On the Principles of Differentiable Quantum Programming Languages*

Shaopeng Zhu
University of Maryland, College Park, USA

Shih-Han Hung
University of Maryland, College Park, USA

Shouvanik Chakrabarti
University of Maryland, College Park, USA

Xiaodi Wu
University of Maryland, College Park, USA

Abstract

Variational Quantum Circuits (VQCs), or the so-called quantum neural-networks, are predicted to be one of the most important near-term quantum applications, not only because of their similar promises as classical neural-networks, but also because of their feasibility on near-term noisy intermediate-size quantum (NISQ) machines. The need for gradient information in the training procedure of VQC applications has stimulated the development of auto-differentiation techniques for quantum circuits. We propose the first formalization of this technique, not only in the context of quantum circuits but also for imperative quantum programs (e.g., with controls), inspired by the success of differentiable programming languages in classical machine learning. In particular, we overcome a few unique difficulties caused by exotic quantum features (such as quantum no-cloning) and provide a rigorous formulation of differentiation applied to bounded-loop imperative quantum programs, its code-transformation rules, as well as a sound logic to reason about their correctness. Moreover, we have implemented our code transformation in OCaml and demonstrated the resource-efficiency of our scheme both analytically and empirically. We also conduct a case study of training a VQC instance with controls, which shows the advantage of our scheme over existing auto-differentiation for quantum circuits without controls.

*This work was partially funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, Quantum Algorithms Team program, the U.S. Air Force Office of Scientific Research MURI grant FA9550-16-1-0082, and the U.S. National Science Foundation grant CCF-1755800, CCF-1816695, and CCF-1942837(CAREER).

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386011

CCS Concepts: • **Theory of computation → Quantum information theory**; **Denotational semantics**; **Operational semantics**; • **Computing methodologies → Machine learning**.

Keywords: quantum programming languages, differentiable programming languages, quantum machine learning

ACM Reference Format:
Shaopeng Zhu, Shih-Han Hung, Shouvanik Chakrabarti, and Xiaodi Wu. 2020. On the Principles of Differentiable Quantum Programming Languages. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3386011

1 Introduction

Background. Recent years have witnessed the rapid development of quantum computing, with practical advances coming from both research and industry. Quantum programming is one topic that has been actively investigated. Early work on language design [23, 33, 40, 41, 45] has been followed up recently by several implementations of these languages, including Quipper [25], Scaffold [2], LIQUi|⟩ [48], Q# [47], and QWIRE [34]. Extensions of program logics have also been proposed for verification of quantum programs [4, 10, 11, 19, 28, 29, 51, 53]. See also surveys [20, 44, 52].

With the availability of prototypes of quantum machines, especially the recent establishment of quantum supremacy [3], the research of quantum computing has entered a new stage where near-term Noisy Intermediate-Scale Quantum (NISQ) computers [38], e.g., the 53-qubit quantum machines from Google [3] and IBM [22], become the important platform for demonstrating quantum applications. Variational quantum circuits (VQCs) [17, 18, 36], or the so-called *quantum neural networks*, are predicted to be one of the most important applications on NISQ machines. It is not only because VQCs bear a lot of similar promises like classical neural networks as well as potential quantum speed-ups from the perspective of machine learning (e.g., see the survey [9]), but also because VQC is, if not the only, one of the few candidates that can be implemented on NISQ machines. Because of this, a lot of study has already been devoted to the design, analysis, and small-scale implementation of VQCs (e.g., see the survey [7]).

Typical VQC applications replace classical neural networks, which are just parameterized classical circuits, by quantum circuits with *classically parameterized unitary gates*. Namely, one will have a "quantum" mapping from input to output replacing classical mapping in machine learning applications. An important component of these applications is a training procedure which optimizes a *loss function* that now depends on the *read-outs* and the *parameters* of VQCs.

Gradient-based approaches are widely used in the training procedure. However, computing these gradients of loss functions from quantum circuits has a similar complexity of simulating quantum circuits, which is infeasible for classical computation. Thus, the ability of evaluating these "quantum" gradients efficiently by quantum computation is critical for the scalability of VQC applications.

Fortunately, analytical formulas of gradients used in VQCs have been studied by [18, 21, 31, 42, 43]. In particular, Schuld et al. [42] proposed the so-called *phase-shift rule* that uses two quantum circuits to compute the partial derivative respective to one parameter for quantum circuits. One of the very successful tools for quantum machine learning, called PennyLane [8], implemented the phase-shift rule to achieve *auto differentiation* (AD) for the read-outs of quantum circuits. However, none of these studies was conducted from the perspective of programming languages and no rigorous foundation or principles have been formalized.

Motivations. An important motivation of this paper is to provide a *rigorous formalization* of the auto-differentiation technique applied to quantum circuits. In particular, we will provide a formal formulation of quantum programs, their semantics, and the meaning of differentiation of them. We will also study the code-transformation rules for auto-differentiation and prove their correctness.

As we will highlight below, research on the formalization will encounter many new challenges that have not been considered or addressed by existing results [18, 21, 31, 42, 43]. Consider one of the basic requirements, e.g., *compositionality*. As we will show, differentiating the composition of quantum programs will necessarily involve running multiple quantum programs on copies of initial quantum states. How to represent the collection of quantum programs succinctly and also bound the number of required copies is a totally new question. Among our techniques to address this question, we also need to change the previously proposed construct, e.g., the phase-shift rule [42], to something different.

Moreover, we want to go beyond the restriction of quantum circuits. Our inspiration comes from classical machine learning examples that demonstrate the advantage of neural-networks with program features (e.g. controls) over the plain ones (e.g., classical circuits), e.g. [24, 26], which is also the major motivation of promoting the the paradigm shift from deep learning toward *differentiable programming*.

Augmenting VQCs with controls, at least for simple ones, is not only feasible on NISQ machines, but also a logical step for the study of their applications in machine learning. Therefore, we are inspired to investigate the ***principles of differentiable quantum imperative languages*** beyond circuits. Indeed, we conduct one such case study in Section 8.

Research Challenges & Solutions. We will rely on a few notations that should be self-explanatory. Please refer to a detailed preliminary on quantum information in Section 2. Let us start with a simple classical program

$$\mathrm{MUL} \equiv v_3 = v_1 \times v_2, \tag{1.1}$$

where v_3 is the product of v_1 and v_2. Consider the differentiation with respect to θ, we have

$$\frac{\partial}{\partial\theta}(\mathrm{MUL}) \equiv \quad v_3 = v_1 \times v_2; \tag{1.2}$$

$$\dot{v}_3 = \dot{v}_1 \times v_2 + v_1 \times \dot{v}_2, \tag{1.3}$$

where MUL keeps track of variables v_1, v_2, v_3 and their derivatives $\dot{v}_1, \dot{v}_2, \dot{v}_3$ at the same time. One simple yet important observation is that classical variables v_1, v_2, v_3 are real-valued and can be naturally differentiated.

Given that quantum states are represented by matrices, what are the natural quantities to differentiate in the quantum setting? One natural choice from the principles of quantum mechanics is the (classical) read-outs of quantum systems through measurements, which we formulate as the *observable semantics* of quantum programs. This natural choice also serves the purpose of gradient computation of loss functions in quantum machine learning, which are typically defined in terms of these read-outs. We directly model the parameterization of quantum programs after VQCs, i.e., each unitary gate becomes *classically parameterized*. [1]

To model the meaning of one quantum program computing the derivative of another, we define the *differential semantics* of programs. There is a subtle quantum-unique design choice. The observable semantics of any quantum program will depend on the observable and its input state. Thus, any program computing its derivative could potentially depend on these two extra factors. We find out this potential dependence is undesirable and propose the strongest possible definition: i.e., one derivative computing program should work for *any pair* of observables and input states. We demonstrate that this strong requirement is not only achievable but also critical for the composition of auto differentiation.

We are ready to describe the technical challenges for the compositionality. Consider the following quantum program:

$$\mathrm{QMUL} \equiv U_1(\theta); U_2(\theta), \tag{1.4}$$

which performs $U_1(\theta)$ and $U_2(\theta)$ gates sequentially. Note that gate application is matrix multiplication in the quantum setting. Roughly speaking, if the product rule of differentiation

[1] The above modeling of quantum programs is very different from classical ones. It is unclear whether any reasonable analogue of classical chain-rule and forward/backward mode can exist within quantum programs.

(as exhibited in (1.3)) remains in the quantum setting, at least symbolically, then one should expect $\frac{\partial}{\partial\theta}(\text{QMUL})$ contains

$$\frac{\partial}{\partial\theta}(U_1(\theta)); U_2(\theta) \text{ and } U_1(\theta); \frac{\partial}{\partial\theta}(U_2(\theta)) \tag{1.5}$$

two different parts as sub-programs similarly in (1.3).

However, we cannot run $\frac{\partial}{\partial\theta}(U_1(\theta)); U_2$ and $U_1(\theta); \frac{\partial}{\partial\theta}(U_2(\theta))$ together due to the quantum *no-cloning* theorem [50]. It is simply because they share the same initial state and we cannot clone two copies of it. This is not an issue classically as we can store all $v_i, \dot{v}_i$ at the same time as in (1.3). As a result, quantum differentiation needs to run *multiple* (sub-)programs on *multiple* copies of the initial state.

This change poses a unique challenge for differentiation of quantum composition: (1) we hope to have a simple scheme of code transformation, ideally close-to-classical, for intuition and easy implementation of the compiler, whereas it needs to express correctly the collection of quantum programs during code transformation; (2) for the purpose of efficiency, we also want to reasonably bound the number of required copies of the initial states, which roughly refers to the number of different quantum programs in the collection.

We develop a few techniques to achieve both goals at the same time. First, we propose the so-called *additive* quantum programs as a *succinct* intermediate representation for the collection of programs during the code transformation. Now the entire differentiation procedure will be divided into two steps: (1) all code transformations happen on additive programs and are very similar to classical ones (see Figure 4) ; (2) the collection of programs can be recovered by a compilation procedure from any additive program. Additive quantum programs are equipped with a new *sum* operation that models the multiple choices as exhibited in (1.5), which resembles a similar idea in the differential lambda-calculus [14].

Second, we also design a new rule for $\frac{\partial}{\partial\theta}(U(\boldsymbol{\theta}))$ which is slightly different from [18, 21, 31, 42, 43]. The existing phase-shift rule makes use of two quantum circuits for one differentiation, which causes a lot of inconvenience in the formulation and potential trouble for efficiency. Instead, we use only one extra ancilla as the control qubit to create a superposition of two quantum circuits and effectively achieve the same differentiation with only one quantum circuit. We also conduct a careful resource analysis of our differentiation procedure and show the number of required copies of initial states is reasonable comparing to the classical setting. The correctness of the code transformation of composition critically relies on our design choice as well as the strong definition related to the differential semantics.

With the previous setup, we can naturally build the differentiation for quantum controls (i.e., the condition statement). Note that a general solution for classical controls is unknown [6] due to the non-smoothness of the guard. Similar to the classical setting [37], we only provide a solution to deal with bounded loops and leave it open for general ones.

Contributions. We formulate the parameterized quantum bounded while-programs with classically parameterized unitary gates modelled after VQCs [17, 32, 36] and their realistic examples on ion-trap machines, e.g. [54], in Section 3.

In Section 4, we illustrate our design of *additive* quantum programs. Specifically, we add the syntax $P_1 + P_2$ to represent the either-or choice between P_1 and P_2 in (1.5). We formulate its semantics and compilation rules that map additive programs into collections of normal ones for our purpose.

In Section 5, we formulate the observable and the differential semantics of quantum programs and formally define the meaning of program $S'(\boldsymbol{\theta})$ computing the differential semantics of $S(\boldsymbol{\theta})$ in the strongest possible sense.

In Section 6, we show that such a strong requirement is indeed achievable by demonstrating the code-transformation rules for the differentiation procedure. Thanks to the use of additive quantum programs, the code transformation is much simplified and as intuitive as classical ones. We develop a logic with the judgement $S'(\boldsymbol{\theta})|S(\boldsymbol{\theta})$ stating that $S'(\boldsymbol{\theta})$ computes the differential semantics of $S(\boldsymbol{\theta})$. We prove it sound and use it to show the correctness of the code transformation.

In Section 7, we conduct a resource analysis to further justify our design. We show that the *occurrence count* of parameters capture the extra resource required in both the classical auto-differentiation and our scheme. Hence, our resource cost is reasonable compared with the classical setting.

Finally, in Section 8, we demonstrate the implementation of our code transformation in OCaml and apply it to the training of one VQC instance with controls via classical simulation. Specifically, this instance shows an advantage of controls in machine learning tasks, which implies the advantage of our scheme over previous ones that cannot handle controls. We have also empirically verified the resource-efficiency of our scheme on representative VQC instances.

Related Classical Work. There is an extensive study of automatic differentiation (AD) or differentiable programming in the classical setting (e.g., see books [12, 27]). The most relevant to us are those studies from the programming language perspective. AD has traditionally been applied to imperative programs in both the forward mode, e.g. [30, 49], and the reverse mode, e.g., [46]. The famous *backpropagation* algorithm [39] is also a special case of reserve-mode AD used to compute the gradient of a multi-layer perceptron. AD has also been recently applied to functional programs [15, 16, 35]. Motivated by the success of deep learning, there is significant recent interest to develop both the theory and the implementation of AD techniques. Please refer to the survey [5] and the keynote talk at POPL'18 [37] and [1] for more details.

2 Quantum Preliminaries

We present basic quantum preliminaries (a summary of notation in Table 1). Details are deferred to the full version [55].

Table 1. A brief summary of notation used in this paper

Spaces	$\mathcal{H}, \mathcal{A}$	$L(\mathcal{H})$ (Linear operators)
States	(pure states)	$\|\psi\rangle, \|\phi\rangle$; $\|0\rangle, \|1\rangle, \|+\rangle, \|-\rangle$
	(density)	ρ, σ; $\|\psi\rangle\langle\psi\|$
Operations	(unitaries)	$U, V, \boldsymbol{\sigma}$; $H, X, Z, X \otimes X$
	(superoperators)	$\mathcal{E}, \mathcal{F}$ (general); Φ (quantum channels)
Measurements	M	$\{M_m\}_m$; $\{\|0\rangle\langle 0\|, \|1\rangle\langle 1\|\}$ (example)
Observables	O	$\sum_m \lambda_m \|\phi_m\rangle\langle\phi_m\|$; $\|0\rangle\langle 0\| - \|1\rangle\langle 1\|$ (example)
Programs	(no parameters)	P, Q
	(parameterized)	$P(\boldsymbol{\theta}), S(\boldsymbol{\theta})$; $R'_{\boldsymbol{\sigma}}(\theta)$(notable examples)
	(additive)	$P(\boldsymbol{\theta}), \overline{S(\boldsymbol{\theta})}$; $\overline{\frac{\partial}{\partial\theta}(S(\boldsymbol{\theta}))}$ (example)
Semantics	(operational)	$\langle P, \rho\rangle \to \langle Q, \rho'\rangle$ (steps)
	(denotational)	$[\![P]\!]\rho = \rho'$
	(observable)	$[\![(O, \rho) \to P(\boldsymbol{\theta})]\!]$
Code Process	(Transform)	$\frac{\partial}{\partial\theta}(\overline{S(\boldsymbol{\theta})})$ (example)
	(Compile)	$\texttt{Compile}(\overline{S'(\boldsymbol{\theta})})$ (example)
Logic	(Judgement)	$\overline{S'(\boldsymbol{\theta})} \| \overline{S(\boldsymbol{\theta})}$ (example)
Count	(Non-Abort)	$\|\#\overline{\frac{\partial}{\partial\theta_j}(P(\boldsymbol{\theta}))}\|$ (example)
	(Occurrence)	$\mathrm{OC}_j(P(\boldsymbol{\theta}))$ (example)

2.1 Math Preliminaries

Let n be a natural number. We refer to the complex vector space $\mathbb{C}^n$ as an *n-dimensional Hilbert space* $\mathcal{H}$. We use $|\psi\rangle$ to denote a complex vector in $\mathbb{C}^n$. The Hermitian conjugate of $|\psi\rangle \in \mathbb{C}^n$ is denoted by $\langle\psi|$. The *inner product* of $|\psi\rangle$ and $|\phi\rangle$, defined as the product of $\langle\psi|$ and $|\phi\rangle$, is denoted by $\langle\psi|\phi\rangle$. The norm of a vector $|\psi\rangle$ is denoted by $\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle}$.

We define *operators* as linear maps between Hilbert spaces, which can be represented by matrices for finite dimensions. Let A be an operator and its Hermitian conjugate $A^\dagger$. A is *Hermitian* if $A = A^\dagger$. The *trace* of A is the sum of the entries on the main diagonal, i.e., $\mathrm{tr}(A) = \sum_i A_{ii}$. $\langle\psi|A|\psi\rangle$ denotes the inner product of $|\psi\rangle$ and $A|\psi\rangle$. Hermitian operator A is *positive semidefinite* if for all vectors $|\psi\rangle \in \mathcal{H}$, $\langle\psi|A|\psi\rangle \geq 0$.

2.2 Quantum States and Operations

The state space of a *qubit* is a 2-dimensional Hilbert space. Two important orthonormal bases of a qubit system are: the *computational* basis with $|0\rangle = (1,0)^\dagger$ and $|1\rangle = (0,1)^\dagger$; the $\pm$ basis, consisting of $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

A *pure* quantum state is a unit vector $|\psi\rangle$. A *mixed* state, which refers to an ensemble of pure states $\{|\psi_i\rangle\}_i$ each with probability p_i, can be represented by a *density operator* that is a trace-one positive semidefinite operator $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$; ρ is a *partial* density operator if $\mathrm{tr}(\rho) \leq 1$. The set of partial density operators on $\mathcal{H}$ is denoted by $\mathcal{D}(\mathcal{H})$.

Operations on quantum systems can be characterized by unitary operators. Denoting the set of linear operators on $\mathcal{H}$ as $L(\mathcal{H})$, an operator $U \in L(\mathcal{H})$ is *unitary* if $U^\dagger U = UU^\dagger = I_{\mathcal{H}}$. A unitary *evolves* a pure state $|\psi\rangle$ to $U|\psi\rangle$, or a density operator ρ to $U\rho U^\dagger$. Common unitary operators include: the *Hadamard* operator H, which transforms between the computational and the $\pm$ basis via $H|0\rangle = |+\rangle$ and $H|1\rangle = |-\rangle$; the *Pauli X* operator which performs a bit flip, i.e., $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$; *Pauli Z* which performs a phase flip, i.e., $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$; *CNOT* gate mapping $|00\rangle \mapsto |00\rangle, |01\rangle \mapsto |01\rangle, |10\rangle \mapsto |11\rangle, |11\rangle \mapsto |10\rangle$. More generally, evolution of a quantum system can be characterized by an *admissible superoperator* $\mathcal{E}$, namely a *completely-positive* and *trace-non-increasing* linear map from $\mathcal{D}(\mathcal{H})$ to $\mathcal{D}(\mathcal{H}')$.

For every superoperator $\mathcal{E}$, there exists a set of *Kraus operators* $\{E_k\}_k$ such that $\mathcal{E}(\rho) = \sum_k E_k \rho E_k^\dagger$ for any input $\rho \in \mathcal{D}(\mathcal{H})$. The *Kraus form* of $\mathcal{E}$ is therefore $\mathcal{E} = \sum_k E_k \circ E_k^\dagger$. The Schrödinger-Heisenberg *dual* of a superoperator $\mathcal{E} = \sum_k E_k \circ E_k^\dagger$, denoted by $\mathcal{E}^*$, is defined as follows: for every state $\rho \in \mathcal{D}(\mathcal{H})$ and any operator A, $\mathrm{tr}(A\mathcal{E}(\rho)) = \mathrm{tr}(\mathcal{E}^*(A)\rho)$. The Kraus form of $\mathcal{E}^*$ is $\sum_k E_k^\dagger \circ E_k$.

2.3 Quantum Measurements

Quantum *measurements* extracts classical information out of quantum systems. A quantum measurement on a system over Hilbert space $\mathcal{H}$ can be described by a set of linear operators $\{M_m\}_m$ with $\sum_m M_m^\dagger M_m = I_{\mathcal{H}}$ (identity matrix on $\mathcal{H}$). If we perform a measurement $\{M_m\}_m$ on a state ρ, the outcome m is observed with probability $p_m = \mathrm{tr}(M_m \rho M_m^\dagger)$ for each m, and the post-measurement state collapses to $M_m \rho M_m^\dagger / p_m$.

3 Parameterized Quantum Bounded While-Programs

We adopt the *bounded-loop* variant of the quantum **while**-language developed by Ying [52], and augment it by parameterizing the unitaries, as this provides sufficient expressibility for parameterized quantum operations: indeed, **abort**, **skip** and initialization behave independently of parameters, while "parameterized measurements" can be implemented with a regular measurement followed by a parameterized unitary.

From here onward, $\overline{v}$ is a finite set of variables, and $\boldsymbol{\theta}$ a length-k vector of real-valued parameters.

3.1 Syntax

Define *Var* as the set of quantum variables. We use the symbol q as a metavariable ranging over quantum variables and define a *quantum register* $\overline{q}$ to be a finite set of distinct variables. For each $q \in Var$, its state space is denoted by $\mathcal{H}_q$. The quantum register $\overline{q}$ is associated with the Hilbert space

$\mathcal{H}_{\overline{q}} = \bigotimes_{q\in\overline{q}} \mathcal{H}_q$. [2] A *$T$-bounded, k-parameterized quantum **while**-program* is generated by the following syntax:

$$\begin{aligned} P(\boldsymbol{\theta}) \;::=\;& \mathbf{abort}[\overline{q}] \mid \mathbf{skip}[\overline{q}] \mid q := |0\rangle \mid \overline{q} := U(\boldsymbol{\theta})[\overline{q}] \mid \\ & P_1(\boldsymbol{\theta}); P_2(\boldsymbol{\theta}) \mid \mathbf{case}\ M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta})}\ \mathbf{end} \mid \\ & \mathbf{while}^{(T)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta})\ \mathbf{done}, \end{aligned}$$

where

$$\begin{aligned} & \mathbf{while}^{(1)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta})\ \mathbf{done} \\ \equiv\ & \mathbf{case}\ M[\overline{q}] = \{0 \to \mathbf{skip},\ \ 1 \to P_1(\boldsymbol{\theta}); \mathbf{abort}\}, \end{aligned} \tag{3.1}$$

$$\begin{aligned} & \mathbf{while}^{(T\geq 2)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta})\ \mathbf{done} \\ \equiv\ & \mathbf{case}\ M[\overline{q}] = \{0 \to \mathbf{skip},\ \ 1 \to P_1(\boldsymbol{\theta}); \mathbf{while}^{(T-1)}\} \end{aligned}$$

Unparameterized programs can be obtained by fixing $\boldsymbol{\theta}^* \in \mathbb{R}^k$ in some $P(\boldsymbol{\theta})$. We denote the set of variables accessible to $P(\boldsymbol{\theta})$ as $\mathrm{qVar}(P(\boldsymbol{\theta}))$; the collection of all "T-bounded while-programs $P(\boldsymbol{\theta})$ s.t. $\mathrm{qVar}(P(\boldsymbol{\theta})) = \overline{v}$" as $\mathbf{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, and similarly, the unparameterized one as $\mathbf{q\text{-}while}^{(T)}_{\overline{v}}$.

Now let us formally define *parameteriztion* of unitaries: let $\boldsymbol{\theta} := (\theta_1, \cdots, \theta_k)$ $(k \geq 1)$. A *k-parameterized unitary* $U(\boldsymbol{\theta})$ is a function $\mathbb{R}^k \to L(\mathcal{H})$ s.t. (1) given any $\boldsymbol{\theta}^* \in \mathbb{R}^k$, $U(\boldsymbol{\theta}^*)$ is an unitary on $\mathcal{H}$, and (2) the parameterized-matrix representation of $U(\boldsymbol{\theta})$ is entry-wise smooth.

A important family is the *single-qubit rotations* about the Pauli axis X, Y, Z with angle θ (matrix exponential here):

$$R_{\boldsymbol{\sigma}}(\theta) := \exp\left(\frac{-i\theta}{2}\boldsymbol{\sigma}\right), \boldsymbol{\sigma} \in \{X, Y, Z\}\ . \tag{3.2}$$

One can also extend Pauli rotations to multiple qubits. For example, consider *two-qubit coupling* gates $\{R_{\sigma\otimes\sigma} := \exp(\frac{-i\theta}{2}\sigma\otimes\sigma)\}_{\sigma\in\{X,Y,Z\}}$, which generate entanglement between two qubits. Combined with single-qubit rotations, they form a *universal gate set* for quantum computation. Another important feature is that they can already be reliably implemented in such as ion-trap quantum computers [54].

As a result, we will work mostly with these gates in the rest of this paper. However, note that one can easily add and study other parameterized gates in our framework as well.

The language constructed above is similar to their classical counterparts. (0) **abort** terminates the program, outputting $\mathbf{0} \in \mathcal{D}(\mathcal{H}_{\overline{q}})$. (1) **skip** does nothing to states in $\mathcal{D}(\mathcal{H}_{\overline{q}})$. (2) $q := |0\rangle$ sets quantum variable q to the basis state $|0\rangle$. The underlying quantum procedure is to apply super-operators $\mathcal{E}^{\mathrm{bool}}_{q\to 0}(\cdot)$ (or $\mathcal{E}^{\mathrm{int}}_{q\to 0}(\cdot)$)[3] to q and identity operations to the rest of variables. The correlation between q and the rest of quantum variables could be potentially disturbed. (3) for any $\boldsymbol{\theta}^* \in \mathbb{R}^k$, $\overline{q} := U(\boldsymbol{\theta}^*)[\overline{q}]$ applies the unitary $U(\boldsymbol{\theta}^*)$ to the qubits in $\overline{q}$. (4) Sequencing has the same behavior as its classical counterpart. (5) for $\boldsymbol{\theta}^* \in \mathbb{R}^k$, **case** $M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta}^*)}$ **end** performs the measurement $M = \{M_m\}$ on the qubits in $\overline{q}$, and executes program $P_m(\boldsymbol{\theta}^*)$ if the outcome of the measurement is m. The bar over $\overline{m \to P_m}$ indicates that there may be one or more repetitions of this expression. (6) $\mathbf{while}^{(T)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta}^*)\ \mathbf{done}$ performs the measurement $M = \{M_0, M_1\}$ on $\overline{q}$, and terminates if the outcome corresponds to M_0, or executes $P_1(\boldsymbol{\theta}^*)$ then reiterates ($T \geq 2$) / aborts ($T = 1$) otherwise. The program iterates at most T times.

We highlight two differences between quantum and classical while languages: (1) Qubits may only be initialized to the state $|0\rangle$. There is no quantum analogue for initialization to any expression (i.e. $x := e$) due to the no-cloning theorem of quantum states. Any state $|\psi\rangle \in \mathcal{H}_q$, however, can be constructed by applying some unitary U to $|0\rangle$. (2) Evaluating the guard of a case statement or loop, which performs a measurement, potentially disturbs the state of the system.

3.2 Operational and Denotational Semantics

We present the *operational semantics* of parameterized programs in Figure 1a. Transition rules are represented as $\langle P,\ \rho\rangle \to \langle P',\ \rho'\rangle$, where $\langle P,\ \rho\rangle$ and $\langle P',\ \rho'\rangle$ are quantum *configurations*.[4] In configurations, P (or P') could be a quantum program or the empty program $\downarrow$, and ρ and ρ' are partial density operators representing the current state. Intuitively, in one step, we can evaluate program P on input state ρ to program P' (or $\downarrow$) and output state ρ'. In order to present the rules in a non-probabilistic manner, the probabilities associated with each transition are encoded in the output partial density operator. For each index m of branches in a loop/control statement, the superoperator $\mathcal{E}_m$ is defined by $\mathcal{E}_m(\rho) = M_m \rho M_m^\dagger$, yielding the post-measurement state.

We present the *denotational semantics* of parameterized programs in 1b, defining $[\![P]\!]$ as a superoperator on $\rho \in \mathcal{H}_{\overline{v}}$ [52]. For more details we refer the reader to Ying [51, 52].

We have the following connection between the denotational semantics and operational for parameterized programs: in short, the meaning of running program $P(\boldsymbol{\theta}^*)$ on input state ρ and any $\boldsymbol{\theta}^* \in \mathbb{R}^k$ is the sum of all possible output states with multiplicity, weighted by their probabilities.

Proposition 3.1 ([52]). *$\forall P(\boldsymbol{\theta}) \in \mathbf{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, and any specific $\boldsymbol{\theta}^* \in \mathbb{R}^k$, $\rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$,*

$$[\![P(\boldsymbol{\theta}^*)]\!](\rho) = \sum \{|\rho' : (P(\boldsymbol{\theta}^*), \rho) \to^* (\downarrow, \rho')|\}. \tag{3.3}$$

*Here $\to^*$ is the reflexive, transitive closure of $\to$ and $\{|\cdot|\}$ denotes a **multi-set**.*

We close the section with a notion arising from the following observation: some programs, while syntactically not "$\mathbf{abort}[\overline{q}]$", semantically aborts. Simple examples include $U(\boldsymbol{\theta})$; **abort** or a case sentence that has **abort** on each branch.

[2] If $\mathrm{type}(q) = \mathbf{Bool}$ then $\mathcal{H}_q = \mathrm{span}\{|0\rangle, |1\rangle\}$. If $\mathrm{type}(q) = \mathbf{Bounded\ Int}$ then $\mathcal{H}_q$ is with basis $\{|n\rangle : n \in [-N, N]\}$ $(N \in \mathbb{Z}^+)$ for some finite N. We require the Hilbert space to be finite dimensional for implementation.

[3] $\mathcal{E}^{\mathrm{bool}}_{q\to 0}(\rho) = |0\rangle_q\langle 0|\rho|0\rangle_q\langle 0| + |0\rangle_q\langle 1|\rho|1\rangle_q\langle 0|$ and $\mathcal{E}^{\mathrm{B-int}}_{q\to 0}(\rho) = \sum_{n=-N}^{N} |0\rangle_q\langle n|\rho|n\rangle_q\langle 0|$ $(N \in \mathbb{Z}^+)$.

[4] Recall that, fixing arbitrary $\boldsymbol{\theta}^* \in \mathbb{R}^k$, both semantics reduce to those of unparameterized programs, so for compactness we write P for $P(\boldsymbol{\theta}^*)$, etc.

$$\text{(Abort)}\ \overline{\langle \mathbf{abort}[\overline{q}],\ \rho\rangle \to \langle\downarrow,\ \mathbf{0}\rangle}$$

$$\text{(Skip)}\ \overline{\langle \mathbf{skip}[\overline{q}],\ \rho\rangle \to \langle\downarrow,\ \rho\rangle}$$

$$\text{(Init)}\ \overline{\langle q := |0\rangle,\ \rho\rangle \to \langle\downarrow,\ \rho_0^q\rangle}$$

$$\text{where } \rho_0^q = \begin{cases} \mathcal{E}^{\text{bool}}_{q\to 0}(\rho) & \text{if } type(q) = \mathbf{Bool} \\ \mathcal{E}^{\text{B-int}}_{q\to 0}(\rho) & \text{if } type(q) = \mathbf{Bdd\ Int} \end{cases}$$

$$\text{(Unitary)}\ \overline{\langle \overline{q} := U(\boldsymbol{\theta}^*)[\overline{q}],\ \rho\rangle \to \langle\downarrow,\ U(\boldsymbol{\theta}^*)\rho U^\dagger(\boldsymbol{\theta}^*)\rangle}$$

$$\text{(Sequence)}\ \frac{\langle P_1(\boldsymbol{\theta}^*),\ \rho\rangle \to \langle P_1'(\boldsymbol{\theta}^*),\ \rho'\rangle}{\langle P_1(\boldsymbol{\theta}^*); P_2(\boldsymbol{\theta}^*),\ \rho\rangle \to \langle P_1'(\boldsymbol{\theta}^*); P_2(\boldsymbol{\theta}^*),\ \rho'\rangle}$$

$$\text{(Case } m\text{)}\ \overline{\langle \mathbf{case}\ M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta}^*)}\ \mathbf{end}, \rho\rangle \to \langle P_m(\boldsymbol{\theta}^*), \mathcal{E}_m(\rho)\rangle\ ,\ \forall \text{ outcome } m \text{ of } M = \{M_m\}}$$

$$\text{(While}^{(T)}\ 0\text{)}\ \overline{\langle \mathbf{while}^{(T)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta}^*)\ \mathbf{done},\ \rho\rangle \to \langle\downarrow, \mathcal{E}_0(\rho)\rangle}$$

$$\text{(While}^{(T)}\ 1\text{)}\ \overline{\langle \mathbf{while}^{(T)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta}^*)\ \mathbf{done}, \rho\rangle \to \langle P_1(\boldsymbol{\theta}^*); \mathbf{while}^{(T-1)}, \mathcal{E}_1(\rho)\rangle}$$

(a)

$$\begin{aligned}
[\![\mathbf{abort}[\overline{q}]]\!]\rho &= \mathbf{0} \\
[\![\mathbf{skip}[\overline{q}]]\!]\rho &= \rho \\
[\![q := |0\rangle]\!]\rho &= \mathcal{E}^{\text{bool}}_{q\to 0}(\rho) \text{ or } \mathcal{E}^{\text{B-int}}_{q\to 0}(\rho) \\
[\![\overline{q} := U(\boldsymbol{\theta}^*)[\overline{q}]]\!]\rho &= U(\boldsymbol{\theta}^*)\rho U^\dagger(\boldsymbol{\theta}^*) \\
[\![P_1(\boldsymbol{\theta}^*); P_2(\boldsymbol{\theta}^*)]\!]\rho &= [\![P_2(\boldsymbol{\theta}^*)]\!]([\![P_1(\boldsymbol{\theta}^*)]\!]\rho) \\
[\![\mathbf{case}\ M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta}^*)}\ \mathbf{end}]\!]\rho &= \textstyle\sum_m [\![P_m(\boldsymbol{\theta}^*)]\!]\mathcal{E}_m(\rho) \\
[\![\mathbf{while}^{(T)}\ M[\overline{q}] = 1\ \mathbf{do}\ P_1(\boldsymbol{\theta}^*)\ \mathbf{done}]\!]\rho &= \textstyle\sum_{n=0}^{T-1} \mathcal{E}_0 \circ ([\![P_1(\boldsymbol{\theta}^*)]\!] \circ \mathcal{E}_1)^n(\rho)
\end{aligned}$$

(b)

Figure 1. Parameterized T-bounded quantum **while** programs: (a) operational semantics (b) denotational semantics.

These programs essentially don't contribute to the finite computation output, as semantically aborted programs always result in **zero** output state **0**.

We formalize this concept (essential-abortion for unparameterized programs may be analogously defined) so that the compilation of our programs could be optimized:

Definition 3.2 ("Essentially Abort"). *Let* $P(\boldsymbol{\theta}) \in$ ***q-while***$^{(T)}_{\overline{v}}(\boldsymbol{\theta})$. $P(\boldsymbol{\theta})$ "essentially aborts" *if one of the following holds:*

1. $P(\boldsymbol{\theta}) \equiv \mathbf{abort}[\overline{q}]$;
2. $P(\boldsymbol{\theta}) \equiv P_1(\boldsymbol{\theta}); P_2(\boldsymbol{\theta})$, *and either* $P_1(\boldsymbol{\theta})$ *or* $P_2(\boldsymbol{\theta})$ *essentially aborts;*
3. $P \equiv \mathbf{case}\ M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta})}\ \mathbf{end}$, *and each* $P_m(\boldsymbol{\theta})$ *essentially aborts.*

$$\text{(Sum Components)}\ \overline{\langle \underline{P_1(\boldsymbol{\theta}^*)} + \underline{P_2(\boldsymbol{\theta}^*)},\ \rho\rangle \to \langle \underline{P_1(\boldsymbol{\theta}^*)},\ \rho\rangle,\ \langle \underline{P_1(\boldsymbol{\theta}^*)} + \underline{P_2(\boldsymbol{\theta}^*)},\ \rho\rangle \to \langle \underline{P_2(\boldsymbol{\theta}^*)},\ \rho\rangle}$$

Figure 2. additive parameterized quantum bounded while-programs: operational semantics. We fix $\boldsymbol{\theta}^* \in \mathbb{R}^k$ and inherit all the other rules from parameterized programs in Fig. 1a.

4 Additive Parameterized Quantum Bounded While-Programs

We introduce a variant of *additive* quantum programs as a succinct way to describe the collection of programs that are necessary to compute the derivatives. To that end, we introduce our design of the syntax and the semantics of additive quantum programs as well as a compilation method that turns any additive quantum program into a collection of normal programs for the actual computation of derivatives.

4.1 Syntax

We adopt the convention to use underlines to indicate additive programs, such as $\underline{P(\boldsymbol{\theta})}$, to distinguish from normal program $P(\boldsymbol{\theta})$. The syntax of $\underline{P(\boldsymbol{\theta})}$ is given by

$$\begin{aligned}
\underline{P(\boldsymbol{\theta})} ::= {} & \underline{\mathbf{abort}[\overline{q}]} \mid \underline{\mathbf{skip}[\overline{q}]} \mid \underline{q := |0\rangle} \mid \underline{\overline{q} := U(\boldsymbol{\theta})[\overline{q}]} \mid \\
& \underline{P_1(\boldsymbol{\theta})}; \underline{P_2(\boldsymbol{\theta})} \mid \underline{\mathbf{case}}\ M[\overline{q}] = \overline{m \to \underline{P_m(\boldsymbol{\theta})}}\ \mathbf{end} \mid \\
& \underline{\mathbf{while}^{(T)}}\ M[\overline{q}] = 1\ \mathbf{do}\ \underline{P_1(\boldsymbol{\theta})}\ \mathbf{done} \mid \underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})},
\end{aligned}$$

where the only new syntax **+** is the *additive* choice. Intuitively, $\underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})}$ allows the program to either execute $\underline{P_1(\boldsymbol{\theta})}$ or $\underline{P_2(\boldsymbol{\theta})}$ nondeterminisitcally. The denotational semantics will include all possible execution traces. We assume **+** has lower precedence order than composition, and is left associative.[5] If $\underline{P(\boldsymbol{\theta})} = \underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})}$, then $\text{qVar}(\underline{P(\boldsymbol{\theta})}) \equiv \text{qVar}(\underline{P_1(\boldsymbol{\theta})}) \cup \text{qVar}(\underline{P_2(\boldsymbol{\theta})})$. Denote the collection of all non-deterministic $\underline{P(\boldsymbol{\theta})}$ s.t. $\text{qVar}(\underline{P(\boldsymbol{\theta})}) = \overline{v}$ as **add-q-while**$^{(T)}_{\overline{v}}(\boldsymbol{\theta})$.

4.2 Operational and Denotational Semantics

We exhibit operational semantics in Figure 2 and define a similar denotational semantics for any $\underline{P(\boldsymbol{\theta}^*)} \in$ **add-q-while**$^{(T)}_{\overline{v}}(\boldsymbol{\theta})$.

Definition 4.1 (Denotational Semantics). $\forall \boldsymbol{\theta}^*, \rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$,

$$[\![\underline{P(\boldsymbol{\theta}^*)}]\!](\rho) \equiv \{|\rho' : \langle \underline{P(\boldsymbol{\theta}^*)}, \rho\rangle \to^* \langle\downarrow, \rho'\rangle|\}. \tag{4.1}$$

Note that there is no sum in (4.1) compared with (3.3). This is because we want to capture the behavior of **+** by storing all possible execution traces in a multi-set. This resembles the idea of the sum operator in the differential lambda-calculus [14].

[5] E.g., $\underline{X} + \underline{Y}; \underline{Z} = \underline{X} + (\underline{Y}; \underline{Z})$, $X + Y + Z := (X + Y) + Z$.

4.3 Compilation Rules

We exhibit the compilation rules in Figure 3 as a way to transform an additive program $\underline{P(\boldsymbol{\theta})}$ into a multiset of normal programs. The compiled set of programs will be later used in the actual implementation of the differentiation procedure. Our compilation rule is also well-defined as it is compatible with the denotational semantics and operational semantics of $\underline{P(\boldsymbol{\theta})}$ in the following sense:

Proposition 4.2. *Denoting with $\coprod$ the union of multisets, then for any $\rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$,*

$$\{|\rho' : \rho' \neq \mathbf{0}, \rho' \in [\![\underline{P(\boldsymbol{\theta}^*)}]\!]\rho|\} = \coprod_{Q(\boldsymbol{\theta}) \in \mathtt{Compile}(\underline{P(\boldsymbol{\theta})})} \{|\rho' \neq \mathbf{0} : \langle Q(\boldsymbol{\theta}^*), \rho\rangle \to^* \langle \downarrow, \rho'\rangle|\}. \quad (4.2)$$

Proof. Structural Induction. See the full version [55] for details. □

Note that (4.2) removes $\mathbf{0}$ from the multi-set as we are only interested in non-trivial final states. In $\mathtt{Compile}(\underline{P(\boldsymbol{\theta})})$, some programs may essentially abort (Definition 3.2). For implementation, we are interested in the number of $Q(\boldsymbol{\theta}) \in \mathtt{Compile}(\underline{P(\boldsymbol{\theta})})$ that do not essentially abort:

Definition 4.3. *The* number of non-aborting programs *of $\underline{P(\boldsymbol{\theta})}$, denoted as $|\#\underline{P(\boldsymbol{\theta})}|$, is defined as*

$$|\#\underline{P(\boldsymbol{\theta})}| = |\mathtt{Compile}(\underline{P(\boldsymbol{\theta})}) \setminus \{|Q(\boldsymbol{\theta}) \in \mathtt{Compile}(\underline{P(\boldsymbol{\theta})}) : Q(\boldsymbol{\theta}) \text{ essentially aborts.}|\}|$$

where $|C|$ is the cardinality of a multiset C and $C_0 \setminus C_1$ denotes the multiset difference of C_0 and C_1.

We remark that $|\#\underline{P(\boldsymbol{\theta})}|$ could be exponentially large for general $\underline{P(\boldsymbol{\theta})}$, e.g., $\underline{P(\boldsymbol{\theta})} \equiv (Q_1 + R_1); \ldots; (Q_n + R_n)$. However, as we show in Section 7, for instances of additive programs from differentiation, this number is well bounded. (i.e., instances with exponential blow-up are *irrelevant* in our context.)

Example 4.1 (Generic-Case)**.** *Consider the following simple program with the case statement*

$$\underline{P(\boldsymbol{\theta})} \equiv \underline{\mathbf{case}}\ M[\overline{q}] = 0 \to \underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})},\quad 1 \to \underline{P_3(\boldsymbol{\theta})}$$

where $P_1(\boldsymbol{\theta}), P_2(\boldsymbol{\theta}), P_3(\boldsymbol{\theta}) \in \boldsymbol{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, none of them essentially aborts, and each of $P_1(\boldsymbol{\theta}), P_2(\boldsymbol{\theta}), P_3(\boldsymbol{\theta})$ contains no control gates. Then for any $\rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$, fixing $\boldsymbol{\theta}^*$ we have

$$\begin{aligned}
\langle \underline{P(\boldsymbol{\theta}^*)}, \rho\rangle &\overset{(Case\ m)}{\to} \langle \underline{P_1(\boldsymbol{\theta}^*)} + \underline{P_2(\boldsymbol{\theta}^*)}, M_0\rho M_0^\dagger\rangle \\
&\overset{(Sum)}{\to} \langle \underline{P_1(\boldsymbol{\theta}^*)}, M_0\rho M_0^\dagger\rangle \\
&\to^* \langle \downarrow, [\![P_1(\boldsymbol{\theta}^*)]\!](M_0\rho M_0^\dagger)\rangle; \\
\langle \underline{P(\boldsymbol{\theta}^*)}, \rho\rangle &\overset{(Case\ m)}{\to} \langle \underline{P_1(\boldsymbol{\theta}^*)} + \underline{P_2(\boldsymbol{\theta}^*)}, M_0\rho M_0^\dagger\rangle \\
&\overset{(Sum)}{\to} \langle \underline{P_2(\boldsymbol{\theta}^*)}, M_0\rho M_0^\dagger\rangle \to^* \langle \downarrow, [\![P_2(\boldsymbol{\theta}^*)]\!](M_0\rho M_0^\dagger)\rangle; \\
\langle \underline{P(\boldsymbol{\theta}^*)}, \rho\rangle &\overset{(Case\ m)}{\to} \langle \underline{P_3(\boldsymbol{\theta}^*)}, M_1\rho M_1^\dagger\rangle \\
&\to^* \langle \downarrow, [\![P_3(\boldsymbol{\theta}^*)]\!](M_1\rho M_1^\dagger)\rangle
\end{aligned}$$

(Atomic) $\mathtt{Compile}(P(\boldsymbol{\theta})) \equiv \{|P(\boldsymbol{\theta})|\}$, if $\underline{P(\boldsymbol{\theta})} \equiv \underline{\mathbf{abort}[\overline{v}]} \mid \underline{\mathbf{skip}[\overline{v}]} \mid \underline{q := |0\rangle} \mid \underline{\overline{v} := U(\boldsymbol{\theta})[\overline{v}]}$.

(Sequence) $\mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}; \underline{P_2(\boldsymbol{\theta})}) \equiv$

$$\begin{cases} \{|\mathbf{abort}|\}, \text{ if } \mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}) = \{|\mathbf{abort}|\}; \\ \{|\mathbf{abort}|\}, \text{ if } \mathtt{Compile}(\underline{P_2(\boldsymbol{\theta})}) = \{|\mathbf{abort}|\}; \\ \{|Q_1(\boldsymbol{\theta}); Q_2(\boldsymbol{\theta}) : Q_b(\boldsymbol{\theta}) \in \mathtt{Compile}(\underline{P_b(\boldsymbol{\theta})})|\}, \\ \qquad\qquad \text{otherwise.} \end{cases}$$

(Case m) $\mathtt{Compile}(\underline{\mathbf{case}}) \equiv \mathtt{FB}(\underline{\mathbf{case}})$, described in Fig.3*b*.

(While$^{(T)}$) $\mathtt{Compile}(\underline{\mathbf{while}^{(T)}})$: use (Case m) and (Sequence).

(Sum) $\mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})}) \equiv$

$$\begin{cases} \mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}) \coprod \mathtt{Compile}(\underline{P_2(\boldsymbol{\theta})}), \text{ if } \forall b \in \{1, 2\}, \mathtt{Compile}(\underline{P_b(\boldsymbol{\theta})}) \neq \{|\mathbf{abort}|\}; \\ \mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}), \text{ if } \mathtt{Compile}(\underline{P_2(\boldsymbol{\theta})}) = \{|\mathbf{abort}|\}, \\ \qquad\qquad \mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}) \neq \{|\mathbf{abort}|\}; \\ \mathtt{Compile}(\underline{P_2(\boldsymbol{\theta})}), \text{ if } \mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})}) = \{|\mathbf{abort}|\}, \\ \qquad\qquad \mathtt{Compile}(\underline{P_2(\boldsymbol{\theta})}) \neq \{|\mathbf{abort}|\}; \\ \{|\mathbf{abort}|\}, \text{otherwise} \end{cases}$$

(a)

1. $\forall m \in [0, w]$, let C_m denote the sub-multiset of $\mathtt{Compile}(\underline{P_m(\boldsymbol{\theta})})$ composed of programs that do *not* essentially abort; without loss of generality, assume $|C_0| \geq |C_1| \geq \cdots \geq |C_w|$.
2. If all C_m's are empty, return $\mathtt{FB}(\mathbf{case}) \equiv \{|\mathbf{abort}[\overline{v}]|\}$; else, pad each C_m to size $|C_0|$ by adding "$\mathbf{abort}[\overline{v}]$".
3. $\forall m \in [0, w]$, index programs in C_m as $\{|Q_{m,0}(\boldsymbol{\theta}), \cdots, \underline{Q_{m,|C_0|-1}(\boldsymbol{\theta})}|\}$. Return $\mathtt{FB}(\mathbf{case}) \equiv \{|\ \mathbf{case}\ M[\overline{q}] = \overline{m \to Q_{m,j^*}}\ \mathbf{end}\ |\}_{j^*}$ with $0 \leq j^* \leq |C_0| - 1$.

(b)

Figure 3. nondeterministic programs: (a) compilation rules. (b) "Fill and Break" ("$\mathtt{FB}(\bullet)$") procedure for computing $\mathtt{Compile}(\underline{\mathbf{case}})$. $\underline{\mathbf{case}}$ stands for $\underline{\mathbf{case}}\ M[\overline{q}] = \overline{m \to \underline{P_m(\boldsymbol{\theta})}}\ \mathbf{end}$; $\underline{\mathbf{while}^{(T)}}$ stands for $\underline{\mathbf{while}^{(T)}}\ M[\overline{q}] = 1\ \mathbf{do}\ \underline{P_1(\boldsymbol{\theta})}\ \mathbf{done}$. Here $\coprod$ denotes union of multisets. One may observe from a routine structural induction and the definition of "essentially abort" that: for all $\underline{P(\boldsymbol{\theta})}$, either $\mathtt{Compile}(\underline{P(\boldsymbol{\theta})}) = \{|\mathbf{abort}|\}$, or $\mathtt{Compile}(\underline{P(\boldsymbol{\theta})})$ does not contain essentially abort programs.

Hence by Definition 4.1.

$$[\![\underline{P(\boldsymbol{\theta}^*)}]\!]\rho = \{|[\![P_1(\boldsymbol{\theta}^*)]\!](M_0\rho M_0^\dagger), [\![P_2(\boldsymbol{\theta}^*)]\!](M_0\rho M_0^\dagger), [\![P_3(\boldsymbol{\theta}^*)]\!](M_1\rho M_1^\dagger)|\}$$

We verify computation results from the compilation rules are consistent with this. Writing "compilation rule" as "CP", one observes $\mathtt{Compile}(\underline{P_1(\boldsymbol{\theta})} + \underline{P_2(\boldsymbol{\theta})}) \overset{CP,Sum}{=} \{|P_1(\boldsymbol{\theta}), P_2(\boldsymbol{\theta})|\}$, while $\mathtt{Compile}(\underline{P_3(\boldsymbol{\theta})}) = \{|P_3(\boldsymbol{\theta})|\}$ since we assumed non-essentially-abortness. Apply our "fill and break" procedure to obtain $C_0 = \{|P_1(\boldsymbol{\theta}), P_2(\boldsymbol{\theta})|\}$, $C_1 = \{|P_3(\boldsymbol{\theta}), \mathbf{abort}[\overline{v}]|\}$.

$$\texttt{Compile}(\underline{P(\boldsymbol{\theta})}) = \left\{\!\!\left|\begin{array}{rl} \underline{\textbf{case}}\ M[\overline{q}] = 0 \to & P_1(\boldsymbol{\theta}), \\ 1 \to & P_3(\boldsymbol{\theta}), \\ \underline{\textbf{case}}\ M[\overline{q}] = 0 \to & P_2(\boldsymbol{\theta}), \\ 1 \to & \textbf{abort}[\overline{v}] \end{array}\right|\!\!\right\}$$

It's easy to check that evolving pursuant to the normal programs operational semantics (Fig 1) agrees with $[\![\underline{P(\boldsymbol{\theta}^*)}]\!]\rho$.

5 Observable and Differential Semantics

To capture physically observable quantities from quantum systems, physicists propose the notation of *observable* which is a Hermitian matrix over the same space. Any observable O is a combination of information about quantum measurements and classical values for each measurement outcome. To see why, let us take its spectral decomposition of $O = \sum_m \lambda_m |\psi_m\rangle\langle\psi_m|$. Then $\{|\psi_m\rangle\langle\psi_m|\}_m$ form a projective measurement. We can design an experiment to perform this projective measurement and output λ_m when the outcome is m. The *expectation* of the output is exactly given by

$$\mathrm{tr}(O\rho) = \sum_m \lambda_m \mathrm{tr}(M_m \rho M_m^\dagger). \tag{5.1}$$

The expectation $\mathrm{tr}(O\rho)$ represents meaningful classical information of quantum systems, which is also used in the loss functions in quantum machine learning applications. Thus, given any observable O, we will define the *observable* semantics of quantum programs as both the mathematical object to take derivatives from the original programs and the read-out of the programs that compute these derivatives.

One can repeat the $\{|\psi_m\rangle\langle\psi_m|\}_m$ measurement and use the statistical information to recover $\mathrm{tr}(O\rho)$. The number of iterations depends on the additive precision δ and the norm of O. To simplify our presentation, also to make a precise resource count as detailed in Section 7, we assume that[6]

$$-I_{\mathcal{H}} \sqsubseteq O \sqsubseteq I_{\mathcal{H}}. \tag{5.2}$$

Note that the observable O is different from quantum predicate P ($0 \sqsubseteq P \sqsubseteq I$), which is defined [13] as the quantum analogue of continuous logic with true values in $[0, 1]$. By statistically concentration bounds (e.g. the Chernoff bound), to approximate $\mathrm{tr}(O\rho)$ with additive error δ, one needs to repeat $O(1/\delta^2)$ times with $O(1/\delta^2)$ copies of initial states.

5.1 Observable Semantics

We define the *observable semantics* of both normal (denoted by $P(\boldsymbol{\theta}), P'(\boldsymbol{\theta})$) and additive (denoted by $\underline{S(\boldsymbol{\theta})}, \underline{S'(\boldsymbol{\theta})}$) parameterized programs as follows.

Definition 5.1 (Observable Semantics). *$\forall P(\boldsymbol{\theta}) \in \boldsymbol{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, any observable $O \in \mathcal{O}_{\overline{v}}$ and input state $\rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$, the* observable semantics *of P, denoted $[\![(O,\rho) \to P(\boldsymbol{\theta})]\!]$, is*

$$[\![(O,\rho) \to P(\boldsymbol{\theta})]\!](\boldsymbol{\theta}^*) \equiv \mathrm{tr}(O[\![P(\boldsymbol{\theta}^*)]\!]\rho), \forall \boldsymbol{\theta}^* \in \mathbb{R}^k. \tag{5.3}$$

[6] $\sqsubseteq$ is defined by $A \sqsubseteq B \iff B - A$ positive semidefinite.

Namely, $[\![(O,\rho) \to P(\boldsymbol{\theta})]\!]$ is a function from $\mathbb{R}^k$ to $\mathbb{R}$ whose value per point is given by (5.3).

Similarly, for any $\underline{S(\boldsymbol{\theta})} \in \boldsymbol{add\text{-}q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$ with $\texttt{Compile}(\underline{S(\boldsymbol{\theta})}) = \{|P_i(\boldsymbol{\theta})|\}_{i=1}^t$ where $P_i(\boldsymbol{\theta}) \in \boldsymbol{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, its observable semantics *is given by, $\forall \boldsymbol{\theta}^* \in \mathbb{R}^k$,*

$$[\![(O,\rho) \to \underline{S(\boldsymbol{\theta})}]\!](\boldsymbol{\theta}^*) \equiv \sum_{i\in[1,t]} [\![(O,\rho) \to P_i(\boldsymbol{\theta})]\!](\boldsymbol{\theta}^*). \tag{5.4}$$

To compute gradients of quantum observables for each parameter, one needs an ancilla variable as hinted by results in quantum information theory about gradient calculations for simple unitaries (e.g., Bergholm et al. [8], Schuld et al. [42]). To that end, we can easily extend quantum programs with ancilla variables. For each $j \in [1, k]$, the *j-th ancilla of $\boldsymbol{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$* is a quantum variable denoted by $A_{j,\overline{v}}$ disjoint from $\overline{v}$. We write A instead of $A_{j,\overline{v}}$ when $j, \overline{v}$ are clear from context. Ancilla A could consist of any number of qubits while we will mostly use *one-qubit A* in this paper.

We will only consider programs augmented with *one ancilla variable A_j* at any time. (So let us fix j for the following discussion). We will then consider programs that operate on the larger space $\mathcal{D}(\mathcal{H}_{\overline{v}\cup\{A\}})$ and an additional observable A to define the *observable semantics with ancilla.*

Definition 5.2 (Observable Semantics with Ancilla). *Given any $P'(\boldsymbol{\theta}) \in \boldsymbol{q\text{-}while}^{(T)}_{\overline{v}\cup\{A_{j,\overline{v}}\}}(\boldsymbol{\theta})$, any observable $O \in \mathcal{O}_{\overline{v}}$, input state $\rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$, and moreover the observable O_A on ancilla A, the* observable semantics *with ancilla of P, overloading the notation $[\![(O,\rho) \to P'(\boldsymbol{\theta})]\!]$, is*

$$[\![((O,O_A),\rho) \to P'(\boldsymbol{\theta})]\!](\boldsymbol{\theta}^*) \equiv \mathrm{tr}\Big((O_A \otimes O)[\![P'(\boldsymbol{\theta}^*)]\!](|\overline{0}\rangle_{\{A\}}\langle\overline{0}| \otimes \rho)\Big), \forall \boldsymbol{\theta}^* \in \mathbb{R}^k. \tag{5.5}$$

Again, $[\![((O,O_A),\rho) \to P(\boldsymbol{\theta})]\!]$ is a function from $\mathbb{R}^k$ to $\mathbb{R}$ whose value per point is given by (5.5).

Similarly, for $\underline{S'(\boldsymbol{\theta})} \in \boldsymbol{add\text{-}q\text{-}while}^{(T)}_{\overline{v}\cup\{A_{j,\overline{v}}\}}(\boldsymbol{\theta})$ s.t. $\texttt{Compile}(\underline{S'(\boldsymbol{\theta})}) = \{|P'_i(\boldsymbol{\theta})|\}_{i=1}^t$ where $P'_i(\boldsymbol{\theta}) \in \boldsymbol{q\text{-}while}^{(T)}_{\overline{v}\cup\{A_{j,\overline{v}}\}}(\boldsymbol{\theta})$, its observable semantics *is: $\forall \boldsymbol{\theta}^* \in \mathbb{R}^k$,*

$$[\![((O,O_A),\rho) \to \underline{S'(\boldsymbol{\theta})}]\!](\boldsymbol{\theta}^*) \equiv \sum_{i\in[1,t]} [\![((O,O_A),\rho) \to P'_i(\boldsymbol{\theta})]\!](\boldsymbol{\theta}^*).$$

The only difference from the normal observable semantics lies in (5.5), where we initialize the ancilla with $|0\rangle$, which is a natural choice and evaluate the observable $O_A \otimes O$. As we will see in the technique, the independence between O_A and O in the form of $O_A \otimes O$ will help us obtain the strongest guarantee of our differentiation procedure.

5.2 Differential Semantics

Given the definition of observable semantics, its differential semantics can be naturally defined by

Definition 5.3 (Differential Semantics). *Given additive program* $\underline{S(\boldsymbol{\theta})} \in \textbf{\textit{add-q-while}}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, *its j-th* differential semantics *is defined by*

$$\frac{\partial}{\partial\theta_j}(\llbracket(O,\rho)\to\underline{S(\boldsymbol{\theta})}\rrbracket), \tag{5.6}$$

which is again a function from $\mathbb{R}^k$ *to* $\mathbb{R}$. *Moreover, for any* $\underline{S'(\boldsymbol{\theta})} \in \textbf{\textit{add-q-while}}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$ *with ancilla A, we say that "*$\underline{S'(\boldsymbol{\theta})}$ computes the *j*-th differential semantics of $\underline{S(\boldsymbol{\theta})}$*" if and only if there exists an observable* O_A *on ancilla A for* $\underline{S'(\boldsymbol{\theta})}$ *such that* $\forall O \in \mathcal{O}_{\overline{v}}, \rho \in \mathcal{D}(\mathcal{H}_{\overline{v}})$,

$$\llbracket((O,O_A),\rho)\to\underline{S'(\boldsymbol{\theta})}\rrbracket = \frac{\partial}{\partial\theta_j}(\llbracket(O,\rho)\to\underline{S(\boldsymbol{\theta})}\rrbracket). \tag{5.7}$$

We remark that (5.6) is well defined because $\llbracket(O,\rho)\to\underline{S(\boldsymbol{\theta})}\rrbracket$ is a function from $\mathbb{R}^k$ to $\mathbb{R}$. It is also a smooth function because we assume that parameterized unitaries are entry-wise smooth, and the observable semantics is obtained by multiplication and addition of such entries. Note also that there is one specific choice of O_A in our current design. We leave it as a parameter to allow flexibility for future designs.

We also remark that the order of quantifiers in (5.7) is the strongest that one can hope for. This is because the observable semantics of $\underline{S(\boldsymbol{\theta})}$ will depend on O and ρ in general. Thus, the program to compute its differential semantics could also depend on O and ρ in general. However, in our definition, $\underline{S'(\boldsymbol{\theta})}$ is a single fixed program that works for any O and ρ regardless of the seemingly complicated relationship. This definition is consistent with the classical case where a single program can compute the derivatives for any input. We can achieve the same definition in the quantum setting and it is critical in the proof of Theorem 6.2 (item (5)).

6 Code Transformations and the Differentiation Logic

We describe the code transformation rules of the differentiation operator $\frac{\partial}{\partial\theta}(\cdot)$ in Section 6.1. We also define a logic and prove its soundness for reasoning about the correctness of these code transformations, with the following judgement

$$\underline{S'(\boldsymbol{\theta})}|\underline{S(\boldsymbol{\theta})}, \tag{6.1}$$

which states that $\underline{S'(\boldsymbol{\theta})}$ computes the differential semantics of $\underline{S(\boldsymbol{\theta})}$ in the sense of Definition 5.3. We fix $\theta = \theta_j$ and hence A stands for $A_{j,\overline{v}}$ and $\frac{\partial}{\partial\theta}$ for $\frac{\partial}{\partial\theta_j}$ through this section.[7]

6.1 Code Transformations

We first define some gates associated with the single-qubit rotation and the two-qubit coupling gates, which will appear in the code transformation rules. Let A be a single qubit.

[7] If A already exists, i.e., $\underline{S(\boldsymbol{\theta})} \in \textbf{add-q-while}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$, we treat $\overline{v}_{\text{new}}$ as $\overline{v}_{\text{old}} \cup A_{\text{old}}$ and add A_{new}. Any observable O on $\overline{v}_{\text{old}}$ becomes $O_{A_{\text{old}}} \otimes O$ on $\overline{v}_{\text{new}}$. Both A_{old} and A_{new} are initialized to $|0\rangle$ in observable semantics.

(Trivial)	$\frac{\partial}{\partial\theta}(\underline{\textbf{abort}[\overline{v}]}), \frac{\partial}{\partial\theta}(\underline{\textbf{skip}[\overline{v}]}), \frac{\partial}{\partial\theta}(\underline{q := \|0\rangle}) \equiv \underline{\textbf{abort}[\overline{v}\cup\{A\}]}$.
(Trivial-U)	$\frac{\partial}{\partial\theta}(\underline{\overline{v} := U(\boldsymbol{\theta})[\overline{v}]}) \equiv \underline{\textbf{abort}[\overline{v}\cup\{A\}]}$, if $\theta_j \notin \boldsymbol{\theta}$.
(1-qb)	$\frac{\partial}{\partial\theta}(\underline{q_1 := R_\sigma(\theta)[q_1]}) \equiv \underline{A, q_1 := R'_\sigma(\theta)[A, q_1]}$.
(2-qb)	$\frac{\partial}{\partial\theta}(\underline{q_1, q_2 := R_{\sigma\otimes\sigma}(\theta)[q_1, q_2]}) \equiv \underline{A, q_1, q_2 := R'_{\sigma\otimes\sigma}(\theta)[A, q_1, q_2]}$.
(Sequence)	$\frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta}); S_2(\boldsymbol{\theta})}) \equiv (\underline{S_1(\boldsymbol{\theta})}; \underline{\frac{\partial}{\partial\theta}(S_2(\boldsymbol{\theta}))}) + (\underline{\frac{\partial}{\partial\theta}(S_1(\boldsymbol{\theta}))}; \underline{S_2(\boldsymbol{\theta})})$.
(Case)	$\frac{\partial}{\partial\theta}(\underline{\textbf{case}\ M[\overline{q}] = m \to \underline{S_m(\boldsymbol{\theta})}\ \textbf{end}}) \equiv \underline{\textbf{case}\ M[\overline{q}] = m \to \frac{\partial}{\partial\theta}(\underline{S_m(\boldsymbol{\theta})})\ \textbf{end}}$.
(while$^{(T)}$)	Use (Case) and (Sequence).
(S-C)	$\frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})} + \underline{S_2(\boldsymbol{\theta})}) \equiv \frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})}) + \frac{\partial}{\partial\theta}(\underline{S_2(\boldsymbol{\theta})})$.

Figure 4. Code Transformation Rules. For (1-qb Rotation) and (2-qb Coupling), $(\sigma \in \{X, Y, Z\})$; $R'_\sigma(\theta), R'_{\sigma\otimes\sigma}(\theta)$ are as in Definition 6.1. $\theta_j \notin \boldsymbol{\theta}$ means "the unitary $U(\boldsymbol{\theta})$ trivially uses θ_j": for example in $P(\boldsymbol{\theta}) \equiv R_X(\theta_1); R_Z(\theta_2)$, $\boldsymbol{\theta} = (\theta_1, \theta_2)$ and $R_X(\theta_1)$ trivially uses θ_2.

Definition 6.1. *1. Consider unitary* $R_\sigma(\theta)$ *where* $\sigma \in \{X, Y, Z\}$. *We define unitary* $C\_R_\sigma(\theta)$ *as*

$$C\_R_\sigma(\theta) \equiv |0\rangle_A\langle 0| \otimes R_\sigma(\theta) + |1\rangle_A\langle 1| \otimes R_\sigma(\theta+\pi). \tag{6.2}$$

We also define a new gadget program $R'_\sigma(\theta)$ *as*

$$R'_\sigma(\theta)[A, q_1] \equiv A := H[A]; A, q_1 := C\_R_\sigma(\theta)[A, q_1]; A := H[A]. \tag{6.3}$$

2. Substituting $\sigma\otimes\sigma$ *for* σ *and* q_1, q_2 *for* q_1 *in Eqns (6.2,6.3), one defines* $C\_R_{\sigma\otimes\sigma}(\theta), R'_{\sigma\otimes\sigma}(\theta)$.

For 1-qubit rotation $R_\sigma(\theta)$, the "controlled-rotation" gate $C\_R_\sigma(\theta)$ maps $|0, q_1\rangle \mapsto |0\rangle \otimes R_\sigma(\theta)\,|q_1\rangle$, and $|1, q_1\rangle \mapsto |1\rangle \otimes R_\sigma(\theta+\pi)\,|q_1\rangle$; $R'_\sigma(\theta)$ conjugates $C\_R_\sigma(\theta)$ with Hadamard. Similarly for corresponding two-qubit coupling gates.

We exhibit our code transformation rules in Figure 4. For Unitary rules we only include 1-qubit rotations and two-qubit coupling gates, since they form a universal gate set and are easy to implement on quantum machines. It is also possible to include more unitary rules (e.g., by following the calculations in [42]), which we will leave as future directions.

6.2 The Differentiation Logic and Its Soundness

We develop the differentiation logic given in Figure 5 to reason about the correctness of code transformations. It suffices to show that our logic is sound. For ease of notation, in future analysis we write $\frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))$ in place of $\frac{\partial}{\partial\theta}(\underline{P(\boldsymbol{\theta})})$ when $P(\boldsymbol{\theta}) \in \textbf{q-while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$.

Theorem 6.2 (Soundness). *Let* $\underline{S(\boldsymbol{\theta})} \in \textbf{\textit{add-q-while}}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$, $\underline{S'(\boldsymbol{\theta})} \in \textbf{\textit{add-q-while}}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$. *Then,* $\underline{S'(\boldsymbol{\theta})}|\underline{S(\boldsymbol{\theta})}$ *implies that* $\underline{S'(\boldsymbol{\theta})}$ *computes the differential semantics of* $\underline{S(\boldsymbol{\theta})}$.

(Abort) $\frac{}{\frac{\partial}{\partial\theta}(\underline{\mathbf{abort}[\overline{v}]})|\underline{\mathbf{abort}[\overline{v}]}}$ (Skip) $\frac{}{\frac{\partial}{\partial\theta}(\underline{\mathbf{skip}[\overline{v}]})|\underline{\mathbf{skip}[\overline{v}]}}$

(Initialization) $\frac{}{\frac{\partial}{\partial\theta}(\underline{q := |0\rangle})|(q := |0\rangle)}\ \theta_j \notin \boldsymbol{\theta}$

(Trivial-Unitary) $\frac{}{\frac{\partial}{\partial\theta}(\overline{q} = U(\boldsymbol{\theta})[\overline{v}])|\overline{q} = U(\boldsymbol{\theta})[\overline{v}]}$

(Rot-Couple) $\frac{\frac{\partial}{\partial\theta}(\overline{q} = R_{\boldsymbol{\sigma}}(\boldsymbol{\theta})[\overline{v}])|(\overline{q} = R_{\boldsymbol{\sigma}}(\boldsymbol{\theta})[\overline{v}])}{\frac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta})})|\underline{S_0(\boldsymbol{\theta})} \qquad \frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})})|\underline{S_1(\boldsymbol{\theta})}}$

(Sequence) $\frac{\frac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta}); S_1(\boldsymbol{\theta})})|(\underline{S_0(\boldsymbol{\theta}); S_1(\boldsymbol{\theta})})}{\forall m,\ \frac{\partial}{\partial\theta}(\underline{S_m(\boldsymbol{\theta})})|\underline{S_m(\boldsymbol{\theta})}}$

(Case) $\frac{}{\frac{\partial}{\partial\theta}(\mathbf{case}\ M[\overline{q}] = \overline{m \to \underline{S_m(\boldsymbol{\theta})}}\ \mathbf{end})|\ \mathbf{case}\ M[\overline{q}] = \overline{m \to \underline{S_m(\boldsymbol{\theta})}}\ \mathbf{end}}$
$\frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})})\Big|\underline{S_1(\boldsymbol{\theta})}$

(While$^{(T)}$) $\frac{}{\frac{\partial}{\partial\theta}(\underline{\mathbf{while}^{(T)}}\ M[\overline{q}] = 1\ \mathbf{do}\ \underline{S_1(\boldsymbol{\theta})}\ \mathbf{done})|\ \underline{\mathbf{while}^{(T)}}\ M[\overline{q}] = 1\ \mathbf{do}\ \underline{S_1(\boldsymbol{\theta})}\ \mathbf{done}}$
$\frac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta})})|\underline{S_0(\boldsymbol{\theta})} \qquad \frac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})})|\underline{S_1(\boldsymbol{\theta})}$

(Sum Component) $\frac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta})} + \underline{S_1(\boldsymbol{\theta})})|(\underline{S_0(\boldsymbol{\theta})} + \underline{S_1(\boldsymbol{\theta})})$

Figure 5. The differentiation logic. Wherever applicable, $q \in \overline{v}, \overline{q} \subseteq \overline{v}, \underline{S_i(\boldsymbol{\theta})} \in \mathbf{add\text{-}q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$. In (Rot-Couple), $\boldsymbol{\sigma} \in \{X, Y, Z, X\otimes X, Y\otimes Y, Z\otimes Z\}$.

Let us highlight the ideas behind the proof of the soundness and all detailed proofs are deferred to the full version [55]. First remember that $\theta = \theta_j$ and for all the proofs we can choose $Z_A = |0\rangle\langle 0| - |1\rangle\langle 1|$ as the observable on the one-qubit ancilla A. Thus, we will omit Z_A and *overload* the notation, $\forall P'(\boldsymbol{\theta}) \in \mathbf{q\text{-}while}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$:

$$[\![(O,\rho)\to P'(\boldsymbol{\theta})]\!] \text{ means } [\![((O,Z_A),\rho)\to P'(\boldsymbol{\theta})]\!], \tag{6.4}$$

to simplify the presentation. We make similar overloading convention for $\underline{S'(\boldsymbol{\theta})} \in \mathbf{add\text{-}q\text{-}while}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$. Let us go through these logic rules one by one.

1. *Abort, Skip, Initialization, Trivial-Unitary* rules work because these statements do not depend on θ.
2. Since While$^{(T)}$ can be deemed as a macro of other statements, the correctness of *While*$^{(T)}$ rule follows by unfolding while$^{(T)}$ and applying other rules.
3. The *Sum Component* rule is due to the property of observable semantics ($[\![\cdot]\!]$) and additive operator (**+**):
$$\frac{\partial}{\partial\theta}([\![P_1 + P_2]\!]) = [\![\frac{\partial}{\partial\theta}(P_1)]\!] + [\![\frac{\partial}{\partial\theta}(P_2)]\!], \tag{6.5}$$
which follows from our definition design.
4. Our *Rot-Couple* rule is different from the phase-shift rule in [42] by using only one circuit in derivative computing. However, the proof of the *Rot-Couple* rule is largely inspired by the one of the phase-shift rule.
5. The proof of the *Sequence* rule relies very non-trivially on our design of the observable semantics with ancilla (Definition 5.2) and the strong requirement of computing differential semantics in Definition 5.3. Firstly, note that
$$\begin{aligned}[\![(O,\rho)\to\frac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta}); S_1(\boldsymbol{\theta})})]\!] &= [\![(O,\rho)\to\tfrac{\partial}{\partial\theta}(\underline{S_0(\boldsymbol{\theta})}); \underline{S_1(\boldsymbol{\theta})}]\!] \\ &\quad + [\![(O,\rho)\to\underline{S_0(\boldsymbol{\theta})}; \tfrac{\partial}{\partial\theta}(\underline{S_1(\boldsymbol{\theta})})]\!].\end{aligned}$$
We use the induction hypothesis to reason about each term above. Consider the case $\underline{S_0(\boldsymbol{\theta})} = S_0(\boldsymbol{\theta})$ and $\underline{S_1(\boldsymbol{\theta})} = S_1(\boldsymbol{\theta})$. Note that $S_0(\boldsymbol{\theta}), S_1(\boldsymbol{\theta}) \in \mathbf{q\text{-}while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$ and $\frac{\partial}{\partial\theta}(S_0(\boldsymbol{\theta})), \frac{\partial}{\partial\theta}(S_1(\boldsymbol{\theta})) \in \mathbf{add\text{-}q\text{-}while}^{(T)}_{\overline{v}\cup\{A\}}(\boldsymbol{\theta})$. First, we show
$$[\![(O,\rho)\to S_0(\boldsymbol{\theta}); \frac{\partial}{\partial\theta}(S_1(\boldsymbol{\theta}))]\!] = [\![(O, [\![S_0(\boldsymbol{\theta})]\!](\rho))\to\frac{\partial}{\partial\theta}(S_1(\boldsymbol{\theta}))]\!]. \tag{6.6}$$
This is because $\frac{\partial}{\partial\theta}(S_1(\boldsymbol{\theta}))$ computes the derivative for *any* input state and observable. We simply choose the input state $[\![S_0(\boldsymbol{\theta})]\!](\rho)$ and observable O. Secondly, we show
$$[\![(O,\rho)\to\frac{\partial}{\partial\theta}(S_0(\boldsymbol{\theta})); S_1(\boldsymbol{\theta})]\!] = [\![([\![S_1(\boldsymbol{\theta})]\!]^*(O),\rho)\to\frac{\partial}{\partial\theta}(S_0(\boldsymbol{\theta}))]\!]. \tag{6.7}$$
For (6.7), we don't change the state ρ but change the observable O by applying the *dual* super-operator $[\![S_1(\boldsymbol{\theta})]\!]^*$. Since $\frac{\partial}{\partial\theta}(S_0(\boldsymbol{\theta}))$ computes the derivative for any input state and any observable, we choose the input state ρ and observable $[\![S_1(\boldsymbol{\theta})]\!]^*(O)$. The dual super-operator $[\![S_1(\boldsymbol{\theta})]\!]^*$ has the property that $\mathrm{tr}(O[\![S_1(\boldsymbol{\theta})]\!](\rho)) = \mathrm{tr}([\![S_1(\boldsymbol{\theta})]\!]^*(O)\rho)$, which corresponds to the Schrodinger picture (evolving states) and Heisenberg picture (evolving observables) respectively in quantum mechanics.
6. The proof of the *Case* rule basically follows from the linearity of the observable semantics and the smooth semantics of Case. It is interesting to compare with the classical case [6] where the non-smoothness of the guard causes an issue for auto differentiation.

Example 6.1 (Simple-Case)**.** *Consider the following simple instantiating of Example 4.1*

$$\begin{aligned}P(\theta) \equiv \mathbf{case}\ M[q_1] = 0 &\to\quad R_X(\theta)[q_1]; R_Y(\theta)[q_1],\\ 1 &\to\quad R_Z(\theta)[q_1]\end{aligned}$$

Let us apply code transformation and compilation. Let CT, CP to denote "code transformation" and "compilation", and "Seq" and "Rot" denote Sequence and Rotation rules resp.

$$\frac{\partial}{\partial\theta}(\underline{P(\theta)}) \overset{CT,\mathbf{case}}{=} \begin{aligned}\mathbf{case}\ M[q_1] = 0 &\to\quad \tfrac{\partial}{\partial\theta}(R_X(\theta)[q_1]; R_Y(\theta)[q_1]),\\ 1 &\to\quad \tfrac{\partial}{\partial\theta}(R_Z(\theta)[q_1])\end{aligned}$$

$$\begin{array}{l} \\ \overset{CT,Seq+Rot}{=} \\ \\ \overset{\texttt{Compile}(\bullet)}{\longmapsto} \end{array} \begin{array}{l} \textbf{case } M[q_1]=0 \to \quad (R'_X(\theta)[A,q_1]; \\ \qquad\qquad\qquad\quad R_Y(\theta)[q_1])+ \\ \qquad\qquad\qquad\quad (R_X(\theta)[q_1]; \\ \qquad\qquad\qquad\quad R'_Y(\theta)[A,q_1]), \\ \qquad\qquad 1 \to \quad R'_Z(\theta)[A,q_1] \\ \\ \left\{\!\!\left| \begin{array}{l} \textbf{case } M[q_1]=0 \to \quad R'_X(\theta)[A,q_1]; \\ \qquad\qquad\qquad\quad R_Y(\theta)[q_1], \\ \qquad\qquad 1 \to \quad R'_Z(\theta)[A,q_1], \\ \\ \textbf{case } M[q_1]=0 \to \quad R_X(\theta)[q_1]; \\ \qquad\qquad\qquad\quad R'_Y(\theta)[A,q_1], \\ \qquad\qquad 1 \to \quad \textbf{abort}. \end{array} \right|\!\!\right\} \end{array}$$

7 Execution and Resource Analysis

In this section we illustrate the execution of the entire differentiation procedure and analyze its resource cost. Consider any program $P(\boldsymbol{\theta}) \in \textbf{q-while}^{(T)}_{\overline{v}}(\boldsymbol{\theta})$ and the parameter θ.

Execution. The first step in differentiation is to apply the code transformation rules (in Section 6) to $P(\boldsymbol{\theta})$ and obtain an additive program $\frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))$. Then one needs to compile $\frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))$ into a multiset $\{|P'_i(\boldsymbol{\theta})|\}_{i=1}^m$ of normal non-aborting programs $P'_i(\boldsymbol{\theta})$. The total count of these programs is given by $m = |\#\frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))|$. Note that the above procedure could be done at the compilation time.

Given any pair of O and ρ, the real execution to compute the derivative of $[\![(O,\rho) \to P(\boldsymbol{\theta})]\!]$ is to approximate the observable semantics $[\![(O,\rho) \to \frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))]\!]$. By Definition 5.2, we need to approximate

$$\sum_{i=1}^{m} \mathrm{tr}\Big((Z_A \otimes O)[\![P'_i(\boldsymbol{\theta})]\!]((|\overline{0}\rangle_{\{A\}}\langle\overline{0}|) \otimes \rho)\Big), \tag{7.1}$$

where each term is the observable $Z_A \otimes O$ on the output state of $P'_i(\theta)$ given input state ρ and the ancilla qubit $|0\rangle$.

To approximate the sum in (7.1) to precision δ , one could first treat the sum divided by m as the observable applied on the program that starts with a uniformly random choice of i from $1, \cdots, m$ and then execute $P'_i(\boldsymbol{\theta})$. By Chernoff bound, one only needs to repeat this procedure $O(m^2/\delta^2)$ times.

Resource count. We are only interested in non-trivial (extra) resource that is something that you wouldn't need if you only run the original program. Ancilla qubits count as the non-trivial resource. However, for our scheme, the number of required ancillae is 1 qubit per each parameter.

The more non-trivial resource is the number of the copies of input state (each copy of the input state is to be prepared from scratch), which is directly related to the number of repetitions in the procedure, which again connects to $m = |\#\frac{\partial}{\partial\theta}(P(\boldsymbol{\theta}))|$. We argue that our code transformation is efficient so that m is reasonably bounded. To that end, we show the relation between m and a natural quantity defined on the original program $P(\boldsymbol{\theta})$ (i.e., before applying any $\frac{\partial}{\partial\theta}(\cdot)$ operator) called the *occurrence count* of the parameter θ.

Definition 7.1. *The "Occurrence Count for θ_j" in $P(\boldsymbol{\theta})$, denoted $\mathrm{OC}_j(P(\boldsymbol{\theta}))$, is defined as follows:*

1. *If $P(\boldsymbol{\theta}) \equiv$ **abort**$[\overline{v}]$|**skip**$[\overline{v}]|q := |0\rangle$ ($q \in \overline{v}$), then $\mathrm{OC}_j(P(\boldsymbol{\theta})) = 0$;*
2. *$P(\boldsymbol{\theta}) \equiv U(\boldsymbol{\theta})$: if $U(\boldsymbol{\theta})$ trivially uses θ_j, then $\mathrm{OC}_j(P(\boldsymbol{\theta})) = 0$; otherwise $\mathrm{OC}_j(P(\boldsymbol{\theta})) = 1$.*
3. *If $P(\boldsymbol{\theta}) \equiv U(\boldsymbol{\theta}) = P_1(\boldsymbol{\theta}); P_2(\boldsymbol{\theta}))$ then $\mathrm{OC}_j(P(\boldsymbol{\theta})) = \mathrm{OC}_j(P_1(\boldsymbol{\theta})) + \mathrm{OC}_j(P_2(\boldsymbol{\theta}))$.*
4. *If $P(\boldsymbol{\theta}) \equiv$ **case** $M[\overline{q}] = \overline{m \to P_m(\boldsymbol{\theta})}$ **end** then $\mathrm{OC}_j(P(\boldsymbol{\theta})) = \max_m \mathrm{OC}_j(P_m(\boldsymbol{\theta}))$.*
5. *If $P(\boldsymbol{\theta}) \equiv$ **while**$^{(T)}$ $M[\overline{q}] = 1$ **do** $P_1(\boldsymbol{\theta})$ **done** then $\mathrm{OC}_j(P(\boldsymbol{\theta})) = T \cdot \mathrm{OC}_j(P_1(\boldsymbol{\theta}))$.*

Intuition of the "Occurrence Count" definition is clear: it basically counts the number of non-trivial occurrences of θ_j in the program, treating **case** as if it is deterministic. To see why this is a reasonable quantity, consider the auto-differentiation in the classical case. For any non-trivial variable v (i.e., v has some dependence on the parameter θ), we will compute both v and $\frac{\partial}{\partial\theta}(v)$ and store them both as variables in the new program. Thus, the classical auto-differentiation essentially needs the number of non-trivial occurrences more space and related resources. As we argued in the introduction, we cannot directly mimic the classical case due to the no-cloning theorem. The extra space requirement in the classical setting turns into the requirement on the extra copies of the input state in the quantum setting. Indeed, we can bound m by the occurrence count.

Proposition 7.2. $|\#\frac{\partial}{\partial\theta_j}(P(\boldsymbol{\theta}))| \le \mathrm{OC}_j(P(\boldsymbol{\theta}))$.

Proof. Structural induction. For details, see the full version [55]. □

8 Implementation and Case Study

We have built a compiler (written in OCaml) that implements our code transformation and compilation rules[8]. We use it to train one VQC instance with controls and empirically verify its resource-efficiency on representative VQC instances. Complete details can be found in the full version [55]. Experiments are performed on a MacBook Pro with a Dual-Core Intel Core i5 Processor clocked at 2.7 GHz, and 8GB of RAM.

8.1 Training VQC Instances with Controls

Consider a simple classification problem over 4-bit inputs $z = z_1z_2z_3z_4 \in \{0,1\}^4$ with true label given by $f(z) = \neg(z_1 \oplus z_4)$. We construct two 4-qubit VQCs P_1 (**no control**) and P_2 (**with control**) that consists of a single-qubit Pauli X,Y and Z rotation gate on each qubit and compare their performance.

For parameters $\Gamma = \{\gamma_1, \ldots, \gamma_{12}\}$ define the program

$$\begin{array}{rl} Q(\Gamma) \equiv & R_X(\gamma_1)[q_1]; R_X(\gamma_2)[q_2]; R_X(\gamma_3)[q_3]; R_X(\gamma_4)[q_4]; \\ & R_Y(\gamma_5)[q_1]; R_Y(\gamma_6)[q_2]; R_Y(\gamma_7)[q_3]; R_Y(\gamma_8)[q_4]; \\ & R_Z(\gamma_9)[q_1]; R_Z(\gamma_{10})[q_2]; R_Z(\gamma_{11})[q_3]; R_Z(\gamma_{12})[q_4], \end{array}$$

[8] Codes are availabe at https://github.com/LibertasSpZ/adcompile.

where q_1, q_2, q_3, q_4 refer to 4 qubit registers. Given parameters $\Theta = \{\theta_1, \ldots, \theta_{12}\}, \Phi = \{\phi_1, \ldots, \phi_{12}\}$, define

$$P_1(\Theta, \Phi) \equiv Q(\Theta); Q(\Phi). \tag{8.1}$$

Similarly, for parameters $\Theta = \{\theta_1, \ldots, \theta_{12}\}$,$\Phi = \{\phi_1, \ldots, \phi_{12}\}$, $\Psi = \{\psi_1, \ldots, \psi_{12}\}$, define

$$P_2(\Theta, \Phi, \Psi) \equiv Q(\Theta); \textbf{case } M[q_1] = \begin{matrix} 0 \to & Q(\Phi) \\ 1 \to & Q(\Psi). \end{matrix} \tag{8.2}$$

Note that P_1 and P_2 execute the same number of gates for each run. To use P_i to perform the classification or in the training, we first initialize q_1, q_2, q_3, q_4 to the classical feature vector $z = z_1z_2z_3z_4$ and then execute P_i. The predicted label y is given by measuring the 4^{th} qubit q_4 in the 0/1 basis.

We conduct a supervised learning by minimizing a *loss* function. A natural choice is the average negative log-likelihood which is commonly used in machine learning to evaluate classifiers that assign a certain probability to each label since quantum outcomes are probabilistic. However, this loss function is not currently supported by Pennylane. Denote the output of the classifier with input z and parameters $\boldsymbol{\theta}$ by $l_{\boldsymbol{\theta}}(z)$. To enable a direct comparison, we will treat $l_{\boldsymbol{\theta}}(z)$ as the average value of the labels from probabilistic quantum outcomes, and use the squared loss function as follows:

$$\text{loss} = \sum_{z \in \{0,1\}^4} 0.5 * (l_{\boldsymbol{\theta}}(z) - f(z))^2. \tag{8.3}$$

Note that loss is a function of $\boldsymbol{\theta} = (\Theta, \Phi)$ (or $\Theta, \Phi.\Psi$). More importantly, for each z, $l_{\boldsymbol{\theta}}(z)$ can be represented by the observable semantics of P_1(or P_2) with observable $|1\rangle\langle 1|$. Thus, the gradient of loss can be obtained by using the collection of $\frac{\partial}{\partial \alpha}(P_1)$ for $\alpha \in \Theta, \Phi$ (or $\frac{\partial}{\partial \alpha}(P_2)$ for $\alpha \in \Theta, \Phi, \Psi$). We classically simulate the training procedure with gradient descent. For the training of P_1, we use Pennylane for a direct comparison (see Figure 6). After 1000 epochs with some hyperparameters, the loss for P_1 (**no control**) attains a minimum of 0.5 in less than 100 epochs and subsequently plateaus. The loss for P_2 (**with control**) continues to decrease and attains a minimum of 0.016. It demonstrates the advantage of both controls in quantum machine learning and our scheme to handle controls, whereas previous schemes (such as Pennylane due to its quantum-node design [8]) fail to do so.

8.2 Benchmark Testing on Representative VQCs

We also test our compiler on important VQC candidates such as quantum neural-networks (QNN) for solving machine learning tasks [18], quantum approximate optimization algorithms (QAOA) for solving combinatorial optimization [17], and variational quantum eigensolver (VQE) for approximating ground state energies in quantum chemsitry [36], all of which are promising candidates for actual implementation on near-term quantum machines. These VQCs typically consists of alternating *layers* of single-qubit gates and two-qubit coupling gates, such as the 1-qubit, 2-qubit Pauli rotation

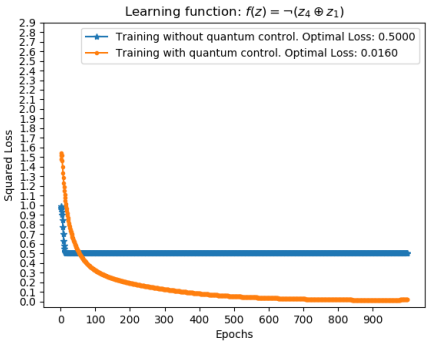

Figure 6. Training P_1 and P_2 to classify inputs according to the labelling function $f(z) = \neg(z_1 \oplus z_4)$.

gates considered in our paper, to represent the alternation between local interaction and neighboring interaction in real quantum physics systems.

We enrich these examples, by adding simple controls (the *if/condition* statement) or 2-bounded loops (the bounded-*while* statement) and increasing the number of qubits to 18~40, to make them sufficiently sophisticated but yet realistic for near-term quantum applications. For example, we use $\texttt{QNN}_{\texttt{M,i}}$ to denote an enriched QNN VQC instance of **m**edium size and with **i**f controls. The size of $\texttt{QNN}_{M,i}$ can also be directly illustrated by the number of qubits (#qb), the gate count (#gates), the number of alternating layers (#layers), and the number of lines to code such instances (#lines). Similarly for $\texttt{QNN}_{\texttt{L,w}}$ except that it is an instance of **l**arge size and with **w**hile controls.

A selective output performance of our compiler is in Table 2, with details in the full version [55]. It is easy to see that our scheme is also empirically resource-efficient as $|\#\frac{\partial}{\partial\theta}(\cdot)|$ is always reasonably bounded.

Table 2. Output on selective examples. $\{M, L\}$ stands for "medium, large"; $\{i, w\}$ stands for including "if, while".

$P(\boldsymbol{\theta})$	OC(·)	$\lvert\#\frac{\partial}{\partial\theta}(\cdot)\rvert$	#gates	#lines	#layers	#qb
$\texttt{QNN}_{M,i}$	24	24	165	189	3	18
$\texttt{QNN}_{M,w}$	56	24	231	121	5	18
$\texttt{QNN}_{L,i}$	48	48	363	414	6	36
$\texttt{QNN}_{L,w}$	504	48	2079	244	33	36
$\texttt{VQE}_{M,i}$	15	15	224	241	3	12
$\texttt{VQE}_{M,w}$	35	15	224	112	5	12
$\texttt{VQE}_{L,i}$	40	40	576	628	5	40
$\texttt{VQE}_{L,w}$	248	40	1984	368	17	40
$\texttt{QAOA}_{M,i}$	18	18	120	142	3	18
$\texttt{QAOA}_{M,w}$	42	18	168	94	5	18
$\texttt{QAOA}_{L,i}$	36	36	264	315	6	36
$\texttt{QAOA}_{L,w}$	378	36	1512	190	33	36

References

[1] Martín Abadi and Gordon D. Plotkin. 2019. A Simple Differentiable Programming Language. *Proc. ACM Program. Lang.* 4, POPL, Article 38 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371106

[2] Ali Javadi Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Fred Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. *Scaffold: Quantum Programming Language.* Technical Report TR-934-12. Princeton University.

[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[4] Alexandru Baltag and Sonja Smets. 2011. Quantum Logic as a Dynamic Logic. *Synthese* 179, 2 (2011).

[5] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5595–5637. http://dl.acm.org/citation.cfm?id=3122009.3242010

[6] Thomas Beck and Herbert Fischer. 1994. The if-problem in automatic differentiation. *J. Comput. Appl. Math.* 50, 1-3 (1994), 119–131.

[7] Marcello Benedetti, Erika Lloyd, and Stefan Sack. 2019. Parameterized quantum circuits as machine learning models. *arXiv e-prints* (Jun 2019). arXiv:1906.07682

[8] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, and Nathan Killoran. 2018. PennyLane: Automatic differentiation of hybrid quantum-classical computations. *arXiv:1811.04968* (2018).

[9] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195.

[10] Olivier Brunet and Philippe Jorrand. 2004. Dynamic Quantum Logic for Quantum Programs. *International Journal of Quantum Information* 2, 1 (2004).

[11] Rohit Chadha, Paulo Mateus, and Amílcar Sernadas. 2006. Reasoning About Imperative Quantum Programs. *Electronic Notes in Theoretical Computer Science* 158 (2006).

[12] George Corliss, Christèle Faure, Andreas Griewank, Lauren Hascoët, and Uwe Naumann (Eds.). 2002. *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Springer-Verlag New York, Inc., New York, NY, USA.

[13] Ellie D'Hondt and Prakash Panangaden. 2006. Quantum Weakest Preconditions. *Mathematical Structures in Computer Science* 16, 3 (2006).

[14] Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theoretical Computer Science* 309, 1-3 (2003), 1–41.

[15] Conal M. Elliott. 2009. Beautiful Differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 191–202. https://doi.org/10.1145/1596550.1596579

[16] Conal M. Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (July 2018), 29 pages. https://doi.org/10.1145/3236765

[17] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. (2014). arXiv:1411.4028

[18] Edward Farhi and Hartmut Neven. 2018. Classification with Quantum Neural Networks on Near Term Processors. (2018). arXiv:1802.06002

[19] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. 2007. Proof Rules for the Correctness of Quantum Programs. *Theoretical Computer Science* 386, 1-2 (2007).

[20] Simon J. Gay. 2006. Quantum Programming Languages: Survey and Bibliography. *Mathematical Structures in Computer Science* 16, 4 (2006).

[21] Gian Giacomo Guerreschi and Mikhail Smelyanskiy. 2017. Practical optimization for hybrid quantum-classical algorithms. (2017). arXiv:1701.01450

[22] Martin Giles. 2019. *IBM's new 53-qubit quantum computer is the most powerful machine you can use.* https://www.technologyreview.com/f/614346/ibms-new-53-qubit-quantum-computer-is-the-most-powerful-machine-you-can-use/

[23] Jonathan Grattage. 2005. A Functional Quantum Programming Language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS ?05)*. IEEE Computer Society, USA, 249?258. https://doi.org/10.1109/LICS.2005.1

[24] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, AdriàPuigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538 (10 2016), 471.

[25] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177

[26] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to Transduce with Unbounded Memory. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1828–1836. http://dl.acm.org/citation.cfm?id=2969442.2969444

[27] Andreas Griewank. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[28] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative Robustness Analysis of Quantum Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 31 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290344

[29] Yoshihiko Kakutani. 2009. A Logic for Formal Verification of Quantum Programs. In *Proceedings of the 13th Asian Conference on Advances in Computer Science: Information Security and Privacy* (Seoul, Korea) *(ASIAN'09)*. Springer-Verlag, Berlin, Heidelberg, 79–93. https://doi.org/10.1007/978-3-642-10622-4_7

[30] Gershon Kedem. 1980. Automatic Differentiation of Computer Programs. *ACM Trans. Math. Softw.* 6, 2 (June 1980), 150–165. https://doi.org/10.1145/355887.355890

[31] Jin-Guo Liu and Lei Wang. 2018. Differentiable learning of quantum circuit Born machines. *Phys. Rev. A* 98 (Dec 2018), 062324. Issue 6. https://doi.org/10.1103/PhysRevA.98.062324

[32] Nikolaj Moll, Panagiotis Barkoutsos, Lev S. Bishop, Jerry M. Chow, Andrew Cross, Daniel J. Egger, Stefan Filipp, Andreas Fuhrer, Jay M. Gambetta, Marc Ganzhorn, Abhinav Kandala, Antonio Mezzacapo, Peter Muller, Walter Riess, Gian Salis, John Smolin, Ivano Tavernelli, and Kristan Temme. 2018. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Science and Technology* 3, 3 (2018), 030503. arXiv:arXiv:1710.01022

[33] Bernhard Ömer. 2003. *Structured Quantum Programming*. Ph.D. Dissertation. Vienna University of Technology.

[34] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

[35] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

[36] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5 (2014), 4213.

[37] Gordon Plotkin. 2018. Some Principles of Differential Programming Languages. *POPL 2018* (2018).

[38] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79. arXiv:1801.00862

[39] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[40] Amr Sabry. 2003. Modeling Quantum Computing in Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 39 – 49. https://doi.org/10.1145/871895.871900

[41] J. W. Sanders and P. Zuliani. 2000. Quantum Programming. In *Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC '00)*. Springer-Verlag, Berlin, Heidelberg, 80 – 99.

[42] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. 2019. Evaluating analytic gradients on quantum hardware. *Phys. Rev. A* 99 (Mar 2019), 032331. Issue 3. https://doi.org/10.1103/PhysRevA.99.032331

[43] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. 2020. Circuit-centric quantum classifiers. *Phys. Rev. A* 101 (Mar 2020), 032308. Issue 3. https://doi.org/10.1103/PhysRevA.101.032308

[44] Peter Selinger. 2004. A brief survey of quantum programming languages. In *In Proceedings of the 7th International Symposium on Functional and Logic Programming*. Springer, 1–6.

[45] Peter Selinger. 2004. Towards a Quantum Programming Language. *Mathematical Structures in Computer Science* 14, 4 (2004).

[46] Bert Speelpenning. 1980. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. Ph.D. Dissertation. Champaign, IL, USA. AAI8017989.

[47] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. 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) *(RWDSL2018)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901

[48] Dave Wecker and Krysta Svore. 2014. LIQUi|⟩: A Software Design Architecture and Domain-Specific Language for Quantum Computing. *CoRR* abs/1402.4467 (2014). arXiv:1402.4467

[49] Robert Edwin Wengert. 1964. A Simple Automatic Derivative Evaluation Program. *Commun. ACM* 7, 8 (Aug. 1964), 463–464. https://doi.org/10.1145/355586.364791

[50] William K. Wootters and Wojciech H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803.

[51] Mingsheng Ying. 2011. Floyd–Hoare Logic for Quantum Programs. *ACM Transactions on Programming Languages and Systems* 33, 6 (2011).

[52] Mingsheng Ying. 2016. *Foundations of Quantum Programming*. Morgan Kaufmann.

[53] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. 2017. Invariants of Quantum Programs: Characterisations and Generation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 818?832. https://doi.org/10.1145/3009837.3009840

[54] D. Zhu, N. M. Linke, M. Benedetti, K. A. Landsman, N. H. Nguyen, C. H. Alderete, A. Perdomo-Ortiz, N. Korda, A. Garfoot, C. Brecque, L. Egan, O. Perdomo, and C. Monroe. 2019. Training of quantum circuits on a hybrid quantum computer. *Science Advances* 5, 10 (2019). https://doi.org/10.1126/sciadv.aaw9918 arXiv:https://advances.sciencemag.org/content/5/10/eaaw9918.full.pdf

[55] Shaopeng Zhu, Shih-Han Hung, Shouvanik Chakrabarti, and Xiaodi Wu. 2020. On the Principles of Differentiable Quantum Programming Languages. (2020). arXiv:2004.01122