Exploring modal types for the Intel Quantum SDK (Extended Abstract)

Jennifer Paykin
jennifer.paykin@intel.com
Intel Labs, Intel Corporation
Hillsboro, Oregon, USA

1 Introduction

Quantum computers operate on a computational model based not on bits 0 and 1, but on qubits: linear combinations of $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ in a complex vector space [13]. As a result, programming quantum algorithms comes with limitations, including the limited operations allowed on qubits (unitary transformations, preparations, and measurements). On top of that, current quantum hardware, which treats a quantum computer as a co-processor, imposes even more constraints. For example, the physical distance between quantum and classical (non-quantum) components means that executing classical operations takes an order of magnitude more time than quantum ones, and so may not be achievable in the amount of time a quantum state can be maintained (coherence time) [21].

Modern quantum programming languages thus tend to fall in two camps. Quantum circuit description languages like Qiskit [16], t|ket⟩ [19], and Cirq [6] enable programmers to construct standalone *quantum circuits* that can be executed on quantum devices as they exist today or in the near future. As such, algorithms that mix classical and quantum logic must construct those two parts separately. On the other hand, idealized higher-level quantum programming languages like Q# [20] enable arbitrary mixing of quantum and classical operations so that programmers can focus on higher-level algorithm design; as a result, not all programs can be immediately deployed on real devices.

The Intel® Quantum SDK is a quantum programming framework that aims for the best of both worlds. It extends C++ with *quantum kernels* that mix classical and quantum operations, but which are compiled to *quantum basic blocks* executable near-term devices using a powerful runtime model [9]. The exact rules for how classical and quantum operations can mix in the Intel Quantum SDK are currently given as informal programming guidelines:

1. All qubit references must be resolvable at compile time. That is, the state of a qubit may change during runtime, but each qubit variable must correspond to a constant unique identifier.
2. If a quantum instruction depends on a classical parameter, that parameter cannot rely on any measurement results from that same kernel. However, it may rely on measurement results from previous quantum kernels.
3. Measurement outcomes can be used within the same quantum kernel as long as no quantum operations depend on them.
4. Classical operations can be contained within dynamic control flow (if statements and loops), but quantum operations can only occur in such structures if they can be unrolled at compile time.
5. The same qubit cannot be used as both a control and another qubit in the same gate.[1]

While some of these rules (e.g. 1 and 5) are checked at compile-time, others (2 and 3) are left to the user to enforce, and can lead to unexpected results if violated. Others (4) are checked at compile-time, but in a strict way that could potentially rule out valid programs.

This extended abstract proposes a type system for a simple quantum programming language inspired by the Intel Quantum SDK, drawing on ideas from modal and linear type systems. It uses three modes—compile-time, classical runtime, and quantum runtime—to ensure that data written by the quantum machine is not consumed in the same quantum kernel. We also give a compilation strategy to a runtime language with a goal of showing that well-typed programs are compiled to quantum basic blocks with the same semantics.

2 A high-level quantum-classical language

We start with a simple hybrid quantum-classical language capturing the main design components of the Intel Quantum SDK, which includes quantum instructions, assignments, and

TyDE '23, June 08, 2023, Seattle, WA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] This last constraint is typically handled by linear or affine type systems [18, 22], but it is easily checked because of Item 1 so we do not dwell on it in this abstract.

if statements.

$$\begin{aligned} \mathit{insn} ::= {} & x := \mathit{insn} \mid \mathit{insn}_1; \mathit{insn}_2 \mid \mathbf{invoke}(\mathit{insn}) \\ & \mid U(e_1, \ldots, e_n)(e'_1, \ldots, e'_m) \mid \texttt{Prep}(e) \mid \texttt{Meas}(e, x) \\ & \mid \mathbf{if}\ e\ \mathbf{then}\ \mathit{insn}_1\ \mathbf{else}\ \mathit{insn}_2 \mid \mathbf{for}\ x = e\ \mathbf{to}\ e'\ \mathbf{do}\ \mathit{insn}' \\ e ::= {} & x \mid c \in \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \mid (e_1, e_2) \mid e_1 + e_2 \mid \cdots \end{aligned}$$

Unitary gates $U(\vec{e_i})(\vec{e'_j})$ are primitive quantum operations that may take both quantum parameters $\vec{e_i}$ and classical parameters $\vec{e'_j}$. A preparation gate $\texttt{Prep}(e)$ resets the state of a qubit e to $|0\rangle$, and $\texttt{Meas}(e, x)$ measures a qubit and writes the measurement result to the boolean variable x.

The **invoke** instruction executes a quantum kernel on a quantum device. It should be the case that every invocation can be compiled to a quantum basic block.

Following [7, 17], we can define a small-step operational semantics that updates a classical state σ (an assignment of classical variables to values) and a quantum state given as a partial density matrix ρ: $\mathit{insn}/\langle\sigma, \rho\rangle \to \mathit{insn}'/\langle\sigma', \rho'\rangle$.[2]

3 Modes and types

Our goal is a type system for this classical-quantum language that allows as much mixing of classical and quantum operations as possible, while still preserving the semantics when compiled. We use the compile-time mode **C** to indicate a computation or variable available at compile-time. We then use the classical (**R**) and quantum (**Q**) runtime modes to indicate whether or not a computation depends on the results from a quantum kernel.

Typing contexts Γ assign modes m and types τ to variables.

$$m ::= \mathbf{C} \mid \mathbf{R} \mid \mathbf{Q} \qquad \Gamma ::= \cdot \mid \Gamma, x :_m \tau$$
$$\tau ::= () \mid \mathbf{qbit} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{R} \mid \tau_1 \times \tau_2$$

Expressions are typed with respect to a mode, written $\Gamma \vdash e :_m \tau$. For example:

$$\frac{x :_m \tau \in \Gamma}{\Gamma \vdash x :_m \tau} \qquad \frac{\Gamma \vdash e_1 :_m \tau_1 \qquad \Gamma \vdash e_2 :_m \tau_2}{\Gamma \vdash (e_1, e_2) :_m \tau_1 \times \tau_2}$$

Fig. 1 shows a sample of typing rules for instructions: the judgment $\Gamma \vdash \mathit{insn} : m$ indicates that insn is well-typed at mode m under context Γ. In the quantum-specific instructions, expressions of qubit type must be available at mode **C** to satisfy Item 1. The parameters to unitary gates must be available at classical runtime, whereas measurement results (written to the variable x) are at quantum runtime, meaning they cannot be used as parameters in the same quantum kernel (Items 2 and 3).

As an example, let $\Gamma = z :_{\mathbf{R}} \mathbb{R}, q :_{\mathbf{C}} \mathbf{qbit}$. Then z can be used to compute a parameter to a unitary gate, as in $\Gamma \vdash \texttt{RZ}(q)(\frac{z\pi}{2}) : \mathbf{Q}$[3]. On the other hand, $x := \texttt{Meas}(q); \texttt{RZ}(q)(\frac{x\pi}{2})$ is not well-typed because the rotation parameter depends on the output of a measurement at mode **Q**.

$$\frac{\Gamma \vdash e_i :_{\mathbf{C}} \mathbf{qbit} \qquad \Gamma \vdash e'_j :_{\mathbf{R}} \mathbb{R}}{\Gamma \vdash U(e_1, \ldots, e_n)(e'_1, \ldots, e'_m) : \mathbf{Q}}$$

$$\frac{\Gamma \vdash e :_{\mathbf{C}} \mathbf{qbit}}{\Gamma \vdash \texttt{Prep}(e) : \mathbf{Q}} \qquad \frac{\Gamma \vdash e :_{\mathbf{C}} \mathbf{qbit} \qquad \Gamma \vdash x :_{\mathbf{Q}} \mathbb{B}}{\Gamma \vdash \texttt{Meas}(e, x) : \mathbf{Q}}$$

$$\frac{\Gamma \vdash e :_{\mathbf{C}} \mathbb{B} \qquad \Gamma \vdash \mathit{insn}_1 : \mathbf{Q} \qquad \Gamma \vdash \mathit{insn}_2 : \mathbf{Q}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ \mathit{insn}_1\ \mathbf{else}\ \mathit{insn}_2 : \mathbf{Q}}$$

$$\frac{\Gamma \vdash e :_m \mathbb{B} \qquad \Gamma \vdash \mathit{insn}_1 : m \qquad \Gamma \vdash \mathit{insn}_2 : m \qquad m \in \{\mathbf{C}, \mathbf{R}\}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ \mathit{insn}_1\ \mathbf{else}\ \mathit{insn}_2 : \mathbf{R}}$$

$$\frac{\Gamma \vdash e_i :_{\mathbf{C}} \mathbb{N} \qquad \Gamma, x :_{\mathbf{C}} \mathbb{N} \vdash \mathit{insn} : \mathbf{Q}}{\Gamma \vdash \mathbf{for}\ x = e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ \mathit{insn} : \mathbf{Q}}$$

$$\frac{\Gamma \vdash e :_m \tau \qquad \Gamma \vdash x :_m \tau}{\Gamma \vdash x := e : m'} \qquad \frac{\Gamma \vdash \mathit{insn}_1 : m \qquad \Gamma \vdash \mathit{insn}_2 : m}{\Gamma \vdash \mathit{insn}_1; \mathit{insn}_2 : m}$$

$$\frac{\Gamma \vdash \mathit{insn} : \mathbf{Q}}{\mathbf{R} \cdot \Gamma \vdash \mathbf{invoke}(\mathit{insn}) : \mathbf{R}} \qquad \frac{m' \cdot \Gamma \vdash \mathit{insn} : m' \cdot m}{\Gamma \vdash \mathit{insn} : m}$$

Figure 1. Typing rules for instructions $\Gamma \vdash \mathit{insn} : m$.

There are two rules for **if** statements (and similarly for **for** statements, but elided for space), depending on if it is executed at quantum runtime or not. For the quantum runtime, if statements are only allowed if they can be unrolled at compile time: the conditional e must have mode **C**. For example, **if** $q_1 = q_2$ **then** $\texttt{Prep}(q_1)$ **else skip** is well-typed at mode **Q** since the qubit references q_1 and q_2 are resolvable at compile time. For the classical runtime or compile time, the conditional can be any expression available at that mode.

The last set of rules has to do with how different modes interact. The rule for $\mathbf{invoke}(\mathit{insn})$ lifts a quantum instruction at mode **Q** to the classical runtime. However, this can only be done in a context without any variables at **Q**, indicated by the context $\mathbf{R} \cdot \Gamma$. This *scaling* operator ensures that all variables in the typing context are at modes **R** or **C**, akin to linear logic's ! promotion rule [1].

$$m \cdot (\Gamma, x :'_m \tau) = m \cdot \Gamma, x_{m \cdot m'} \tau$$
$$m \cdot m = m \qquad \mathbf{C} \cdot m = \mathbf{C}$$
$$m_1 \cdot m_2 = m_2 \cdot m_1 \qquad \mathbf{R} \cdot \mathbf{Q} = \mathbf{R}$$

Finally, the last rule enables use of classical computations at mode **Q** (or compile-time computations at mode **R**). This is safe because when compiled, they can be pushed to outside the quantum kernel. For example, $\mathit{insn}_0 = \mathbf{if}\ x\ \mathbf{then}\ y := \frac{\pi}{2}\ \mathbf{else}\ y := \frac{3\pi}{2}$ can be typed at mode **R** under the context

[2] We omit its definition for the sake of space.

[3] $\texttt{RZ}(q)(e)$ is a unitary operator known as a Z rotation, and it takes one qubit and one classical parameter.

$\mathbf{R} \cdot \Gamma = x :_{\mathbf{R}} \mathbb{B}, y :_{\mathbf{R}} \mathbb{R}$ where $\Gamma = x :_{\mathbf{Q}} \mathbb{B}, y :_{\mathbf{Q}} \mathbb{R}$. Therefore:

$$\frac{\dfrac{}{q :_{\mathbf{C}} \mathbf{qbit}, x :_{\mathbf{Q}} \mathbb{B} \vdash \mathtt{Meas}(q,x) : \mathbf{Q}} \qquad \dfrac{\dfrac{}{\mathbf{R} \cdot \Gamma \vdash insn_0 : \mathbf{R}}}{\Gamma \vdash insn_0 : \mathbf{Q}}}{q :_{\mathbf{C}} \mathbf{qbit}, \Gamma \vdash \mathtt{Meas}(q,x); insn_0 :_{\mathbf{Q}} \mathbb{R}}$$

4 A quantum runtime language

Next, we specify a subset of the above language executable on realistic quantum device. This is done by distinguishing between purely classical instructions and quantum basic blocks (QBBs) consisting of purely quantum instructions, where all qubit references must be constant indexes $i \in \mathbb{N}$ into the quantum state. In addition, in unitary gates classical parameters will all be given as variables rather than expressions; the values assigned to variables must be fixed at the start of quantum runtime.

$$\begin{aligned} cinsn ::= {} & \mathbf{skip} \mid cinsn_1; cinsn_2 \mid z := cinsn \mid \mathbf{invoke}(qbb) \\ & \mid \mathbf{if}\ e\ \mathbf{then}\ cinsn_1\ \mathbf{else}\ cinsn_0 \mid \mathbf{for}\ x = e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ insn \\ qbb ::= {} & \mathbf{skip} \mid qbb_1; qbb_2 \\ & \mid U(i_1, \ldots, i_n)(x_1, \ldots, x_m) \mid \mathtt{Prep}(i) \mid \mathtt{Meas}(i,x) \end{aligned}$$

The same typing rules apply to the runtime language as the high-level language, although of course there are no if statements or assignments at mode **Q**.

The semantics of the runtime language is similarly given as small-step operational semantics.

Compiling the hybrid high-level language to the runtime language consists of two main steps:

- Conditional if statements containing quantum instructions are completely unrolled.
- Quantum kernels are broken into three parts: classical instructions executed before the QBB, the QBB itself, and classical instructions executed after the QBB.

Compilation occurs under a compile-time environment γ that assigns compile-time variables $x :_{\mathbf{C}} \tau \in \Gamma$ to values. We can guarantee that if $\Gamma \vdash e :_{\mathbf{C}} \tau$ then we can compile e to a value v of type τ under γ, written $\gamma \vdash e \downarrow v$.

A purely classical instruction $\Gamma \vdash insn :_{\mathbf{R}} \tau$ will be compiled to a single classical instruction in the runtime language: $\gamma \vdash insn \rightsquigarrow^{\mathbf{R}} cinsn$ such that $\Gamma \vdash cinsn :_{\mathbf{R}} \tau$. On the other hand, a quantum instruction $\Gamma \vdash insn :_{\mathbf{Q}} \tau$ will be compiled to a tuple of instructions $[cinsn, qbb, cinsn']$, written $\gamma \vdash insn \rightsquigarrow^{\mathbf{Q}} [cinsn, qbb, cinsn']$, corresponding to $cinsn; \mathbf{invoke}(qbb); cinsn'$.

For example, a unitary gate $U(e_1, \ldots, e_n)(e'_1, \ldots, e'_n)$ that satisfies $\Gamma \vdash e_i :_{\mathbf{C}} \mathbf{qbit}$ and $\Gamma \vdash e'_j :_{\mathbf{R}} \mathbb{R}$ will be compiled as

$$\frac{\gamma \vdash e_i \downarrow v_i \qquad x_j \text{ fresh}}{\gamma \vdash U(\overrightarrow{e_i})(\overrightarrow{e'_1}) \rightsquigarrow^{\mathbf{Q}} [\overrightarrow{x_j := e'_j}, U(\overrightarrow{v_i})(\overrightarrow{x_j}), \mathbf{skip}]}$$

If statements typed at mode **Q** will be unrolled as follows (and similarly for **false** and **for** loops):

$$\frac{\gamma \vdash e \downarrow \mathbf{true} \qquad \gamma \vdash insn_1 \rightsquigarrow^{\mathbf{Q}} [cinsn, qbb, cinsn']}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ insn_1\ \mathbf{else}\ insn_2 \rightsquigarrow^{\mathbf{Q}} [cinsn, qbb, cinsn']}$$

For $\Gamma \vdash x := e$, if the mode of e is **R**, it is moved to before the QBB, and if **Q**, it is moved to after the QBB.

$$\gamma \vdash x := e \rightsquigarrow^{\mathbf{Q}} \begin{cases} [x := e, \mathbf{skip}, \mathbf{skip}] & e \text{ at mode } \mathbf{R} \\ [\mathbf{skip}, \mathbf{skip}, x := e] & e \text{ at mode } \mathbf{Q} \end{cases}$$

Finally, for $insn_1; insn_2$, if $\gamma \vdash insn_i \rightsquigarrow^{\mathbf{Q}} [cinsn_i, qbb_i, cinsn'_i]$:

$$\begin{aligned} \gamma \vdash insn_1; insn_2 \rightsquigarrow^{\mathbf{Q}} [& cinsn_1; cinsn_2, \\ & qbb_1; qbb_2, \\ & cinsn'_1; cinsn'_2] \end{aligned}$$

where we assume variables written by $cinsn_2$ are not read by qbb_1 or $cinsn'_1$.[4] Intuitively, this is safe because variables written to in $cinsn'_1$ are all at mode **Q**, and thus will not be needed by $cinsn_2$ or qbb_2.

The safety of this compilation strategy is captured by the following conjecture, which is left for future work:

Conjecture 1. *Let γ is a compile-time environment for Γ.*
If $\Gamma \vdash insn : \mathbf{R}$ and $\gamma \vdash insn \rightsquigarrow^{\mathbf{R}} cinsn$, then $insn \cong cinsn$.
If $\Gamma \vdash insn : \mathbf{Q}$ and $\gamma \vdash insn \rightsquigarrow^{\mathbf{Q}} [cinsn, qbb, cinsn']$, then $insn \cong cinsn; \mathbf{invoke}(qbb); cinsn'$.

The equivalence of *insn* with *cinsn* will depend on the small-step semantics of the two languages; in particular we will define a logical relation between configurations in the two languages and prove that all well-typed programs and their compilation satisfy that relation.

5 Related and Future work

The modal type theory in this abstract is closely related to indexed variations of linear type systems including quantitative type theory [1], linear Haskell [2], and Granule [14], as well as modal type systems for coeffects [8, 10, 15] and staged compilation [5].

Though this system is focused on the particular programming environment of the Intel Quantum SDK, it could be adapted to other systems that mix classical and quantum computation for near-term quantum devices [4, 12], as well as for quantum intermediate representations that target a variety of hybrid quantum-classical architectures [3, 11].

This abstract is the first step towards developing a sound type checker for the Intel Quantum SDK. To apply it in practice, we must extend the type system to deal with arrays and pointers, loops, functions, and arbitrary classical C++/LLVM constructs. Once implemented, it would provide users crucial feedback for programming and debugging in terms of how classical and quantum instructions interact at a high level.

[4] Such variables can always be renamed without loss of generality.

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, 2018.

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[3] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[4] Evandro Chagas Ribeiro Da Rosa and Rafael De Santiago. Ket quantum programming. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(1):1–25, 2021.

[5] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM (JACM)*, 48(3):555–604, 2001.

[6] Cirq Developers. Cirq, December 2022. URL https://doi.org/10.5281/zenodo.7465577. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[7] Yuan Feng and Mingsheng Ying. Quantum Hoare logic with classical variables. *ACM Transactions on Quantum Computing*, 2(4):1–43, 2021.

[8] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. *ACM SIGPLAN Notices*, 51(9):476–489, 2016.

[9] Pradnya Khalate, Xin-Chuan Wu, Shavindra Premaratne, Justin Hogaboam, Adam Holmes, Albert Schmitz, Gian Giacomo Guerreschi, Xiang Zou, and A. Y. Matsuura. An LLVM-based C++ compiler toolchain for variational hybrid quantum-classical algorithms and quantum accelerators, 2022. arXiv:2202.11142.

[10] Daniel R Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[11] Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 255–264. IEEE, 2021.

[12] Tiffany M Mintz, Alexander J Mccaskey, Eugene F Dumitrescu, Shirley V Moore, Sarah Powers, and Pavel Lougovski. QCOR: A language extension specification for the heterogeneous quantum-classical model of computation. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(2):1–17, 2020.

[13] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[14] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.

[15] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. *ACM SIGPLAN Notices*, 49(9):123–135, 2014.

[16] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. URL https://doi.org/10.5281/zenodo.2573505.

[17] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[18] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

[19] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩ : A retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, nov 2020. doi:10.1088/2058-9565/ab8e92. URL https://doi.org/10.1088/2058-9565/ab8e92.

[20] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the real world domain specific languages workshop 2018*, pages 1–10, 2018.

[21] Benoît Valiron, Neil J Ross, Peter Selinger, D Scott Alexander, and Jonathan M Smith. Programming the quantum future. *Communications of the ACM*, 58(8):52–61, 2015.

[22] André Van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004.