarks.

Simulator	Version	Reference
Cirq	0.11.1	Developers (2021)
Forest QVM	1.17.2	Smith et al. (2017)
ProjectQ	0.7.0	Steiger et al. (2018)
Q#	0.18	Krysta Svore et al. (2018)
QSystem	1.2.0	Rosa and Taketani (2020)
Qiskit-Aer	0.8.2	ANIS et al. (2021)
Qrack	5.4.0	Strano et al. (2020)
QuEST	3.2.1	Jones et al. (2019)

Table 7 – Computer setup used in the benchmarks.

	Model
CPU	Intel(R) Core(TM) i7-8565U
RAM	2x 8GB DDR4 (Speed: 2667 MT/s)
Linux	5.13.9-arch1-1

setup of Table 7. All experiments consist of a state preparation with qubits initialized in $|0\rangle$, followed by the measurement of all qubits.

The first benchmark is a GHZ state preparation, entangling all qubits in a superposition of only two base states.

$$\text{GHZ} = \frac{1}{\sqrt{2}}\left(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}\right) \tag{22}$$

As shown in Figure 8a, since the qubits are at two quantum states on every computation step, KBW has a linear scale as QSystem, the original implementation of the Bitwise representation (ROSA; TAKETANI, 2020). We see a similar result in the W state preparation benchmark (Figure 8b),

$$W = \frac{1}{\sqrt{n}}(|100\ldots0\rangle + |010\ldots\rangle + |001\ldots0\rangle + \cdots + |000\ldots1\rangle) \tag{23}$$

where the number of base states in superposition is linear with the number of qubits.

The third benchmark applies a Quantum Fourier Transformation (QFT), leaving the qubits in a superposition of 2^n base states. However, since the initial state is $|0\rangle$, the final quantum state

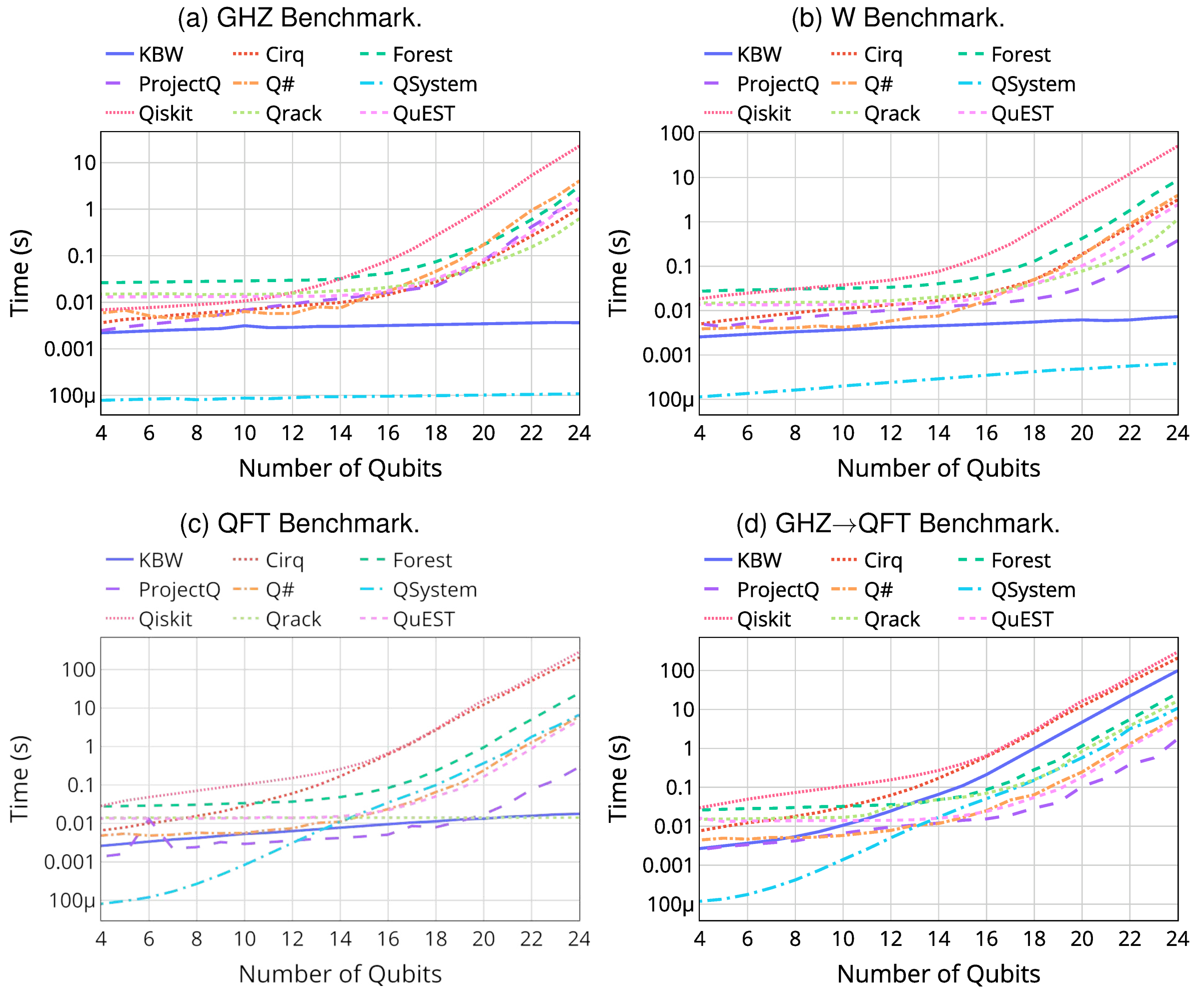
$$|\psi\rangle = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle \tag{24}$$

has no entanglement, allowing KBW to store the qubits in separated spaces. In Figure 8c, we see a linear scale similar to Qrack, as expected.

The last benchmark is the worst case for KBW, where we have all qubits entangled in total superposition. This experiment first prepares a GHZ state then applies a

QFT. We can see in Figure 8d that KBW scales exponentially as the others simulators. We chose these simple benchmarks to illustrate when KBW can have advantage over the other simulator. Those advantages have a different impact on different algorithms. For example, Shor's algorithm runs faster than Grover's algorithm in KBW, even that it uses more qubits. In Figure 9a, we compare Grover's and Shor's algorithms execution time in KBW. Code 4.3.2 presents the used Grover's algorithm implementation, and Code 4.3.3, the order-finding algorithm used in Shor's algorithm. The execution time of Figure 9a considers the execution of Shor's algorithm as a whole and not only the order-finding routine.

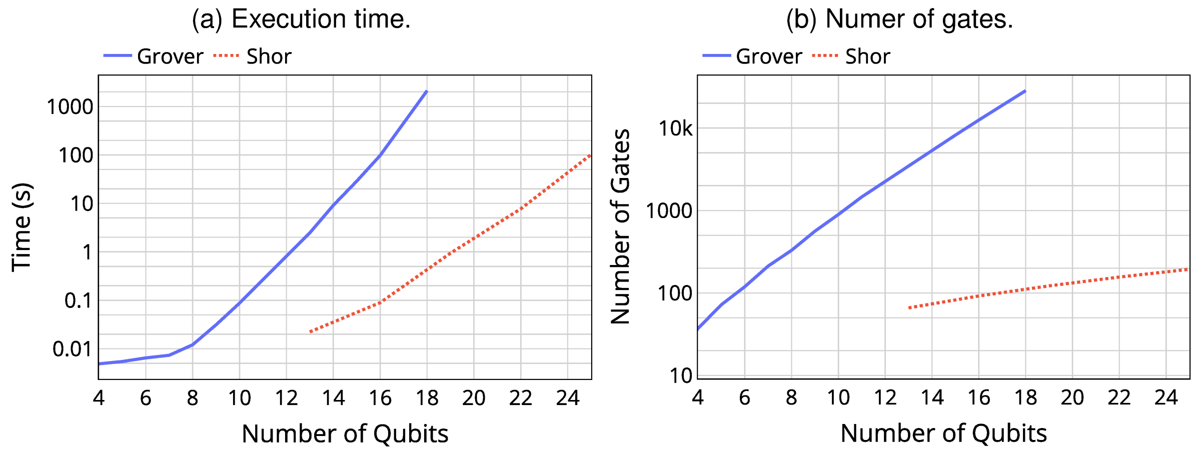
Figure 8 – Benchmarks.



4.4 QUANTUM HARDWARE EXECUTION

Quantum assembly describes the execution of a quantum application in terms of logical qubits. Those are ideal qubits, free of noise and fully connected, where no error corrections are needed, and every qubit communicates with each other, *e.g.*, it is possible to apply a CNOT between any qubit. However, quantum computers work

Figure 9 – Comparison between Groves' algorithm and Shor's algorithm.



Code 4.3.2 – Grover's search algorithm (GROVER, 1997).

```
def U(a, b):
    # U|a⟩|b⟩ = |a⟩|f(a) ⊕ b⟩,
    # f(a) = 1 if a = 3 else 0
    with control(a, on_state=3):
        X(b)

q = quant(size)
H(q) # q = |+⟩^⊗size
aux = quant()
X(aux) # aux = |1⟩
H(aux) # aux = |−⟩
steps = int((pi/4)*sqrt(2**size))
for _ in range(steps):
    U(q, aux) # Apply oracle
    with around([H, X], q):
        ctrl(q[1:], Z, q[0])
result = measure(q)
```

with physical qubits subject to noise (errors) and connectivity limitations. An example is the quantum computer IBMQ Melbourne of 15 qubits, which Figure 10 shows its qubit connectivity and calibration.

Logical qubits of a quantum assembly need to be mapped into physical qubits to run on a quantum computer. This process is called qubit allocation (SIRAICHI et al., 2018) or quantum circuit mapping (ITOKO et al., 2020). Heuristics for optimization may vary from each quantum computer vendor since each one explores different qubits layouts and construction technology.

Despite being too expensive for NISQ computers, quantum error correction is a fundamental part of fault-tolerant quantum computers (DEVITT et al., 2013). So, the

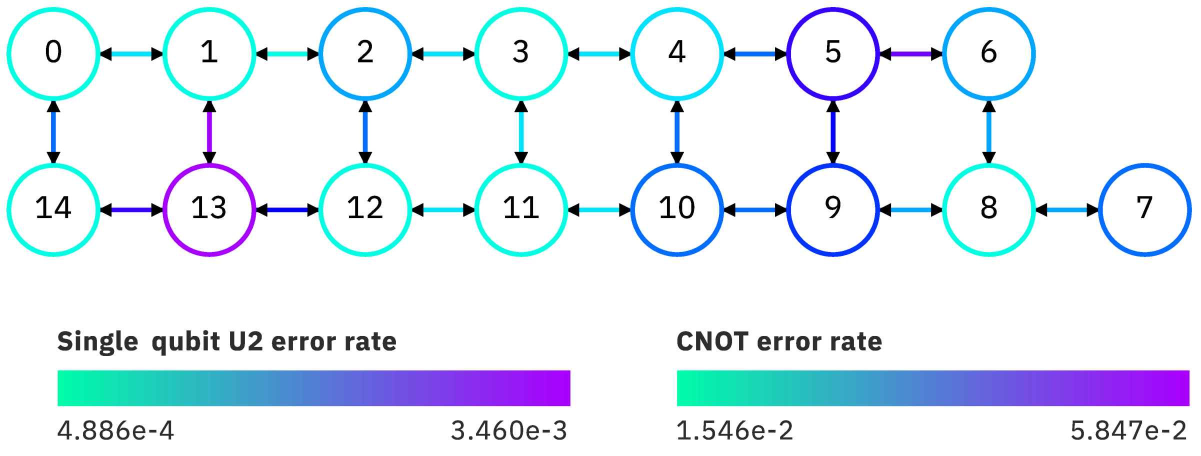
Code 4.3.3 – Order-finding quantum algorithm used in Shor's algorithm (SHOR, 1997).

```
from ket.lib import qft
from ket.plugins import pown
L = N.bit_length()
# a = random number coprime to N

def find():
    reg1 = quant(2*L+1)
    H(reg1) # reg1 = $|+\rangle^{\otimes 2L+1}$
    reg2 = pown(a, reg1, N) # ket_pown.so plugin: Code 4.3.1
#   $|\text{reg1}\rangle|\text{reg2}\rangle = |\text{reg1}\rangle|a^{\text{reg1}} \mod N\rangle$
    adj(qft, reg1)
#   reg1 = $\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}|k\frac{2^{2L+1}}{r}\rangle\, e^{\theta_k}$
    return measure(reg1).get()

r = reduce(gcd,[find() for _ in range(3)])
order = 2**(2*L+1)//r
```

Figure 10 – Quantum computer ibmq_16_melbourne v2.0.6 calibrated at April 16, 2020. The nodes represent the qubits and the links represent the possibility of applying a CNOT between two qubits. Screenshot by author from https://quantum-computing.ibm.com.



quantum assembly may pass through a QEC encoder before been sent to a quantum computer. Similar to the qubit allocation, QEC encoding also depends on the quantum architecture and may be performed by a proprietary software provided by the quantum computer vendor.

With the implementation of the proposed runtime architecture, the Ket Quantum Programming framework, we focus on noiseless simulated quantum execution, which is enough to validate our proposal. Quantum hardware execution, as well as the necessary process like quantum gate decomposition, are future works. However, today's quantum computers have limited control flow support that restricts the KQASM code execution. Although, we expect that quantum computers will overcome this restriction soon.

4.5 CONSIDERATIONS

The proposed runtime architecture provides the means for the dynamic hybrid classical-quantum programming and execution of the Ket language. With its core component, the shared library Libket, we meet the two first specific objectives of this work. To mitigate the interaction limitation between classical and quantum computers, Libket introduces the `future` variables to delay the quantum execution, allowing Ket to integrate it with the classical Python construction `if` and `while`. To support generic quantum programming and dynamic quantum execution, minimizing the number of quantum operations and classical control in the KQASM code, Libket uses the classical runtime to generate the quantum code.

With the quantum computer simulator Ket Bitwise Simulator, we meet the fourth specific objective of this work. The KBW implementation fits in the quantum execution scenario of Section 1.3, processing in batch, not supporting interaction during the quantum computation. Also, we implemented KBW to run in the cloud, accessible to Libket through a REST API. As presented in the QFT benchmark (Figure 8c), KBW improves the Bitwise representation by keeping the qubit states split when not entangled, leading to an exponential speed-up when limiting the amount of entanglement of the quantum system.

5 KET: A NOVEL QUANTUM PROGRAMMING LANGUAGE

Ket is a Python-embedded classical-quantum programming language that friendly exposes the Libket functionalities enabling dynamic quantum programming in an architecture suitable for the current and near-future quantum computers. As a high-level quantum programming language, Ket lets you quickly develop and test quantum algorithms and applications using the powerful Python syntax with the addition of a few quantum specific constructions. This approach should be a natural way for Python programmers to start with quantum programming, smoothing the learning of quantum computation and leveraging quantum software development.

Briefly, Ket provides three types/classes for quantum programming, a universal set of single-qubit quantum gates, constructions for controlled and inverse quantum operations, and quantum computer's classical control flow integration. In this chapter, we present the syntax and semantics of those features with examples of use. We also discuss the design decisions and limitations of Ket's implementation as an embedded language.

5.1 TYPES & QUANTUM OPERATIONS

Ket extends Python with three types designed for quantum programming: the `quant` type, which implements a list of qubits opaque reference used in quantum operations; the `future` type that holds a reference to a 64-bits integer stored in a quantum computer, such as a measurement result; and, the `dump` type that contains a snapshot for the quantum state, useful for debugging quantum applications with a simulator.

The `quant` construction allocates new qubits in the state $|0\rangle$ and returns its reference. The built-in single qubit quantum gates of Table 8 and the measure function are the only ones that can change the quantum state. Therefore no other operation has quantum side effects, ensuring that a `quant` variable manipulation does not violate the no-cloning theorem (WOOTTERS; ZUREK, 1982) or result in a non-unitary evolution. One can index qubits of a `quant` variable using brackets as in a Python list and concatenate `quant` variables with the addition operation.

When no longer in usage, one can free a `quant` variable so another `quant` can allocate its qubits. Before the call for free, all qubits must be at the state $|0\rangle$. Otherwise, the `quant` variable must release the qubits as dirty. A `quant` can allocate dirty qubits, but their usage may cause side effects due to previous entanglements. It is possible to use the `with` statement to ensure that Libket free all qubits of a `quant` variable at the end of the scope. This construction helps not leak auxiliary qubits. In code 5.1.1, we present several examples of how to create and manipulate a `quant` variable. The operations of lines 2, 4, and 5 allocate qubits, and lines 9 to 14 show several ways to index qubits inside a `quant` variable. The operations of lines 23, 24, 30, and 33 are quantum gate

Table 8 – Quantum gates available in Ket.

Gate	Function	Effect
Pauli-X (σ_x)	`X(q: quant)`	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$
Pauli-Y (σ_y)	`Y(q: quant)`	$Y\|0\rangle = -i\|1\rangle$ $Y\|1\rangle = i\|0\rangle$
Pauli-Z (σ_z)	`Z(q: quant)`	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$
Hadamard	`H(q: quant)`	$H\|0\rangle = \frac{1}{\sqrt{2}}\|0\rangle + \frac{1}{\sqrt{2}}\|1\rangle$ $H\|1\rangle = \frac{1}{\sqrt{2}}\|0\rangle - \frac{1}{\sqrt{2}}\|1\rangle$
S	`S(q: quant)`	$S\|0\rangle = \|0\rangle$ $S\|1\rangle = i\|1\rangle$
S-dagger	`SD(q: quant)`	$S^\dagger\|0\rangle = \|0\rangle$ $S^\dagger\|1\rangle = -i\|1\rangle$
T	`T(q: quant)`	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = \frac{1+i}{\sqrt{2}}\|1\rangle$
T-dagger	`TD(q: quant)`	$T^\dagger\|0\rangle = \|0\rangle$ $T^\dagger\|1\rangle = \frac{1-i}{\sqrt{2}}\|1\rangle$
Phase	`phase(λ: float[, q: quant])`	$P\|0\rangle = \|0\rangle$ $P\|1\rangle = e^{i\lambda}\|1\rangle$
Rotation-X	`RX(θ: float[, q: quant])`	$RX\|0\rangle = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$ $RX\|1\rangle = -i\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$
Rotation-Y	`RY(θ: float[, q: quant])`	$RY\|0\rangle = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$ $RY\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$
Rotation-Z	`RZ(θ: float[, q: quant])`	$RZ\|0\rangle = e^{-i\theta/2}\|0\rangle$ $RZ\|1\rangle = e^{i\theta/2}\|1\rangle$

applications, and the call of line 34 free the qubit of the `quant` variable `aux`, which is necessary because of the statement of line 25.

The build-in functions of Table 8 that implement single-qubit gates take a `quant` variable as input and apply the quantum operation on every qubit of it. The gates also return the input `quant` variable, allowing to chain the quantum calls, *e.g.*,

$$SHX|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \equiv \texttt{S(H(X(quant())))}. \tag{25}$$

For the parameterized quantum gates `phase`, `RX`, `RY`, and `RZ`, the input `quant` is optional, and when not provided, the function return a new gate with the given angle parameter. For example, it is possible to implement the gates `Z`, `S`, `SD`, `T`, and `TD` as

```
Z = phase(pi);     S = phase(pi/2);   SD = phase(-pi/2);
T = phase(pi/4);  TD = phase(-pi/4)
```
. (26)

Code 5.1.1 – Examples of `quant` variable manipulation.

```
# Allocate 1 qubit
q  = quant()
# Allocate 20 qubits
qs = quant(20)
# Allocate 10 dirty qubits
d_qs = quant.dirty(10)

# Qubit index as in Python list
head, tail  = qs[0], qs[1:]
head, *tail = qs
init, last  = qs[:-1], qs[-1]
*init, last = qs
even_qubits = qs[::2]
odd_qubits  = qs[1::2]

# Invert qubits order
reverse = reversed(q)
# Unpack quant
a, b = quant(2)
# Concatenate quant
c = a+b

H(a) # Hadamard gate
X(b) # Pauli-X gate
with quant() as aux:
    # aux must be free before
    # the scope ends
    with around(H, aux):
        with control(aux):
            swap(a, b)
    result = measure(aux)
    if result == 1:
        X(aux)
    aux.free() # Free qubit |0⟩
```

The only way to gather classical information about a quantum state is through measurement. The function `measure(q: quant, free: bool = False) -> future` measures all qubits of a `quant` and returns a `future` that refers to the measurement result, which is the concatenation of every resulting bit. Measurements always collapse the quantum state, and in simulated quantum execution, the measured qubits are still available to use. Optionally, it is possible to free the `quant` after the measure by setting the argument to `True`.

The `future` variable resulting from a measure holds a promise for the measurement result because the value is not immediately available for the classical computer. To get the value from the quantum computer, first, it needs to execute the quantum code (see Chapter 4 for more details), which is triggered by the method `get` of a `future` or the function `exec_quantum`.

Any binary operation between `future` variables or `future` and `int` returns a new `future` variable that refers to the resulting value in the quantum computer. The Python assignment statement always overrides the variable in the classical computer, so to overrides a variable in the quantum computer one can use the syntax:

```
<var>.set = <exp>
```
(27)

In Code 5.1.2, we present ways to initialize a `future` variable, including the `future` constructor (line 4) that passes a number to the quantum computer.

In simulated quantum execution, it is possible to take a snapshot of the quantum state and store it in a `dump` variable. It can come in handy for debugging a quantum application or helping to understand a quantum algorithm. The non-cloning theorem and the impossibility of verifying the superposition without collapsing the quantum state make it impossible to implement this feature in quantum hardware. Like a `future` variable, the result of a `dump` is available only after the quantum execution. Making the `dump` of a `quant` has no side effects in the quantum execution, but the `dump` value can easily take many gigabytes of memory.

A `dump` variable can simulate multiples measurements to speed the simulation of quantum algorithms that need to run several times to take a measurement statistic. In code 5.1.3, we present a commented example of `dump` usage.

Every operation performed in the quantum computer is attached to a process, which one can create using the `with run()` statement. Ket allows to nest process, but as they are a separated quantum execution, there is no communication between them (like a scope in programming languages). The operations performed outside of a `with-run` statement communicates with the global process, initialized at the program startup.

5.2 CONTROLLED OPERATIONS

In addition to the single-qubit gates of Table 8, Ket provides the CNOT and SWAP gates. However, those are not built-in gates, instead, they are adding qubits of control to the available single-qubit gates. In code 5.2.1, we present a possible implementation of the CNOT gate using the `ctrl(ctl: quant, func, *args)` function to add control qubits to the Pauli-X gate (line 4). To implement multi-qubits gates, Ket allows adding multiples control qubits to any quantum gate or callable using the mentioned `ctrl` and the `with control(*ctl: quant[, on_state])` statement. This feature is enough to implement complex multi-qubits quantum operations. For example, the Quantum Fourier Transformation of Code 5.2.2 uses only single-qubit gates and the `with-control` construction to call a controlled-phase gate (lines 8-9).

The `with-control` is the quantum analog for the `if-then` classical statement, applying the quantum operations only when the control qubits are in a given state, taking into account the quantum superposition. By default, the quantum computer only performs the quantum operations if all control qubits are in state $|1\rangle$. Ket allows

Code 5.1.2 – Examples of `future` initialization.

```
q = H(quant(10))
a = measure(q) # New future from measure
b = m*2         # New future from a binary operation
c = future(42) # New future from a given value
a.set = b+c*3   # Overwrite the value on the quantum computer
exec_quantum() # Get the values from the quantum computer
```

Code 5.1.3 – Example of `dump` usage.

```
q = quant(3)
ctrl(H(q[:-1]), X, q[-1])
d = dump(q)
states = d.states() # Get the basis states of q, [0, 2, 4, 7]
print('Number of basis states:', len(states))
#Number of basis states: 4
print('The amplitude of state |111⟩:', d.amplitude(0b111))
#The amplitude of state |111⟩: (0.4999999999999999+0j)
print('Probability of measure state |000⟩:', d.probability(0))
#Probability of measure state |000⟩: 0.2499999999999999
print(d.show('b2')) # Print the state, separate the two first qubits
#|00⟩|0⟩                (25%)
#0.5                     ≅  1/√4
#
#|01⟩|0⟩                (25%)
#0.5                     ≅  1/√4
#
#|10⟩|0⟩                (25%)
#0.5                     ≅  1/√4
#
#|11⟩|1⟩                (25%)
#0.5                     ≅  1/√4
```

Code 5.2.1 – Possible implementation of the CNOT gate in Ket.

```
def cnot(ctl : quant, trg : quant ):
    # CNOT ctl[i], trg[i]
    for c, t in zip(ctl, trg):
        ctrl(c, X, t)
```

modifying this behavior with the keyword argument `on_state` that changes the control state. As we assumed that the quantum computer only controls in state $|1\dots1\rangle$, to change the control state, before calling for controlled operation, Ket takes the desired control state to $|1\dots1\rangle$ and inverse this permutation after the call.

Controlled operations can generate entanglement if the control qubits are in superposition with a control state. For example, in Code 5.2.3, the `quant ctl` is in a superposition of the states $|0\rangle$ (the control state, defined in line 3) and $|1\rangle$, taking the `quant trg` in a superposition where the Pauli-X gate was applied and was not applied (lines 3-4). As the two qubits are entangled now, for example, if we measure 0 in the control qubit, the other will collapse to a state where the operation was applied. In Code 5.2.3, line 16, we see that in this particular execution, the measurement of the control

Code 5.2.2 – Quantum Fourier Transformation (QFT) in ket.

```python
def qft(q : quant, invert : bool = True) -> quant:
    if len(q) == 1:
        H(q)
    else:
        head, *tail = q
        H(head)
        for i in range(len(tail)):
            with control(tail[i]):
                phase(2*pi/2**(i+2), head)
        qft(tail, invert=False)

    # At the end of this Quantum Fourier Transformation, the qubits are
    # in the reversed order. Depending on the algorithm, swapping the
    # qubits reference is enough.
    if invert:
        for i in range(len(q)//2):
            swap(q[i], q[len(q)-i-1])
        return q
    else:
        return reversed(q)
```

qubit returned 1, collapsing the `quant` `trg` to not applied.

Except for qubit allocation, deallocation, measure, and classical operation in the quantum computer, *e.g.*, `future.set`, any other instruction is allowed inside a controlled scope, even call functions that call other controlled gates. Same for the `ctrl`, which adds qubits of control to a callable. With the function `ctrl`, it is also possible to create a new controlled quantum gate that takes a `quant` as input. For example, we can implement a multi-controlled-not gate that uses the first qubits of a `quant` variable to control the last as “`mcnot = ctrl(slice(0, -1), X, -1)`”. For this construction, the quantum gate or function must take only a `quant` as input and call `ctrl` with the signature:

```
(ctl : int | slice, gate, trg : int | slice) -> Callable[[quant], quant]
```
(28)

5.3 INVERSE OPERATIONS

As quantum computation is intrinsically reversible and several quantum algorithms use inverse quantum operations, Ket provides means for calling functions inverted, facilitating inverse quantum gates application. For example, to apply an Inverse Quantum Fourier Transformation needed for Shor’s factorization algorithm, we can use the `adj(func, *args)` function to call the inverse of the QFT implemented in Code 5.2.2.

Code 5.2.3 – Entangling qubits with a controlled operation.

```
ctl = H(quant()) # = 1/√2(|0⟩ + |1⟩)
trg = quant()    # = |0⟩
with control(ctl, on_state=0):
    X(trg)       # 1/√2(|0, X(trg)⟩ + |1, trg⟩)

before_measure = dump(ctl+trg)
measure(ctl)     # |1, trg⟩, collapse to not applied
after_measure = dump(ctl+trg)
print(f"Before the measure:\n{before_measure.show('b1:b1')}")
#Before the measure:
#|0⟩|1⟩                (50%)
#0.707107                 ≅  1/√2
#
#|1⟩|0⟩                (50%)
#0.707107                 ≅  1/√2
print(f"After the measure:\n{after_measure.show('b1:b1')}")
#After the measure:
#|1⟩|0⟩                (100%)
#1                        ≅  1/√1
```

Alternatively, it is possible to implement an Inverse Quantum Fourier Transformation, writing the QFT implementation inside a scope started by a `with inverse()` statement, as in Code 5.3.1.

Code 5.3.1 – Inverse Quantum Fourier Transformation (IQFT) in ket.

```
def iqft(q : quant):
    with inverse():
        for i in range(len(q))
            head, tail = q[i], q[i+1:]
            H(head)
            for j in range(len(tail)):
                with control(tail[j]):
                    phase(2*pi/2**(j+2), head)
        for i in range(len(q)//2):
            swap(q[i], q[len(q)-i-1])
```

The `adj` function allows creating an inverse quantum operation from a callable. For example, it is possible to implement the gates `SD` and `TD` and the operation IQFT as

$$\texttt{SD = adj(S);}\quad \texttt{TD = adj(T);}\quad \texttt{iqft = adj(qft)}\ . \tag{29}$$

A usual construction of quantum algorithms is $VUV^{\dagger}$, where U and V are unitary operators (quantum gates), and $V\dagger$ is the inverse of V. For example, a possible implementation for Grover's diffusion operator is $VUV^{\dagger}$, with $U = XH$ and V = Multi-controlled-Z. In Ket, we can implement this operator with the `with around(func, *args)` statement (see Code 5.3.2). Using the `with-around` construction, the operators U and V are, respectively, the new scope and the arguments. We implement this statement inspired by the `within-apply` construction of Q# that has a similar semantic.

Code 5.3.2 – Grover's diffusion operator in Ket.

```python
def grover_diffusion(q : quant):
    with around([H, X], q):  # start: H(X(q))
        ctrl(q[1:], Z, q[0]) # end:   adj([H, X], q)
```

We can use the `with-around` statement to remove entanglement with auxiliary qubits. For example, in Code 5.3.3, the `with-around` of line 9 starts entangling the primary qubits with the auxiliary ones; and where the scope ends, the statement removes the entanglement applying the inverse operation. Except for qubit allocation, deallocation, measure, and classical instructions in the quantum computer, *e.g.*, `future.set`, it is possible to invert any other instruction. Controlled and inverse operations can interoperate without restrictions.

Code 5.3.3 – Oracle $|x\rangle\,|y\rangle \rightarrow |x\rangle\,|f(x) \oplus y\rangle$, where $f(x) = 1$ if x is periodic, else $f(x) = 0$.

```python
 1  def evaluate_periods(query : quant, aux : quant):
 2      for period in range(1, len(aux)+1):
 3          with around(ctrl(query[i+period], X, query[i], later_call=True)
            ↪   for i in range(len(query)-period)):
 4              with control(query[:len(query)-period], on_state=0):
 5                  X(aux[period-1])
 6  
 7  def oracle(x : quant, y : quant):
 8      with quant(len(x)//2) as aux:
 9          with around(evaluate_periods, x, aux):
10              with control(aux, on_state=0):
11                  X(y)
12              X(y)
13          aux.free()
```

5.4 QUANTUM COMPUTER'S CONTROL FLOW

We can split the Ket execution into classical and quantum runtime. As presented in Section 4.1, Libket generates the quantum code in the classical runtime using quantum and classical operations, including classical conditional statements. When using a `future` variable as the test for an `if-then-else` or a `while` statement, Ket cannot execute those in the classical runtime since the value is not available for the classical computer yet. So, Libket constructs the semantics of the statements in the quantum code to execute in the quantum runtime. The integration of the type `future` with the Python statements `if-then-else` and `while` enables programming the classical control flow in the quantum computer, as if the values were in the classical computer. This approach seamlessly integrates the quantum computer's control flow in a hybrid classical-quantum programming architecture that respects the limitations of the scenario of Section 1.3.

The Quantum Teleportation protocol (Figure 6) is an example of an application that uses measurement results to control the application of quantum gates. In Code 5.4.1, we present a Quantum Teleportation implementation in Ket (left) and the quantum code generated at runtime (right). On the left, lines 17-22, we see the measurements of lines 15 and 16 controlling the application of the gates Pauli-X and Pauli-Z. Note that the measurement result is not available in the classical computer at this time, but we use the same syntax as if it was. On the left, lines 11-24, we see that Libket placed the semantics of lines 17-22 in the KQASM code, moving the execution of the `if-then` statement to the quantum computer. Also, note that the `if-then` instructions of the `bell` function (left, line 3-6) are not present in the KQASM code since the values of the tests are available in the classical computer when called in line 12. However, if the parameters `x` and/or `y` of the function `bell` are the type `future`, Ket would tell Libket to place those `if-then` statements in the quantum code.

In code 5.4.1, left, the number in the comments matches the generated line in the KQASM code (right). The `future` variables `m0`, `m1`, and `result` refer to the registers `i0`, `i1`, and `i6`, respectively, on the quantum code, and the `quant` variables `alice`, `alice_b`, `bob` refer to the qubits `q0`, `q1`, and `q2`. The KQASM code only executes in line 28, triggered by the method `get` of the `future result`.

Ket also integrates the Python `while-else`[1], `continue`, and `break` statements in the quantum computer. Code 5.4.2 present an example of a `while` and `if-then-else` statements that executes on the quantum computer to prepare a quantum state using postselection. In the left code, the number in the comments matches the generated line in the KQASM code (right). The `future` variables `ok` and `res` refer to the registers `i0`, and `i3`, respectively, on the quantum code, and the `quant` variables `q` and `aux` refer to

[1] Python allows an optional `else` clause that executes if the loop does not end with a `break` statement.

Code 5.4.1 – Quantum Teleportation implemented in Ket.

Ket

```
 1  def bell(x, y) -> quant:
 2      q = quant(2) # 3-4
 3      if x == 1:
 4          X(q[0])
 5      if y == 1:
 6          X(q[1])
 7      H(q[0]) # 5
 8      ctrl(q[0], X, q[1]) # 6
 9      return q
10
11  def teleport(alice) -> quant:
12      alice_b, bob = bell(0, 0)
13      ctrl(alice, X, alice_b) # 7
14      H(alice) # 8
15      m0 = measure(alice) # 9
16      m1 = measure(alice_b) # 10
17      if m1 == 1: # 11-14
18          X(bob) # 15
19          # 16-17
20      if m0 == 1: # 18-21
21          Z(bob) # 22
22          # 23-24
23      return bob
24
25  alice = quant() # 2
26  bob = teleport(alice)
27  result = measure(bob) # 25
28  result.get() # Execute KQASM
```

Generated KQASM

```
 1  LABEL @entry
 2      ALLOC    q0
 3      ALLOC    q1
 4      ALLOC    q2
 5      H        q1
 6      CTRL [q1],      X    q2
 7      CTRL [q0],      X    q1
 8      H        q0
 9      MEASURE  i0   [q0]
10      MEASURE  i1   [q1]
11      INT      i2    1
12      INT      i3    i1    ==   i2
13      BR    i3  @if.then0  @if.end1
14  LABEL @if.then0
15      X        q2
16      JUMP     @if.end1
17  LABEL @if.end1
18      INT      i4   1
19      INT      i5   i0    ==   i4
20      BR    i5  @if.then2  @if.end3
21  LABEL @if.then2
22      Z        q2
23      JUMP     @if.end3
24  LABEL @if.end3
25      MEASURE  i6   [q2]
```

the qubits `q0`-`q1` and `q2`. The KQASM code only executes in line 17, triggered by the method `show` of the `dump` `d`. In Appendix A, Code A.2.3 presents the same code but using the `break` statement.

To tell if the statement will run in the quantum computer or the classical computer, Ket checks the type of the test expression. If the test is an instance of the `future`, then the code runs on the quantum computer; else, it runs on the classical computer. The type checking is made at runtime since it is impossible to define the variable type a priori in Python. To implement the semantic of the statements `if-then-else` and `while`, the Ket interpreter uses the Python AST to make the necessary adaptations before the execution. We present the AST transformation in Appendix A.

Code 5.4.2 – Quantum state $\frac{1}{\sqrt{3}}(|00\rangle + |01\rangle + |10\rangle)$ preparation with postselection.

Ket

```python
q = quant(2) # 2-3
with quant() as aux: # 4
    ok = future(False) # 5
    while ok != True: # 6-11
        H(q) # 12-12
        ctrl(q, X, aux) # 14
        res = measure(aux) # 15
        if res == 0: # 16-19
            ok.set = True # 20-21
            # 22
        else: # 23
            X(q+aux) # 24-26
            #27-28
        # 29-30
    aux.free() # 31
d = dump(q) # 32
print(d.show()) # Execute KQASM
#|00⟩                   (33.3333%)
#0.57735                    ≅  1/√3
#
#|01⟩                   (33.3333%)
#0.57735                    ≅  1/√3
#
#|10⟩                   (33.3333%)
#0.57735                    ≅  1/√3
```

Generated KQASM

```
1  LABEL @entry
2      ALLOC     q0
3      ALLOC     q1
4      ALLOC     q2
5      INT       i0    0
6      JUMP      @while.test0
7  LABEL @while.test0
8      INT       i1    1
9      INT       i2    i0    !=    i1
10     BR i2 @while.loop1 @while.end2
11 LABEL @while.loop1
12     H         q0
13     H         q1
14     CTRL [q0, q1], X    q2
15     MEASURE   i3   [q2]
16     INT       i4    0
17     INT       i5    i3    ==    i4
18     BR    i5   @if.then3  @if.else4
19 LABEL @if.then3
20     INT       i6    1
21     SET       i0    i6
22     JUMP      @if.end5
23 LABEL @if.else4
24     X       q0
25     X       q1
26     X       q2
27     JUMP    @if.end5
28 LABEL @if.end5
29     JUMP    @while.test0
30 LABEL @while.end2
31     FREE    q2
32     DUMP    [q0, q1]
```

5.5 DESIGN DECISIONS AND LIMITATIONS

Next, we will discuss some design decisions and limitations of the Ket programming language implementation.

Control: Quantum controlled operations are the quantum analog of the `if-then` statement, so why does not use the Python `if-then` to apply controlled gates instead of the `with-control`? The answer is: to avoid confusing programmers. A controlled operation is not equal to a quantum `if-then` because it requires all qubits to be in the state $|1\rangle$ to execute, and not in a state different of $|0\dots0\rangle$ to execute. So, the use of `if-then` to express controlled operation could mislead programmers, hoping the quantum program to apply the traditional semantics into the quantum state.

Run: The ability to describe multiple unrelated quantum execution is essential to program a more extensive quantum application. In the Ket programming language, it is achieved by using the `with-run` statement. However, the ability to automatically attach the operations on the quantum computer to a process based on the qubit entanglement would increase the coding dynamism. We plan this feature for future implementations.

No-cloning theorem: As described by Wootters and Zurek (1982) in the no-cloning theorem, we cannot copy a qubit. So, the semantic of a quantum programming language should ensure this property. A possible strategy, implemented in QWIRE (PAYKIN et al., 2017), is to use a linear type system on the quantum variables to secure the no-violation of the theorem. However, Ket uses a different approach. A `quant` variable holds an array of qubits reference and not the actual qubits. This perspective allows the use of the `quant` type in assignment statements without the worry of the no-cloning theorem since the language will perform it by referencing and not copying (see Code 5.5.1 as an example).

Code 5.5.1 – Example of the effect of an assignment with a `quant`.

```
a = quant(5) # a = |00000⟩
X(a)         # a = |11111⟩
b = a[2:4]   # b = |11⟩
X(b)         # b = |00⟩ and a = |11001⟩
```

Future indexing qubits: While indexing the qubits of a `quant` using a `future` variable would improve coding dynamics, This feature would make it impossible to identify the violation of the no-cloning theorem during the classical computer runtime. For example, in a controlled quantum operation, we cannot ensure that a `future` variable is not indexing a control qubit used as a target and vice versa.

Get: The `future.get` tells Libket that the local computer needs a value from the quantum computer. Ideally, a static analysis of the source code should provide this information, performing the `get` when necessary. However, this analysis would either fail or impose a limitation in a language like Python, the Ket base-language.

Set: Since the assignment statement is essential for the Python dynamism, it does not allow the overload of the operator (the `__setattr__` does not qualify). So, Ket provides the `future.set` to assign a value to a `future` variable. We chose not to change the language semantics for this type.

Overhead: As Python only defines the variable types at runtime, the integration of the `future` type in the statements `if-then` and `while` requires additional instructions to identify if it will run in the classical or quantum computer. Those supplementary operations introduce overhead on the statement execution.

A new language? Why do we not provide Ket as a Python package instead of a new language? As the integration of the `future` type with the Python control statements is fundamental for our proposal, providing Ket only as a Python package would not support it. The Ket interpreter uses the Python parser to generate the program AST and the Python compiler to compile it. However, before sending the tree for the compiler, the interpreter performs some additional transformations (see Appendix A for details). You can also use Ket as a Python library (with limitation), and we provide the function `import_ket` to import a Ket code as a Python module and the decorator `code_ket` to handle `future` variables inside a Python function.

5.6 CONSIDERATIONS

With Ket, we improve over the two first specific objectives of this work that we already met with Libket. The seamless integration of the type `future` with the Python statements `if-then-else` and `while` makes easy the interaction between classical and quantum computers; and allows generic hybrid classical-quantum programming, where the same code can run on the classical or quantum computer depending on where the information is. Ket also validates our proposal, which is our third specific objective.

We compare the expressivity power of Ket with Q#, a domain-specific language that allows hybrid classical-quantum programming, but only on the quantum computer side[2]. While Q# requires a general purpose classical programming language to coordinate the quantum computer execution, Ket allows programming a complete classical-quantum application with a single source code. None of the other Classical-Quantum Programming languages in our related works (Section 3.2) allows the separation of classical and quantum code needed for the cloud-based quantum computation scenario. Therefore, we highlight that Ket, within our related works, is the only Classical-Quantum Programming language to support classical and quantum code separation, allowing each code to execute on its respective architecture.

Despite the limitation of implementing Ket as an embedded programming language, we believe that the quantum instructions added to Python interact well with the language, diverging as little as possible from the base syntax and semantic, being a good entry point for Python programmers in quantum programming. Ket also benefits from all the Python codebase and tools. We also highlight the `dump` instruction that helps in debugging, studying, and developing quantum algorithms.

For more Ket code examples, see in Appendix B, where we implemented some problems of the Microsoft Q# Coding Contest in Ket, comparing with the reference implementation in Q#.

[2] The quantum computer ability to execute simple classical instructions and control flow is a prerequisite for our proposed runtime architecture.

6 CONCLUSION

While quantum computers can provide exponential speedup in some problems, some intrinsic and construction limitations difficulties the implementation of classical-quantum applications. Due to decoherence, cloud-based quantum computers must execute a code as fast as possible, restricting the interaction between classical and quantum computers, making them batch processors.

Focusing on the constraints of cloud-based quantum computers and respecting the inherent restrictions of quantum programming, we presented the Ket Classical-Quantum Programming Language, which at runtime can generate quantum code, separating the quantum execution to run efficiently in the quantum computer. Also, making quantum code with as minimal control flow as possible to streamline the quantum computation.

With quantum computers processing in batch, a classical computer cannot interact with a quantum computer during execution, making it difficult to use classical information generated in the quantum computer to control the quantum execution. For example, the following interaction is not possible: During the quantum execution, the classical computer uses measurement results to control the application of quantum operations. Mitigate this limitation is the first specific objective in this work.

While we can rely upon the quantum computer's ability to execute control flow to implement generic quantum applications, this approach increases the execution time and the odds of decoherence. Provide generic quantum programming, keeping the quantum execution as specific as possible is the second objective of this work.

We met the two first secondary objectives of this work with the shared library Libket and improved our solution with the Python-embedded language Ket. With the `future` variables integrated with the Python statements `if-then-else` and `while`, Ket can seamlessly move the control flow from the classical computer to the quantum computer mitigating their interaction limitation. This integration also allows programming code that can run either in the classical and quantum computer, depending on where the value of the test variable is available. With Libket generating the quantum code at runtime, it can limit the quantum execution only to decisions and evaluations that the classical computer could not do, therefore, allowing Ket to use any Python construction to program generic classical-quantum applications. Ket implementation meets the third specific objective of this work, validating concepts of the proposed quantum programming language. Libket is C++ only, independent of Python and the Ket language, and can be used to implement a backend for other quantum programming languages like Silq.

To implement total support for Ket, first, a quantum computer needs to support classical control flow. However, this is not the case for today's NISQ computer, but we expect to see an initial control flow support shortly. Although, without control flow, Ket and its runtime architecture are suitable for current NISQ computers.

We believe Ket to be a convenient way for Python programmers to start with quantum computation, leveraging Python programming dynamics for qubit manipulation. Despite Ket implementing the same low-level qubit manipulation as Q#, it is possible to use classes to implement high-level behavior like auto uncomputation similar to Silq. With simulated quantum execution, Ket can use `dump` variables as a tool to make it easy to debug and study quantum algorithms. Even that quantum code visualization as a quantum circuit is not implemented yet, Libket can output the KQASM code sent to execution for inspection.

To execute the KQASM code generated by Libket, we implement the Ket Bitwise Simulator (KBW), improving the Bitwise representation. KBW detaches the simulation time from the number of qubits, allowing the simulation of multiple qubits in quantum systems with a low amount of superposition or entanglement with an exponential advantage in some cases and an exponential scale similar to other simulators in the worst case. With KBW, we meet the last specific objective of this work.

The Ket Quantum Assembly (KQASM) language takes advantage of the KBW ability to execute any one-qubit gate with many qubits of control without the need for quantum gate decomposition. When the controlled gate does not entangle the control and target qubits, the simulation time only increases linearly compared with the execution of the non-controlled gate. As KBW simulation time is independent of the number of qubits, KQASM does not need to provide any information a priori on the total number of qubits that KBW needs to simulate. Also, KBW can allocate and free qubits arbitrarily.

Even that quantum simulation has a high potential for parallel execution, implementing multithreading in KBW is a future work. Although, even without multithreading to take full advantage of the available hardware, the Bitwise representation alone is enough to enable the execution of some quantum algorithms faster and with more qubits than other benchmarked quantum simulators. For example, Pires et al. (2021) used 42 qubits to simulate a QAOA algorithm in a laptop in about 5min. The same simulation, with a state vector simulator, requires in the order of $2^{42} = 4.398.046.511.104$ of memory.

Ket Quantum Programming is an open-source project available at `https://gitlab.com/quantum-ket` under the Apache License 2.0. For the documentation, see `https://quantum-ket.gitlab.io`. We distribute the Ket interpreter (`ket-lang`) and Ket Bitwise Simulator (`kbw`) as compiled manylinux2014 wheel in the Python Package Index to be installed using pip.

We met all the specific objectives enumerated in Section 1.4, and the primary objective of this work with the Ket Quantum Programming development. The proposed runtime architecture with the development of Libket fulfills the secondary objectives **O1** and **O2**. The Ket implementation satisfies objective **O3**, and Ket Bitwise Simulator meets objective **O4**.

As future works, we plan for a formal specification of the Ket programming language, extending Python's grammar and semantics; and implement the handling of semantic errors and errors that can occur during quantum execution. In the proposal of the runtime architecture and the implementation of the Ket Quantum Programming, we do not rightly address the problem of debugging quantum programs, and there is a lack of precision in the description of the quantum hardware execution. As future works, we also want to address the quantum debugging problem and answer the following questions: (i) With the inability to look into the superposition and the no-cloning theorem, how to debug a quantum hardware execution? And, (ii) considering the exponential increase in information, how to efficiently debug a simulated quantum execution? For the second question, we believe that the `dump` variables are a good start point. However, we consider that assert and visualization tools would improve debugging efficiency.

REFERENCES

ALIFERIS, Panos; GOTTESMAN, Daniel; PRESKILL, John. Quantum accuracy threshold for concatenated distance-3 codes. **Quantum Information and Computation**, v. 6, n. 2, p. 97–165, Apr. 2005. ISSN 15337146. DOI: 10.26421/QIC6.2-1. arXiv: 0504218 [quant-ph].

ANIS, MD SAJID et al. **Qiskit: An Open-source Framework for Quantum Computing**. [S.l.: s.n.], 2021. DOI: 10.5281/zenodo.2573505.

ARUTE, Frank et al. Quantum supremacy using a programmable superconducting processor. **Nature**, v. 574, n. 7779, p. 505–510, Oct. 2019. ISSN 1476-4687. DOI: 10.1038/s41586-019-1666-5.

BEAZLEY, David M. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In: PROCEEDINGS of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4. Monterey, California: USENIX Association, 1996. (TCLTK'96), p. 15.

BENNETT, Charles H.; BRASSARD, Gilles; CRÉPEAU, Claude; JOZSA, Richard; PERES, Asher; WOOTTERS, William K. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. **Phys. Rev. Lett.**, American Physical Society, v. 70, p. 1895–1899, 13 Mar. 1993. DOI: 10.1103/PhysRevLett.70.1895.

BERGHOLM, Ville et al. **PennyLane: Automatic differentiation of hybrid quantum-classical computations**. [S.l.: s.n.], 2020. arXiv: 1811.04968 [quant-ph].

BHATTACHARJEE, Debjyoti; SAKI, Abdullah Ash; ALAM, Mahabubul; CHATTOPADHYAY, Anupam; GHOSH, Swaroop. MUQUT: Multi-Constraint Quantum Circuit Mapping on NISQ Computers: Invited Paper. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [S.l.: s.n.], 2019. P. 1–7. DOI: 10.1109/ICCAD45719.2019.8942132.

BICHSEL, Benjamin; BAADER, Maximilian; GEHR, Timon; VECHEV, Martin. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In: PROCEEDINGS of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. London, UK: Association for Computing Machinery, 2020. (PLDI 2020), p. 286–300. DOI: 10.1145/3385412.3386007.

BOCHAROV, Alex; ROETTELER, Martin; SVORE, Krysta M. Efficient Synthesis of Universal Repeat-Until-Success Quantum Circuits. **Phys. Rev. Lett.**, American Physical Society, v. 114, p. 080502, 8 Feb. 2015. DOI: `10.1103/PhysRevLett.114.080502`.

CHEN, Zhao-Yun; GUO, Guo-Ping. **QRunes: High-Level Language for Quantum-Classical Hybrid Programming**. [S.l.: s.n.], 2019. arXiv: `1901.08340 [quant-ph]`.

CROSS, Andrew W.; BISHOP, Lev S.; SMOLIN, John A.; GAMBETTA, Jay M. **Open Quantum Assembly Language**. [S.l.: s.n.], 2017. arXiv: `1707.03429 [quant-ph]`.

D-WAVE SYSTEMS INC. **dwavesystems/dwave-ocean-sdk: Installer for D-Wave's Ocean tools.** Available from: `https://github.com/dwavesystems/dwave-ocean-sdk`. Visited on: 12 Aug. 2021.

DEUTSCH, David. Quantum theory, the Church-Turing principle and the universal quantum computer. **Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences**, The Royal Society London, v. 400, n. 1818, p. 97–117, 1985.

DEVELOPERS, Cirq. **Cirq**. [S.l.]: Zenodo, Aug. 2021. DOI: `10.5281/zenodo.5182845`.

DEVITT, Simon J; MUNRO, William J; NEMOTO, Kae. Quantum error correction for beginners. **Reports on Progress in Physics**, IOP Publishing, v. 76, n. 7, p. 076001, June 2013. DOI: `10.1088/0034-4885/76/7/076001`.

DIRAC, P. A. M. A new notation for quantum mechanics. **Mathematical Proceedings of the Cambridge Philosophical Society**, Cambridge University Press, v. 35, n. 3, p. 416–418, 1939. DOI: `10.1017/S0305004100021162`.

FARHI, Edward; GOLDSTONE, Jeffrey; GUTMANN, Sam. **A Quantum Approximate Optimization Algorithm**. [S.l.: s.n.], 2014. arXiv: `1411.4028 [quant-ph]`.

FEITOSA, Samuel S.; VIZZOTTO, Juliana K.; PIVETA, Eduardo K.; DU BOIS, Andre R. FJQuantum – A Quantum Object Oriented Language. **Electronic Notes in Theoretical Computer Science**, v. 324, p. 67–77, 2016. WEIT 2015, the Third Workshop-School on Theoretical Computer Science. ISSN 1571-0661. DOI: `10.1016/j.entcs.2016.09.007`.

FEYNMAN, Richard P. Simulating physics with computers. **International Journal of Theoretical Physics**, v. 21, n. 6, p. 467–488, June 1982. ISSN 1572-9575. DOI: `10.1007/BF02650179`.

FU, X. et al. eQASM: An Executable Quantum Instruction Set Architecture. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). [S.l.: s.n.], 2019. P. 224–237. DOI: `10.1109/HPCA.2019.00040`.

GARHWAL, Sunita; GHORANI, Maryam; AHMAD, Amir. Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages. **Archives of Computational Methods in Engineering**, v. 28, n. 2, p. 289–310, Mar. 2021. ISSN 1886-1784. DOI: `10.1007/s11831-019-09372-6`.

GAY, SIMON J. Quantum programming languages: survey and bibliography. **Mathematical Structures in Computer Science**, Cambridge University Press, v. 16, n. 4, p. 581–600, 2006. DOI: `10.1017/S0960129506005378`.

GIDNEY, Craig; EKERÅ, Martin. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 5, p. 433, Apr. 2021. ISSN 2521-327X. DOI: `10.22331/q-2021-04-15-433`.

GREEN, Alexander S.; LUMSDAINE, Peter LeFanu; ROSS, Neil J.; SELINGER, Peter; VALIRON, Benoît. Quipper: A Scalable Quantum Programming Language. In: PROCEEDINGS of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Seattle, Washington, USA: Association for Computing Machinery, 2013. (PLDI '13), p. 333–342. DOI: `10.1145/2491956.2462177`.

GROVER, Lov K. Quantum Mechanics Helps in Searching for a Needle in a Haystack. **Phys. Rev. Lett.**, American Physical Society, v. 79, p. 325–328, 2 July 1997. DOI: `10.1103/PhysRevLett.79.325`.

HARRIS, Charles R. et al. Array programming with NumPy. **Nature**, v. 585, p. 357–362, 2020. DOI: `10.1038/s41586-020-2649-2`.

HIETALA, Kesha; RAND, Robert; HUNG, Shih-Han; WU, Xiaodi; HICKS, Michael. A Verified Optimizer for Quantum Circuits. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 5, POPL, Jan. 2021. DOI: `10.1145/3434318`.

ITEN, Raban; COLBECK, Roger; KUKULJAN, Ivan; HOME, Jonathan; CHRISTANDL, Matthias. Quantum circuits for isometries. **Phys. Rev. A**, American Physical Society, v. 93, p. 032318, 3 Mar. 2016. DOI: 10.1103/PhysRevA.93.032318.

ITOKO, Toshinari; RAYMOND, Rudy; IMAMICHI, Takashi; MATSUO, Atsushi. Optimization of quantum circuit mapping using gate transformation and commutation. **Integration**, v. 70, p. 43–50, 2020. ISSN 0167-9260. DOI: 10.1016/j.vlsi.2019.10.004.

JAVADIABHARI, Ali; PATIL, Shruti; KUDROW, Daniel; HECKEY, Jeff; LVOV, Alexey; CHONG, Frederic T.; MARTONOSI, Margaret. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In: PROCEEDINGS of the 11th ACM Conference on Computing Frontiers. Cagliari, Italy: Association for Computing Machinery, 2014. (CF '14). DOI: 10.1145/2597917.2597939.

JONES, Tyson; BROWN, Anna; BUSH, Ian; BENJAMIN, Simon C. QuEST and High Performance Simulation of Quantum Computers. **Scientific Reports**, v. 9, n. 1, p. 10736, July 2019. ISSN 2045-2322. DOI: 10.1038/s41598-019-47174-9.

JORDAN, Stephen. **Quantum Algorithm Zoo**. Available from: https://quantumalgorithmzoo.org. Visited on: 16 Aug. 2021.

KHAMMASSI, N.; GUERRESCHI, G. G.; ASHRAF, I.; HOGABOAM, J. W.; ALMUDEVER, C. G.; BERTELS, K. **cQASM v1.0: Towards a Common Quantum Assembly Language**. [S.l.: s.n.], 2018. arXiv: 1805.09607 [quant-ph].

KILLORAN, Nathan; IZAAC, Josh; QUESADA, Nicolás; BERGHOLM, Ville; AMY, Matthew; WEEDBROOK, Christian. Strawberry Fields: A Software Platform for Photonic Quantum Computing. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 3, p. 129, Mar. 2019. ISSN 2521-327X. DOI: 10.22331/q-2019-03-11-129.

KISSINGER, Aleks; WETERING, John van de. PyZX: Large Scale Automated Diagrammatic Reasoning. **Electronic Proceedings in Theoretical Computer Science**, Open Publishing Association, v. 318, p. 229–241, May 2020. ISSN 2075-2180. DOI: 10.4204/eptcs.318.14.

KITAEV, A. Yu. **Quantum measurements and the Abelian Stabilizer Problem**. [S.l.: s.n.], 1995. arXiv: 9511026 [quant-ph].

LIU, Shusen; ZHOU, Li; GUAN, Ji; HE, Yang; DUAN, Runyao; YING, Mingsheng. *Q*|*SI*> : a quantum programming environment. **SCIENTIA SINICA Informationis**, v. 47, n. 10, p. 1300–1315, 2017. DOI: `10.1360/N112017-00095`.

LLVM PROJECT. **LLVM Language Reference Manual**. Available from: `https://llvm.org/docs/LangRef.html`. Visited on: 12 Aug. 2021.

MLNARIK, Hynek. **Operational Semantics and Type Soundness of Quantum Programming Language LanQ**. [S.l.: s.n.], 2007. arXiv: `0708.0890 [quant-ph]`.

MOLINA, Abel; WATROUS, John. Revisiting the simulation of quantum Turing machines by quantum circuits. **Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences**, The Royal Society, v. 475, n. 2226, p. 20180767, June 2019. ISSN 1471-2946. DOI: `10.1098/rspa.2018.0767`.

NAM, Yunseong; ROSS, Neil J.; SU, Yuan; CHILDS, Andrew M.; MASLOV, Dmitri. Automated optimization of large quantum circuits with continuous parameters. **npj Quantum Information**, v. 4, n. 1, p. 23, May 2018. ISSN 2056-6387. DOI: `10.1038/s41534-018-0072-4`.

NIELSEN, Michael A.; CHUANG, Isaac L. Introduction to quantum mechanics. In: QUANTUM Computation and Quantum Information: 10th Anniversary Edition. [S.l.]: Cambridge University Press, 2010a. P. 60–119. DOI: `10.1017/CBO9780511976667.006`.

NIELSEN, Michael A.; CHUANG, Isaac L. Quantum circuits. In: QUANTUM Computation and Quantum Information: 10th Anniversary Edition. [S.l.]: Cambridge University Press, 2010b. P. 171–215. DOI: `10.1017/CBO9780511976667.008`.

ÖMER, Bernhard. **A procedural formalism for quantum computing**. 1998. MA thesis.

ÖMER, Bernhard. Classical Concepts in Quantum Programming. **International Journal of Theoretical Physics**, v. 44, n. 7, p. 943–955, July 2005. ISSN 1572-9575. DOI: `10.1007/s10773-005-7071-x`.

PAKIN, Scott. A quantum macro assembler. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). [S.l.: s.n.], 2016. P. 1–8. DOI: `10.1109/HPEC.2016.7761637`.

PARR, Terence. **The definitive ANTLR 4 reference**. [S.l.]: Pragmatic Bookshelf, 2013.

PAYKIN, Jennifer; RAND, Robert; ZDANCEWIC, Steve. QWIRE: A Core Language for Quantum Circuits. In: PROCEEDINGS of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. Paris, France: Association for Computing Machinery, 2017. (POPL 2017), p. 846–858. DOI: `10.1145/3009837.3009894`.

PIRES, Otto Menegasso; SANTIAGO, Rafael de; MARCHI, Jerusa. Two Stage Quantum Optimization for the School Timetabling Problem. In: 2021 IEEE Congress on Evolutionary Computation (CEC). [S.l.: s.n.], 2021. P. 2347–2353. DOI: `10.1109/CEC45853.2021.9504701`.

PRESKILL, John. **Quantum computing and the entanglement frontier**. [S.l.: s.n.], 2012. arXiv: `1203.5813 [quant-ph]`.

PRESKILL, John. Quantum Computing in the NISQ era and beyond. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 2, p. 79, Aug. 2018. ISSN 2521-327X. DOI: `10.22331/q-2018-08-06-79`.

RISTÈ, D.; BULTINK, C. C.; LEHNERT, K. W.; DICARLO, L. Feedback Control of a Solid-State Qubit Using High-Fidelity Projective Measurement. **Phys. Rev. Lett.**, American Physical Society, v. 109, p. 240502, 24 Dec. 2012. DOI: `10.1103/PhysRevLett.109.240502`.

ROSA, Evandro Chagas Ribeiro da; TAKETANI, Bruno G. **QSystem: bitwise representation for quantum circuit simulations**. [S.l.: s.n.], 2020. arXiv: `2004.03560 [quant-ph]`.

SELINGER, Peter. A Brief Survey of Quantum Programming Languages. In: KAMEYAMA, Yukiyoshi; STUCKEY, Peter J. (Eds.). **Functional and Logic Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 1–6.

SHOR, Peter W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. **SIAM Journal on Computing**, v. 26, n. 5, p. 1484–1509, 1997. DOI: `10.1137/S0097539795293172`.

SINGH, Alexander; GIANNAKIS, Konstantinos; ANDRONIKOS, Theodore. **Qumin, a minimalist quantum programming language**. [S.l.: s.n.], 2017. arXiv: `1704.04460 [cs.PL]`.

SIRAICHI, Marcos Yukio; SANTOS, Vinicius Fernandes dos; COLLANGE, Caroline; PEREIRA, Fernando Magno Quintao. Qubit Allocation. In: PROCEEDINGS of the 2018 International Symposium on Code Generation and Optimization. Vienna, Austria: Association for Computing Machinery, 2018. (CGO 2018), p. 113–125. DOI: 10.1145/3168822.

SMITH, Robert S.; CURTIS, Michael J.; ZENG, William J. **A Practical Quantum Instruction Set Architecture**. [S.l.: s.n.], 2017. arXiv: 1608.03355 [quant-ph].

SOFGE, Donald A. A Survey of Quantum Programming Languages: History, Methods, and Tools. In: SECOND International Conference on Quantum, Nano and Micro Technologies (ICQNM 2008). [S.l.: s.n.], 2008a. P. 66–71. DOI: 10.1109/ICQNM.2008.15.

SOFGE, Donald A. A Survey of Quantum Programming Languages: History, Methods, and Tools. In: SECOND International Conference on Quantum, Nano and Micro Technologies (ICQNM 2008). [S.l.: s.n.], 2008b. P. 66–71. DOI: 10.1109/ICQNM.2008.15.

STEIGER, Damian S.; HÄNER, Thomas; TROYER, Matthias. ProjectQ: an open source software framework for quantum computing. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 2, p. 49, Jan. 2018. ISSN 2521-327X. DOI: 10.22331/q-2018-01-31-49.

STRANO, Daniel; BOLLAY, Benn; NLEWYCKY; BABEJ, Tomas. **vm6502q/qrack: Issue #357 addressed**. [S.l.]: Zenodo, May 2020. DOI: 10.5281/zenodo.3842287.

SVORE, Krysta et al. Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL. In: PROCEEDINGS of the Real World Domain Specific Languages Workshop 2018. Vienna, Austria: Association for Computing Machinery, 2018. (RWDSL2018). DOI: 10.1145/3183895.3183901.

VAN METER, Rodney; ITOH, Kohei M. Fast quantum modular exponentiation. **Phys. Rev. A**, American Physical Society, v. 71, p. 052320, 5 May 2005. DOI: 10.1103/PhysRevA.71.052320.

VARTIAINEN, Juha J.; MÖTTÖNEN, Mikko; SALOMAA, Martti M. Efficient Decomposition of Quantum Gates. **Phys. Rev. Lett.**, American Physical Society, v. 92, p. 177902, 17 Apr. 2004. DOI: 10.1103/PhysRevLett.92.177902.

VIRTANEN, Pauli et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. **Nature Methods**, v. 17, p. 261–272, 2020. DOI: `10.1038/s41592-019-0686-2`.

WECKER, Dave; SVORE, Krysta M. **LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing**. [S.l.: s.n.], 2014. arXiv: `1402.4467 [quant-ph]`.

WOOTTERS, W. K.; ZUREK, W. H. A single quantum cannot be cloned. **Nature**, v. 299, n. 5886, p. 802–803, Oct. 1982. ISSN 1476-4687. DOI: `10.1038/299802a0`.

Appendix

APPENDIX A – PYTHON AST TRANSFORMATION FOR KET

To integrate the `future` variables with the Python statements `if-then-else` and `while`, the Ket interpreter performs transformations on the Python Abstract Syntax Trees (AST) before compiling. In this Appendix, we present how we implement those transformations. In the codes below, we mix the Python syntax with the Python AST structure to make the visualization easy. For information about Python AST, we refer to its online documentation at `https://docs.python.org/3/library/ast.html`.

In the AST transformation, the Ket interpreter only changes the nodes `If`, `While`, `Break`, and `Continue`, with the last two being trivial. Both nodes, `If` and `While`, have the same structure: an expression `test` and two lists of statements, `body` and `orelse`, with the last one representing the scope `else`. In Python, the `else` of a `while` only executes the `test` is false.

A.1 STATEMENT IF-THEN-ELSE

Code A.1.1 shows the `if-then` transformation. First, we assign the `test` to a variable (line 1); and verify if it is an instance of a `future` type as the new test (line 4). For the new `body`, we use labels and branching instructions to implement the statement semantics (lines 6-11). This construction tells the classical computer to place the `if-then` in the quantum code if the `test` is a `future` variable. If it is not, then execute the code as it is (line 13). For the `if-then-else`, the only change is in the `body`, as in Code A.1.2 lines 6-14.

A.2 STATEMENT WHILE-ELSE

Code A.2.1 shows the `while` integration. First, we need to create a new block in the quantum code for the test of the `while` statement. We change the node `While` with an `If` node with the same `test` as for the `if-then` statement (line 6). The `body` of the new `IF` node is the construction of the `while` statement in the quantum computer (lines 8-14), and the `orelse` is the reconstruction of the original code using a `while true` (lines 17-24). We need this reconstruction to keep the semantic of the classical code end do not execute the `test` expression twice. For the `while-else`, it only changes the `body` (Code A.2.2, lines 8-18) and the statement reconstruction (Code A.2.2, lines 21-32). For the `Break` node, the transformation changes it to a `jump` to the `while.end` block; and for the `Continue` node, it changes to a `jump` to the `while.test` block. Code A.2.3 has of how the `break` translates in the quantum code.

Those transformations keep the semantic of the classical code and add the necessary checks to decide when to move the statement to the quantum computer. The only obstacle of this implementation is the overhead added to every `if-then-else`

Code A.1.1 – `if-then` transformation.

```
test_ = node.test
node.test = test_
If(
  test=isinstance(test_, future),
  body= [
    then = label('if.then')
    end  = label('if.end')
    branch(test_, then, end)
    *node.body
    jump(end)
    end.begin()
  ],
  orelse=[node]
)
```

Code A.1.2 – `if-else` transformation.

```
test_ = node.test
node.test = test_
If(
  test=isinstance(test_, future),
  body= [
    then  = label('if.then')
    else_ = label('if.else')
    end   = label('if.end')
    branch(test_, then, else_)
    *node.body
    jump(end)
    else_.begin()
    *node.orelse
    end.begin()
  ],
  orelse=[node]
)
```

and `while` statement. It is possible to mitigate this overhead by adopting Ket as a Python package and using the `code_ket` decorator to transform the AST of some functions only.

Code A.2.1 – `while` transformation.

```
begin = label('while.test')
jump(begin)
begin.begin()
test_ = node.test
If(
  test=isinstance(test_, future),
  body= [
    loop = label('while.loop')
    end  = label('while.end')
    branch(test_, loop, end)
    loop.begin()
    *node.body
    jump(begin)
    end.begin()
  ],
  orelse=[
    If(
      test=test_,
      body=[
        not_done = True
        while not_done:
          *node.body
          if not node.test:
            not_done = False
      ]
    )
  ]
)
```

Code A.2.2 – `while-else` transformation.

```
begin = label('while.test')
jump(begin)
begin.begin()
test_ = node.test
If(
  test=isinstance(test_, future),
  body= [
    loop  = label('while.loop')
    else_ = label('while.else')
    end   = label('while.end')
    branch(test_, loop, else_)
    loop.begin()
    *node.body
    jump(begin)
    else.begin()
    *node.orelse
    jump(end)
    end.begin()
  ],
  orelse=[
    If(
      test=test_,
      body=[
        not_done = True
        while not_done:
          *node.body
          if not node.test:
            not_done = False
        if not not_done:
          *node.oracle
      ],
      orelse=node.orelse
    )
  ]
)
```

Code A.2.3 – Code 5.4.2 using the `break` statement.

```python
q = quant(2)
with quant() as aux:
    while future(True):
        H(q)
        ctrl(q, X, aux)
        res = measure(aux)
        if res == 0:
            break
        X(q+aux)
    aux.free()
print(dump(q).show())
```

```
LABEL @entry
    ALLOC    q0
    ALLOC    q1
    ALLOC    q2
    JUMP     @while.test0
LABEL @while.test0
    INT      i0    1
    BR i0 @while.loop1 @while.end2
LABEL @while.loop1
    H        q0
    H        q1
    CTRL [q0, q1], X   q2
    MEASURE  i1   [q2]
    INT      i2    0
    INT      i3    i1  ==    i2
    BR 3 @if.then3 @if.end4
LABEL @if.then3
    JUMP    @while.end2
LABEL @dead.code5
    JUMP    @if.end4
LABEL @if.end4
    X       q0
    X       q1
    X       q2
    JUMP    @while.test0
LABEL @while.end2
    FREE    q2
    DUMP    [q0, q1]
```

APPENDIX B – MICROSOFT Q# CODING CONTEST WITH KET

In this appendix, We solve problems from the Microsoft Q# Coding Contest Summer 2018 (Chapter B.1), Winter 2019 (Chapter B.2), and Summer 2020 (Chapter B.3) using the quantum programming language Ket. We based all the Ket implementations in the editorials of the events, keeping the same solution adapted for Ket. For each problem, we present the link for its definition in the Codeforces site, which hosted the contests, the Q# reference implementation, and the Ket solution.

In the reference implementations, we kept the comments, updated some deprecated syntax, and omitted the `open` statements. In the Ket solutions, we dropped the `import` statements. In terms of quantum primitives, the Q# and Ket solutions are equivalent, although the Ket implementation is significantly shorter than the Q# implementation due to the classical control. Even though both languages have the same expressive power, the dynamism that Ket inherits from Python allows a more concise quantum program without harm its readability.

B.1 MICROSOFT Q# CODING CONTEST - SUMMER 2018

The Microsoft Q# Coding Contest Summer 2018 editorial at `https://assets.codeforces.com/rounds/997-998/main-contest-editorial.pdf` presents the problems and explained solutions.

Generate superposition of all basis states (A1)

Problem `https://codeforces.com/contest/1002/problem/A1`

Q# Reference Implementation

```
operation Solve (qs : Qubit[]) : Unit {
    for i in 1 .. Length(qs) {
        H(qs[i-1]);
    }
}
```

Ket Implementation

```
def solve(qs : quant):
    H(qs)
```

Generate superposition of zero state and a basis state (A2)

Problem `https://codeforces.com/contest/1002/problem/A2`

Q# Reference Implementation

```
operation Solve (qs : Qubit[], bits : Bool[]) : Unit {
    // Hadamard first qubit
    H(qs[0]);
   // iterate through the bitstring and CNOT to qubits corresponding to true bits
    for i in 1..Length(qs)-1 {
        if (bits[i]) {
            CNOT(qs[0], qs[i]);
        }
    }
}
```

Ket Implementation

```
def solve(qs : quant, bits : List[bool]):
    H(qs[0])
    with control(qs[0]):
        idx = [i for i, b in enumerate(bits) if b and i]
        X(qs.at(idx))
```

Generate superposition of two basis states (A3)

Problem `https://codeforces.com/contest/1002/problem/A3`

Q# Reference Implementation

```
function FindFirstDiff (bits0 : Bool[], bits1 : Bool[]) : Int {
    mutable firstDiff = -1;
    for i in 0 .. Length(bits1)-1 {
        if (bits1[i] != bits0[i] && firstDiff == -1) {
            set firstDiff = i;
        }
    }
    return firstDiff;
}

operation Solve (qs : Qubit[], bits0 : Bool[], bits1 : Bool[]) : Unit {
    // find the index of the first bit at which the bitstrings are different
    let firstDiff = FindFirstDiff(bits0, bits1)
    // Hadamard corresponding qubit to create superposition
    H(qs[firstDiff]);
    // iterate through the bitstrings again setting the final state of qubits
    for i in 0 .. Length(qs)-1 {
        if (bits0[i] == bits1[i]) {
            // if two bits are the same apply X or nothing
            if (bits0[i]) {
                X(qs[i]);
            }
        } else {
            // if two bits are different, set their difference using CNOT
            if (i > firstDiff) {
                CNOT(qs[firstDiff], qs[i]);
                if (bits0[i] != bits0[firstDiff]) {
                    X(qs[i]);
                }
            }
        }
    }
}
```

Ket Implementation

```
def solve(qs : quant, bits0 : List[bool], bits1 : List[bool]):
    diffs = [i for i,j in enumerate(zip(bits0, bits1)) if j[0]!=j[1]]
    H(qs[diffs[0]])
    for i in range(len(qs)):
        if bits0[i] and bits1[i]:
            X(qs[i])
    for i in diffs[1:]:
        ctrl(qs[diffs[0]], X, qs[i])
        if bits0[i] != bits0[diffs[0]]:
            X(qs[i])
```

Generate W state (A4)

Problem `https://codeforces.com/contest/1002/problem/A4`

Q# Reference Implementation

```
operation Solve (qs : Qubit[]) : Unit {
    let N = Length(qs);
    if (N == 1) {
        // base of recursion: |1>
        X(qs[0]);
    } else {
        let K = N / 2;
        // create W state on the first K qubits
        Solve(qs[0..K-1]);
        // the next K qubits are in |0...0> state allocate ancilla in |+> state
        use anc = Qubit[1] {
            let here = anc[0];
            H(here);
            for i in 0..K-1 {
                (Controlled SWAP)([here], (qs[i], qs[i+K]));
            }
            // unentangle here from the rest of the qubits
            for i in K..N-1 {
                CNOT(qs[i], here);
            }
        }
    }
}
```

Ket Implementation

```
def solve(qs : quant):
    n = len(qs)
    if n == 1:
        return X(qs)
    k = n//2
    solve(qs[:k])
    with H(quant()) as aux:
        for i in range(k):
            with control(aux):
                swap(qs[i], qs[i+k])
        for i in range(k):
            cnot(qs[i], aux)
        X(aux).free()
```

Distinguish zero state and W state (B1)

Problem https://codeforces.com/contest/1002/problem/B1

Q# Reference Implementation

```
operation Solve (qs : Qubit[]) : Int {
    // measure all qubits
    mutable countOnes = 0;
    for i in 0..Length(qs)-1 {
        if (M(qs[i]) == One) {
            set countOnes = countOnes + 1;
        }
    }
    // if there is exactly one One, it's W state,
    // if there are no Ones, it’s |0...0>
    if (countOnes == 0) {
        return 0;
    }
    return 1;
}
```

Ket Implementation

```
def solve(qs : quant) -> int:
    return 0 if measure(qs).get() == 0 else 1
```

Distinguish GHZ state and W state (B2)

Problem `https://codeforces.com/contest/1002/problem/B2`

Q# Reference Implementation

```
operation Solve (qs : Qubit[]) : Int {
    // measure all qubits; if there is exactly one One, it’s W state,
    // if there are no Ones or all are Ones, it’s GHZ
    mutable countOnes = 0;
    for i in 0..Length(qs)-1 {
        if (M(qs[i]) == One) {
            set countOnes = countOnes + 1;
        }
    }
    if (countOnes == 1) {
        return 1;
    }
    return 0;
}
```

Ket Implementation

```
def solve(qs : quant) -> int:
    return int(sum(int(i) for i in bin(measure(qs).get())[2:]) == 1)
```

Distinguish four 2-qubit states (B3)

Problem `https://codeforces.com/contest/1002/problem/B3`

Q# Reference Implementation

```
operation Solve (qs : Qubit[]) : Int {
    // These states are produced by H x H, applied to four basis states.
    // To measure them, apply H x H followed by basis state measurement.
    H(qs[0]);
    H(qs[1]);
    return ResultAsInt([M(qs[1]); M(qs[0])]);
}
```

Ket Implementation

```
def solve(qs : quant) -> int:
    return measure(H(qs)).get()
```

Distinguish zero state and plus state with minimum error (C1)

Problem `https://codeforces.com/contest/1002/problem/C1`

Q# Reference Implementation

```
operation Solve (q : Qubit) : Int {
    // Rotate the input state by Pi/8 means to apply Ry with angle 2*Pi/8.
    Ry(0.25*PI(), q);
        if (M(q) == Zero) {
        return 0;
    }
    return 1;
}
```

Ket Implementation

```
def solve(q : quant) -> int:
    return measure(RY(pi/4, q)).get()
```

Distinguish zero state and plus state without errors (C2)

Problem `https://codeforces.com/contest/1002/problem/C2`

Q# Reference Implementation

```
operation Solve (q : Qubit) : Int {
    mutable output = 0;
    let basis = RandomInt(2);
    // randomize over std and had
    if (basis == 0) {
        // use standard basis
        let result = M(q);
        if (result == One) {
            // this can only arise if the state was |+>
            set output = 1;
        }
        else {
            set output = -1;
        }
    }
    else {
        // use Hadamard basis
        H(q);
        let result = M(q);
        if (result == One) {
            // this can only arise if the state was |0>
            set output = 0;
        }
        else {
            set output = -1;
        }
    }
    return output;
}
```

Ket Implementation

```
def solve(q : quant) -> int:
    def _solve(mea, ret):
        return ret if mea(q).get() else -1
    std = (measure, 1)
    had = (lambda q : measure(H(q)), 0)
    return _solve(*choice([std, had]))
```

B.2 MICROSOFT Q# CODING CONTEST - WINTER 2019

The Microsoft Q# Coding Contest Winter 2019 editorial at `https://assets.codeforces.com/rounds/1116/contest-editorial.pdf` presents the problems and explained solutions.

Generate state $|00\rangle + |01\rangle + |10\rangle$ (A1)

Problem `https://codeforces.com/contest/1116/problem/A1`

Q# Reference Implementation

```
operation Solve1 (qs : Qubit[]) : Unit {
    Ry(2.0 * ArcSin(1.0 / Sqrt(3.0)), qs[0]);
    (ControlledOnInt(0, H))([qs[0]], qs[1]);
}

operation Solve2 (qs : Qubit[]) : Unit {
    use ancilla = Qubit() {
        repeat {
            ApplyToEach(H, qs);
            Controlled X(qs, ancilla);
            let res = MResetZ(ancilla);
        }
        until (res == Zero)
        fixup {
            ResetAll(qs);
        }
    }
}
```

Ket Implementation

```
def solve_1(qs : quant):
    RY(2*asin(1/sqrt(3)), qs[0])
    with control(qs[0], on_state=0):
        H(qs[1])

def solve_2(qs : quant):
    with quant() as aux:
        while future(1):
            H(qs)
            ctrl(qs, X, aux)
            res = measure(aux)
            if res == 0:
                break
            X(qs|aux)
        aux.free()
```

Generate equal superposition of four basis states (A2)

Problem `https://codeforces.com/contest/1116/problem/A2`

Q# Reference Implementation

```
operation Solve (qs : Qubit[], bits : Bool[][]) : Unit {
    use anc = Qubit[2] {
        // Put the ancillas into equal superposition of
        // 2-qubit basis states
        ApplyToEach(H, anc);
        // Set up the right pattern on the main qubits
        //cwith control on ancillas
        for i in 0 .. 3 {
            for j in 0 .. Length(qs) - 1 {
                if ((bits[i])[j]) {
                    (ControlledOnInt(i, X))(anc, qs[j]);
                }
            }
        }
        // Uncompute the ancillas, using patterns on main qubits as ontrol
        for i in 0 .. 3 {
            if (i % 2 == 1) {
                (ControlledOnBitString(bits[i], X))(qs, anc[0]);
            }
            if (i / 2 == 1) {
                (ControlledOnBitString(bits[i], X))(qs, anc[1]);
            }
        }
    }
}
```

Ket Implementation

```
def solve(qs : quant, bits : List[int]):
    with quant(2) as aux:
        H(aux)
        for i in range(4):
            for j in range(len(qs)):
                if bits[i][j]:
                    with control(aux, on_state=i):
                        X(qs[j])
        for i in range(4):
            if i % 2:
                with control(qs, on_state=bits[i]):
                    X(aux[1])
            if i // 2:
                with control(qs, on_state=bits[i]):
                    X(aux[0])
        aux.free()
```

Distinguish three-qubit states (B1)

Problem `https://codeforces.com/contest/1116/problem/B1`

Q# Reference Implementation

```
operation WState (qs : Qubit[]) : Unit {
    let N = Length(qs);
    if (N == 1) {
        X(qs[0]);
    } else {
        let theta = ArcSin(1.0 / Sqrt(ToDouble(N)));
        Ry(2.0 * theta, qs[0]);

        (ControlledOnInt(0, WState))(qs[0 .. 0], qs[1 .. N - 1]);
    }
}

operation Solve (qs : Qubit[]) : Int {
    // map the first state to 000 state and the second one
    // to something orthogonal to it
    R1(-2.0 * PI() / 3.0, qs[1]);
    R1(-4.0 * PI() / 3.0, qs[2]);
    Adjoint WState(qs);
    return MeasureInteger(LittleEndian(qs)) == 0 ? 0 | 1;
}
```

Ket Implementation

```
def prepare(qs : quant):
    n = len(qs)
    if n == 1:
        X(qs[0])
    else:
        theta = asin(1.0 / sqrt(n))
        RY(2.0 * theta, qs[0])
        with control(qs[0], on_state=0):
            prepare(qs[1:])

def solve(qs : quant) -> int:
    phase(-2*pi/3, qs[1])
    phase(-4*pi/3, qs[2])
    adj(prepare, qs)
    return 0 if measure(qs).get() == 0 else 1
```

Not A, not B or not C? (B2)

Problem `https://codeforces.com/contest/1116/problem/B2`

Q# Reference Implementation

```
operation Solve (q : Qubit) : Int {
    mutable output = 0;
    let alpha = ArcCos(Sqrt(2.0 / 3.0));
    use a = Qubit() {
        Z(q);
        CNOT(a, q);
        Controlled H([q], a);
        S(a);
        X(q);
        (ControlledOnInt(0, Ry))([a], (-2.0 * alpha, q));
        CNOT(a, q);
        Controlled H([q], a);
        CNOT(a, q);
        // finally, measure in the standard basis
        let res0 = MResetZ(a);
        let res1 = M(q);
        // dispatch on the cases
        if (res0 == Zero && res1 == Zero) {
            set output = 0;
        } elif (res0 == One && res1 == Zero) {
            set output = 1;
        } elif (res0 == Zero && res1 == One) {
            set output = 2;
        } else {
            // this should never occur
            set output = 3;
        }
    }
    return output;
}
```

Ket Implementation

```
def solve(q : quant) -> int:
    alpha = acos(sqrt(2/3))
    with quant() as a:
        Z(q)
        ctrl(a, X, q)
        ctrl(q, H, a)
        S(a)
        X(q)
        with control(a, on_state=0):
            RY(-2*alpha, q)
        with around(ctrl(0, X, 1), a|q):
            ctrl(q, H ,a)
        res0 = measure_free(a)
        res1 = measure(q)
    if res0.get() == res1.get() == 0:
        return 0
    elif res0.get() == 1 and res1.get() == 0:
        return 1
    elif res0.get() == 0 and res1.get() == 1:
        return 2
    else:
        return 3
```

Alternating bits oracle (C1)

Problem https://codeforces.com/contest/1116/problem/C1

Q# Reference Implementation

```
operation FlipAlternating (register : Qubit[], firstIndex : Int) : Unit
    // iterate over elements in every second position,
    // starting with firstIndex
    for i in firstIndex .. 2 .. Length(register) - 1 {
        X(register[i]);
    }
}

operation Solve (x : Qubit[], y : Qubit) : Unit {
    // first mark the state with 1s in even positions,
    // then mark the state with 1s in odd positions
    for firstIndex in 0..1 {
        FlipAlternating(x, firstIndex);
        Controlled X(x, y);
        Adjoint FlipAlternating(x, firstIndex);
    }
}
```

Ket Implementation

```
def solve(x : quant, y : quant):
    for i in range(2):
        with around(X, x[i::2]):
            ctrl(x, X, y)
```

“Is the bit string periodic?” oracle (C2)

Problem `https://codeforces.com/contest/1116/problem/C2`

Q# Reference Implementation

```
operation Evaluate (queryReg : Qubit[], periodAux : Qubit[]) : Unit {
    let N = Length(queryReg);
    for period in 1 .. Length(periodAux) {
        // Evaluate the possibility of the string having period K.
        // For each pair of equal qubits, CNOT the last one into
        // the first one.
        for i in 0..N-period-1 {
            CNOT(queryReg[i + period], queryReg[i]);
        }
        // If all pairs are equal, the first N-K qubits should be
        //  all in state 0.
        (ControlledOnInt(0, X))(queryReg[0..N-period-1], periodAuxperiod-1]);
        // Uncompute
        for i in N-period-1..-1..0 {
            CNOT(queryReg[i + period], queryReg[i]);
        }
    }
}

operation Solve (x : Qubit[], y : Qubit) : Unit {
    // Try all possible periods and see whether any of them produces
    // the necessary string. The result is OR on the period clauses
    let N = Length(x);
    // Valid periods are from 1 to N-1, so N-1 ancillas
    use anc = Qubit[N - 1] {
        Evaluate(x, anc);
        (ControlledOnInt(0, X))(anc, y);
        X(y);
        Adjoint Evaluate(x, anc);
    }
}
```

Ket Implementation

```
def evaluate_periods(query : quant, aux : quant):
    for period in range(1, len(aux)+1):
        cnot_ = lambda q, p: [cnot(q[i+p], q[i]) for i in range(len(q)p)]
        with around(cnot_, query, period):
            with control(query[:len(query)-period], on_state=0):
                X(aux[period-1])

```

```
def solve(x : quant, y : quant):
    with quant(len(x)//2) as aux:
        with around(evaluate_periods, x, aux):
            with control(aux, on_state=0):
                X(y)
            X(y)
        aux.free()
```

“Is the number of ones divisible by 3?” oracle (C3)

Problem https://codeforces.com/contest/1116/problem/C3

Q# Reference Implementation

```
operation AddMod3 (queryReg : Qubit[], ancillaReg : Qubit[]) : Unit
    let sum = ancillaReg[0];
    let carry = ancillaReg[1];
    for q in queryReg {
        // we need to implement addition mod 3:
        // bit sum carry | sum carry
        // 1 0 0 | 1 0
        // 1 1 0 | 0 1
        // 1 0 1 | 0 0
        // compute sum bit
        (ControlledOnBitString([true, false], X))([q, carry], sum);
        // bit sum carry | carry
        // 1 1 0 | 0
        // 1 0 0 | 1
        // 1 0 1 | 0
        (ControlledOnBitString([true, false], X))([q, sum], carry);
    }
}

operation Solve (x : Qubit[], y : Qubit) : Unit {
    // Allocate two ancillas and implement a mini-adder on them:
    // add each qubit to one of the ancillas,
    // using the second one as a "carry".
    // If both qubits end up in 0 state, the number of 1s is
    // divisible by 3.
    use anc = Qubit[2] {
        WithA(AddMod3(x, _), (ControlledOnInt(0, X))(_, y), anc);
    }
}
```

Ket Implementation

```
def add_mod_3(query : quant, aux : quant):
    for bit in query:
        with control(bit, aux[0], on_state=[1, 0]):
            X(aux[1])
        with control(bit, aux[1], on_state=[1, 0]):
            X(aux[0])

def solve(x : quant, y : quant):
    with quant(2) as aux:
        with around(add_mod_3, x, aux):
            with control(aux, on_state=0):
                X(y)
        aux.free()
```

B.3 MICROSOFT Q# CODING CONTEST - SUMMER 2020

The Microsoft Q# Coding Contest Summer 2018 editorial at `https://codeforces.com/blog/entry/79208` presents the problems and explained solutions.

Figure out direction of CNOT (A1)

Problem `https://codeforces.com/contest/1357/problem/A1`

Q# Reference Implementation

```
operation Solve (unitary : (Qubit[] => Unit is Adj+Ctl)) : Int {
    // apply to |01⟩ and measure 1st qubit: CNOT12 will do
    // nothing, CNOT21 will change to |11⟩
    use qs = Qubit[2] {
        within { X(qs[1]); }
        apply { unitary(qs); }
        return MResetZ(qs[0]) == Zero ? 0 | 1;
    }
}
```

Ket Implementation

```
def solve(unitary) -> int:
    q = quant(2)
    X(q[1])
    unitary(q)
    return 0 if measure(q[0]).get() == 0 else 1
```

Distinguish I, CNOTs and SWAP (A2)

Problem https://codeforces.com/contest/1357/problem/A2

Q# Reference Implementation

```
operation Solve (unitary : (Qubit[] => Unit is Adj+Ctl)) : Int {
    // first run: apply to |11⟩; CNOT12 will give |10⟩,
    // CNOT21 will give |01⟩, II and SWAP will remain |11⟩
    use qs = Qubit[2] {
        X(qs[0]);
        X(qs[1]);
        unitary(qs);
        let ind = MeasureInteger(LittleEndian(qs));
        if (ind == 1 or ind == 2) {
            // respective CNOT
            return ind;
        }
    }
    // second run: distinguish II from SWAP, apply to |01⟩:
    // II will remain |01⟩, SWAP will become |10⟩
    use qs = Qubit[2] {
        X(qs[1]);
        unitary(qs);
        let ind = MeasureInteger(LittleEndian(qs));
        return ind == 1 ? 3 | 0;
    }
}
```

Ket Implementation

```
def solve(unitary) -> int:
    with X(quant(2)) as q
        unitary(q)
        m = measure(q.inverted()).get()
    if m == 1 or m == 2:
        return m
    with quant(2) as q:
        X(q[1])
        unitary(q)
        m = measure(q.inverted()).get()
    return 3 if m == 1 else 0
```

Distinguish H from X (A3)

Problem https://codeforces.com/contest/1357/problem/A3

Q# Reference Implementation

```
operation Solve (unitary : (Qubit => Unit is Adj+Ctl)) : Int {
    // apply operation unitary - X - unitary to |0⟩ state and measure:
    // |0⟩ means H, |1⟩ means X
    // X will end up as XXX = X, H will end up as
    // HXH = Z (does not change |0⟩ state)
    use q = Qubit() {
        within {
            unitary(q);
        } apply {
            X(q);
        }
        return MResetZ(q) == Zero ? 0 | 1;
    }
}
```

Ket Implementation

```
def solve(unitary) -> int:
    q = quant(1)
    with around(unitary, q):
        X(q)
    return measure(q).get()
```

Distinguish Rz from R1 (A4)

Problem https://codeforces.com/contest/1357/problem/A4

Q# Reference Implementation

```
operation Solve (unitary : ((Double, Qubit) => Unit is Adj+Ctl)) : Int {
    use qs = Qubit[2] {
        within {
            H(qs[0]);
        } apply {
            Controlled unitary(qs[0..0], (2.0 * PI(), qs[1]));
        }
        return MResetZ(qs[0]) == Zero ? 1 | 0;
    }
}
```

Ket Implementation

```
def solve(unitary) -> int:
    a, b = quant(2)
    with around(H, X(a)):
        ctrl(a, unitary, 2*pi, b)
    return  measure(a).get()
```

Distinguish Rz(θ) from Ry(θ) (A5)

Problem https://codeforces.com/contest/1357/problem/A5

Q# Reference Implementation

```
operation Solve (theta : Double, unitary : (Qubit => Unit is Adj+Ctl))  Int {
    use q = Qubit() {
        let k = Floor(PI() / theta);
        mutable res = 0;
        for rep in 0..10 {
            for i in 1..k {
                unitary(q);
            }
            if (M(q) == One) {
                X(q);
                set res = 1;
            }
        }
        return res;
    }
}
```

Ket Implementation

```
def solve(theta : float, unitary) -> int:
    k = int(pi//theta)
    for _ in range(10):
        q = quant()
        for _ in range(k):
            unitary(q)
        if measure(q).get() == 1:
            return 1
    return 0
```

Distinguish four Pauli gates (A6)

Problem https://codeforces.com/contest/1357/problem/A6

Q# Reference Implementation

```
operation Solve(unitary : (Qubit => Unit is Adj+Ctl)) : Int {
    // apply operation to the 1st qubit of a Bell state and measure in
    // Bell basis
    use qs = Qubit[2] {
        H(qs[0]);
        CNOT(qs[0], qs[1]);
        unitary(qs[0]);
        CNOT(qs[0], qs[1]);
        H(qs[0]);
        // after this I -> 00, X -> 01, Y -> 11, Z -> 10
        let ind = MeasureInteger(LittleEndian(qs));
        let returnValues = [0, 3, 1, 2];
        return returnValues[ind];
    }
}
```

Ket Implementation

```
def solve(unitary) -> int:
    q = quant(2)
    with around(H, q[0]):
        with around(ctrl(0, X, 1), q):
            unitary(q[0])
    return [0, 1, 3, 2][measure(q).get()]
```

Distinguish Y, XZ, -Y and -XZ (A7)

Problem `https://codeforces.com/contest/1357/problem/A7`

Q# Reference Implementation

```
operation Solve(unitary : (Qubit => Unit is Adj+Ctl)) : Int {
    // Run phase estimation on the unitary and the +1 eigenstate of the
    // Y gate |0⟩ + i|1⟩

    // Construct a phase estimation oracle from the unitary
    let oracle = DiscreteOracle(Oracle_Reference(unitary, _, _));

    // Allocate qubits to hold the eigenstate of U and the phase in a
    // big endian register
    mutable phaseInt = 0;
    use (eigenstate, phaseRegister) = (Qubit[1], Qubit[2]) {
        let phaseRegisterBE = BigEndian(phaseRegister);
        // Prepare the eigenstate of U
        H(eigenstate[0]);
        S(eigenstate[0]);
        // Call library
        QuantumPhaseEstimation(oracle, eigenstate, phaseRegisterBE);
        // Read out the phase
        set phaseInt = MeasureInteger(BigEndianAsLittleEndianphaseRegisterBE));
        ResetAll(eigenstate);
        ResetAll(phaseRegister);
    }
    // Convert the measured phase into return value
    return phaseInt;
}
```

Ket Implementation

```
def solve(unitary) -> int:
    target = S(H(quant()))
    n = 5
    m_qubits = quant(n)
    H(m_qubits)
    for i in range(n):
        for _ in range(2**(n-i-1)):
            ctrl(m_qubits[i], unitary, target)
    adj(qft, m_qubits)
    result = measure(m_qubits).get()/2**n
    return {0.0 : 0, 0.75 : 1, 0.5 : 2, 0.25 : 3}[result]
```

"Is the bit string balanced?" oracle (B1)

Problem https://codeforces.com/contest/1357/problem/B1

Q# Reference Implementation

```
operation Solve(inputs : Qubit[], output : Qubit) : Unit is Adj+Ctl {
    let log = BitSizeI(Length(inputs));
    use inc = Qubit[log] {
        within {
            for q in inputs {
                (Controlled Increment)([q], LittleEndian(inc));
            }
        } apply {
            (ControlledOnInt(Length(inputs) / 2, X))(inc, output);
        }
    }
}
```

Ket Implementation

```
def solve(inputs : quant, output : quant):
    def increment(q):
        if len(q) > 1:
            ctrl(q[-1], increment, q[:-1])
        X(q[-1])
    size = ceil(log2(len(inputs)))+1
    ctrl_incr = lambda qs, inc : [ctrl(q, increment, inc) for q in qs]
    with quant(size) as inc:
        with around(ctrl_incr, inputs, inc):
            with control(inc, on_state=len(inputs)//2):
                X(output)
        inc.free()
```

"Is the number divisible by 3?" oracle (B2)

Problem https://codeforces.com/contest/1357/problem/B2

Q# Reference Implementation

```
operation IncrementMod3 (counterRegister : Qubit[]) : Unit is Adj+Ctl {
    let sum = counterRegister[0];
    let carry = counterRegister[1];
    // we need to implement +1 mod 3:
    // sum carry | sum carry
    //  0    0   |  1    0
    //  1    0   |  0    1
    //  0    1   |  0    0
    // compute sum bit
    (ControlledOnInt(0, X))([carry], sum);
    // sum carry | carry
    //  1    0   |   0
    //  0    0   |   1
    //  0    1   |   0
    (ControlledOnInt(0, X))([sum], carry);
}

operation Solve(register : Qubit[], output : Qubit) : Unit is Adj+Ctl {
    use counter = Qubit[2] {
        within {
            for i in 0 .. Length(register) - 1 { // starting from LSB
                if (i % 2 == 0) {
                    // i-th power of 2 is 1 mod 3
                    Controlled IncrementMod3([register[i]], counter);
                } else {
                    // i-th power of 2 is 2 mod 3 - same as -1,
                    // which is Adjoint of +1
                    Controlled Adjoint IncrementMod3([register[i]], ounter);
                }
            }
        } apply {
            // divisible by 3 only if the result is divisible by 3
            (ControlledOnInt(0, X))(counter, output);
        }
    }
}
```

Ket Implementation

```
    def solve(inputs : quant, output : quant):
    def add3_(counter : quant):
        sum = counter[0]
        carry = counter[1]
        with control(carry, on_state=0):
            X(sum)
        with control(sum, on_state=0):
            X(carry)
        add3 = lambda i, c : add3_(c) if i % 2 == 0 else adj(add3_, c)
        cadd3 = lambda q, c : [ctrl(q[i], add3, i, c) for i in range(lenq))]
        with quant(2) as counter:
            with around(cadd3, inputs, counter):
                ctrl(counter, X, output, on_state=0)
            counter.free()
```

Prepare superposition of basis states with 0s (C1)

Problem `https://codeforces.com/contest/1357/problem/C1`

Q# Reference Implementation

```
operation Solve(qs : Qubit[]) : Unit {
    use ancilla = Qubit() {
        repeat {
            // Create equal superposition of all basis states
            ApplyToEach(H, qs);
            // Create (our state) x |0⟩ + |11...11⟩ x |1⟩
            Controlled X(qs, ancilla);
            let res = MResetZ(ancilla);
        }
        until (res == Zero)
        fixup {
            ApplyToEach(X, qs);
        }
    }
}
```

Ket Implementation

```
def solve(qubits : quant):
    with quant() as aux:
        while future(True):
            H(qubits)
            ctrl(qubits, X, aux)
            res = measure(aux)
            if res == 0:
                break
            X(aux|qubits)
        aux.free()
```

Prepare superposition of basis states with the same parity (C2)

Problem `https://codeforces.com/contest/1357/problem/C2`

Q# Reference Implementation

```
operation Solve(qs : Qubit[], parity : Int) : Unit {
    use ancilla = Qubit() {
        // Create equal superposition of all basis states
        ApplyToEach(H, qs);
        // Calculate the parity of states using CNOTs
        ApplyToEach(CNOT(_, ancilla), qs);
        let res = MResetZ(ancilla);
        if ((res == Zero ? 0 | 1) != parity) {
            X(qs[0]);
        }
    }
}
```

Ket Implementation

```
def solve(qubits : quant, parity : int):
    with quant() as aux:
        H(qubits)
        for i in qubits:
            with control(i):
                X(aux)
        res = measure_free(aux)
        if res != parity:
            X(qubits[0])
```