Toward A Quantum Programming Language for Higher-Level Formal Verification

FINN VOICHICK and MICHAEL HICKS, University of Maryland, USA

We present QuantumOne, a quantum programming language formalized within the Coq proof assistant and designed for formal verification of quantum algorithms. This language allows for working with superpositions of tagged unions, allowing for quantum data more complicated than simple qubits. This paper outlines the formal syntax, operational semantics, and typing judgment of QuantumOne.

CCS Concepts: • **Software and its engineering → Formal language definitions**; *Data types and structures*; Procedures, functions and subroutines; *Specialized application languages*; Formal software verification.

Additional Key Words and Phrases: quantum computing, tagged union, Coq

1 INTRODUCTION

Today's quantum programming languages often express the quantum portions of an algorithm as *circuits*, which operate on individual *qubits*. This design is natural, as near-term quantum computers are resource-limited, offering only a few dozen qubits for computation. However, this approach is tedious and error prone; quantum algorithms may benefit from a higher level of expression.

For example, Ambainis et al.'s boolean formula algorithm [2] involves a quantum walk on a tree, requiring a superposition of vertex indices. Because of the operations being done on these vertex indices, the most natural way to index vertices is often the *path* taken from the root to the vertex, a variable-length list. Expressing such an algorithm in an existing language, such as Quipper [4], would require writing a program dealing solely with some bit representation of a list. Expressing the program in Qiskit would be similar. SQIR [5] and QBRICKS [3] are both verification tools that only allow quantum data to be represented with qubits, but real algorithms deal with superpositions of objects that are more clumsy to represent as bitstrings.

Looking a little further out, programs can hopefully be written at a higher level of abstraction, but still in a way that leverages key features of quantum computers, such as state superposition and entanglement.

This paper presents work in progress on QuantumOne, a high-level language for writing quantum programs. QuantumOne has two key features. First, it uses sum and product types as the basis for quantum data, rather than qubits. These are closer to concepts that today's programmers are familiar with (e.g., records and objects). Second, QuantumOne's type system does not enforce a strict separation between classical and quantum data, but instead uses normal (classical) operations to specify quantum concepts. The notion of quantum measurement is naturally encoded by *discarding* data, and the concept of entanglement is encoded by variable *duplication*.

Some similar features are present in QML [1], but their semantics are defined in terms of category theory, while QuantumOne's are specified using a novel small-step operational semantics. These semantics have been formalized within the Coq proof assistant to serve as the basis for proofs of algorithm correctness. We intend in the future to write a formally-verified compiler from QuantumOne to SQIR [5]. This would allow a programmer to write a quantum program in QuantumOne within the Coq proof assistant, formally prove its correctness using our operational semantics and properties of QuantumOne, then compile the program to a quantum circuit that is provably correct.

The rest of this paper presents the syntax, semantics, and types of QuantumOne. Appendix C discusses some related languages a bit more.

PlanQC 2021, June 22, 2021, Anywhere on Earth
2021.

2 QUANTUMONE: SYNTAX

Let x range over an infinite set of variables (for example, ASCII strings), and g range over a set of single-qubit gates (for example, special unitary group SU(2)). Then, QuantumOne syntax is defined with the following two mutually inductive definitions for expressions and functions:[1]

$$\begin{aligned} e ::= {} & \langle\rangle \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \langle e_1, e_2\rangle \mid x \\ & \mid (\texttt{case}\ e\ \texttt{of} \mid \texttt{inl}\ x_0 \Rightarrow e_0 \mid \texttt{inl}\ x_1 \Rightarrow e_1) \\ & \mid (\texttt{control}\ e\ \texttt{with} \mid \texttt{inl}\ x_0 \Rightarrow e_0 \mid \texttt{inl}\ x_1 \Rightarrow e_1) \\ & \mid f\ e \mid \texttt{let}\ \langle x_1, x_2\rangle = e'\ \texttt{in}\ e \\ f ::= {} & \lambda x.e \mid g \end{aligned}$$

QuantumOne has standard classical features: a unit value $\langle\rangle$, variables x, constructors `inl` and `inr` and `case` analysis for sum types, constructor $\langle e_1, e_2\rangle$ and `let`-based pattern matching for pairs, and function definition and application. With units and sums, we can encode bits:

$$0 \coloneqq \texttt{inl}\ \langle\rangle \qquad 1 \coloneqq \texttt{inr}\ \langle\rangle$$

$$(\texttt{case}\ e\ \texttt{of} \mid 0 \Rightarrow e_0 \mid 1 \Rightarrow e_1) \coloneqq (\texttt{case}\ e\ \texttt{of} \mid \texttt{inl}\ x_* \Rightarrow e_0 \mid \texttt{inl}\ x_* \Rightarrow e_1)$$

Above, x_* is a fresh variable not free in either e_0 or e_1. (Encoding of bits on `control` is similar.)

In addition to these standard features, there are two "quantum" features: single-qubit gate application (the only way to produce superpositions) and `control` statements (a way to produce entanglement). The `control` statements are like `case` statements except that they produce a *pair* containing the value guarded on as well as the result. QuantumOne does not include a special measurement operation, but a measurement function can be constructed, shown in Section 2.3.

2.1 Example: Deutsch's algorithm

Here is how we might define Deutsch's algorithm in QuantumOne.

$$\texttt{hadamard}\ \left(\texttt{let}\ \langle x_d, x_t\rangle = U_f\ \langle \texttt{hadamard}\ 0, \texttt{hadamard}\ 1\rangle\ \texttt{in}\ x_d\right)$$

Here, `hadamard` is a Hadamard gate, and U_f a provided oracle function. This oracle should implement the unitary transformation $\langle x, y\rangle \mapsto \langle x, y \oplus f(x)\rangle$ for some function $f : \mathbb{Z}_2 \to \mathbb{Z}_2$. For example, the oracle corresponding to the identity function f is a CNOT gate and can be manually implemented by setting U_f to the following:

$$\lambda x.\texttt{let}\ \langle x_d, x_t\rangle = x\ \texttt{in control}\ x_d\ \texttt{with} \mid 0 \Rightarrow x_t \mid 1 \Rightarrow \texttt{not}\ x_t$$

Here, `not` denotes the Pauli-X gate. QuantumOne does not have a way to "oraclize" a classical function, but this could be a convenient addition to the language. The main challenge is that the language's type system (see Section 4) does not distinguish between classical and quantum data, so there would need to be a way to create oracles for functions that are not classical.

2.2 Duplication via Entanglement

A key reason that some languages distinguish between classical and quantum types is the no-cloning theorem. For example, the type system of Selinger and Valiron's quantum lambda calculus [8] distinguishes "reusable" (classical) types from other (potentially quantum) types, and their typing relation uses linear types to ensure that quantum variables are never reused.

QuantumOne takes an alternative approach which doesn't distinguish between classical and quantum types. We interpret "duplication" to mean "sharing via entanglement" rather than "cloning," very similar to the approach taken in QML [1]. For example, the program $(\lambda x.\langle x, x\rangle)(\texttt{hadamard}\ 0)$

[1]The parentheses on `case` and `control` are there to clarify that pipe characters ("|") are part of the language.

would produce a Bell state. In general, any variable use that *would* be disallowed in a linear type system is instead treated as *sharing*, and in a compiled circuit this would be implemented using CNOT gates to "copy" the value to another register. Section 3 shows how QuantumOne's operational semantics allows for this kind of duplication.

2.3 Measurement via Discarding

Unlike many quantum programming languages, QuantumOne does not include measurement as a special operation. Rather, measurement is accomplished by *discarding* information. Information is discarded whenever a variable is created but not used, so one can measure a variable by duplicating it and then discarding the duplicate. In particular, "measurement" can be defined by the function $\lambda x.(\lambda y.x)\ x$. The semantics and use of this function are described in Section 3.

2.4 Meta programming

QuantumOne does not include any form of recursion or higher-order functions, but it does include first-order functions. Our implementation of QuantumOne is embedded in the Coq proof assistant, so we can use meta-programming at the Coq level to construct QuantumOne functions recursively.

3 OPERATIONAL SEMANTICS

The semantics of many prior quantum programming languages are defined denotationally using unitary or density matrices over qubit-based quantum states. Doing so for QuantumOne would require translating its higher level constructs to qubits first, which could occlude a direct understanding of a program's meaning. As such, we define an operational semantics directly in terms of QuantumOne constructs. We are also working on a compiler for QuantumOne programs, discussed in Appendix D.

The operational semantics, given in full in Appendix A, defines two judgments. The first is $e \Longrightarrow S$, which says that QuantumOne expression e single-steps to *state* S. A state S a complex linear combination (a superposition) of *classical states*, each of which is a pair $|e, d\rangle$, where e is a QuantumOne expression and d is a list of values that have been discarded. Many of the operational rules are essentially classical except that they keep track of discarded values. For example, consider the rules for function application:

$$\frac{x \in \mathrm{FV}(e)}{(\lambda x.e)v \Longrightarrow |[x := v]e, []\rangle} \qquad \frac{x \notin \mathrm{FV}(e)}{(\lambda x.e)v \Longrightarrow |e, [v]\rangle}$$

Application of single-qubit gate g is written

$$g\ 0 \Longrightarrow G_{00}|0, []\rangle + G_{10}|1, []\rangle \qquad g\ 1 \Longrightarrow G_{01}|0, []\rangle + G_{11}|1, []\rangle$$

where g's semantics is represented by the unitary matrix with components G_{00}, G_{01}, G_{10}, and G_{11}.

The second judgment has the form $S \longrightarrow S''$. It is defined by a single rule that chooses a classical state $|e, d\rangle$ from within S, steps it using $e \Longrightarrow S'$, and then incorporates the result S' into the original S, producing S''. To see this judgment at work in conjunction with duplicating and discarding constructs, consider running a "coin flip" program that measures a $|+\rangle$ state, producing a maximally mixed state. “Measurement” is encoded via the term given in Section 2.3.

$$\begin{aligned}|((\lambda x.(\lambda y.x)\ x)\ (\texttt{hadamard}\ 0), [])\rangle &\longrightarrow \tfrac{1}{\sqrt{2}}|(\lambda x.(\lambda y.x)\ x)\ 0, []\rangle + \tfrac{1}{\sqrt{2}}|(\lambda x.(\lambda y.x)\ x)\ 1, []\rangle \\ &\longrightarrow \tfrac{1}{\sqrt{2}}|(\lambda y.0)\ 0, []\rangle + \tfrac{1}{\sqrt{2}}|(\lambda x.(\lambda y.x)\ x)\ 1, []\rangle \\ &\longrightarrow \tfrac{1}{\sqrt{2}}|0, [0]\rangle + \tfrac{1}{\sqrt{2}}|(\lambda x.(\lambda y.x)\ x)\ 1, []\rangle \\ &\longrightarrow^{*} \tfrac{1}{\sqrt{2}}|0, [0]\rangle + \tfrac{1}{\sqrt{2}}|1, [1]\rangle\end{aligned}$$

3.1 Expression Equivalence

In a fully-reduced program state, the expression in each term within the superposition is a *value*, either a unit, a tagged value, or a pair of values. Given a fully-reduced program state, one can compute the density matrix corresponding to the program state by computing a partial trace over the discard lists, “tracing out” the discarded information. The deferred measurement principle allows us to defer this partial trace operation until the end of the program’s execution, meaning that our operational semantics do not need to consider mixed states even though the language allows for measurement. This is why we can describe the “coin flip” program above as producing a maximally mixed state.

We can describe two expressions as equivalent if they reduce to program states corresponding to the same density matrix. For example, applying a Pauli-X gate to the coin-flip expression would be an equivalent program to the coin-flip expression alone because both correspond to the maximally mixed state:

$$\tfrac{1}{\sqrt{2}}|0, [0]\rangle + \tfrac{1}{\sqrt{2}}|1, [1]\rangle \sim \tfrac{1}{\sqrt{2}}|1, [0]\rangle + \tfrac{1}{\sqrt{2}}|0, [1]\rangle$$

3.2 Discussion

Our semantics takes a different approach from languages like Selinger and Valiron’s quantum lambda calculus [8], which define the program state as a classical expression with labelled holes pointing to indices in a separate quantum register. We found the “superposition of expressions” approach to be more convenient because of the way that values can be duplicated via entanglement; using labelled holes would require new holes to be created whenever values are duplicated, which would complicate the semantics with pointer arithmetic.

4 TYPING

QuantumOne defines a typing judgment of the form $\Gamma \vdash e : T$ where Γ is the usual partial map from variables to types. Judgment $\Gamma \triangleright f : F$ states that “f has function type F under assumptions from typing context Γ. Types T and F are standard, and defined as follows:

$$\begin{aligned}&T ::= \texttt{Unit} \mid T_0 + T_1 \mid T_1 \times T_2 \qquad \texttt{Bit} := \texttt{Unit} + \texttt{Unit} \\ &F ::= T \to T'\end{aligned}$$

Here are the typing rules:

$$\Gamma \vdash \langle\rangle : \texttt{Unit} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e : T_0}{\Gamma \vdash \texttt{inl}\ e : T_0 + T_1} \qquad \frac{\Gamma \vdash e : T_1}{\Gamma \vdash \texttt{inr}\ e : T_0 + T_1}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \langle e_1, e_2\rangle : T_1 \times T_2} \qquad \frac{\begin{array}{c} x_1 \neq x_2 \qquad \Gamma \vdash e' : T_1 \times T_2 \\ [x_1 \mapsto T_1][x_2 \mapsto T_2]\Gamma \vdash e : T\end{array}}{\Gamma \vdash \texttt{let}\ \langle x_1, x_2\rangle = e'\ \texttt{in}\ e : T}$$

$$\frac{\Gamma \vdash e : T_0 + T_1 \qquad [x_0 \mapsto T_0]\Gamma \vdash e_0 : T \qquad [x_1 \mapsto T_1]\Gamma \vdash e_1 : T}{\Gamma \vdash \texttt{case}\ e\ \texttt{of} \mid \texttt{inl}\ x_0 \Rightarrow e_0 \mid \texttt{inr}\ x_1 \Rightarrow e_1 : T}$$

$$\frac{\begin{array}{c}\mathrm{FV}(e_0) \setminus \{x_0\} = \mathrm{FV}(e_1) \setminus \{x_1\} \qquad \text{discard-free}(e_0) \qquad \text{discard-free}(e_1) \\ \Gamma \vdash e : T_0 + T_1 \qquad [x_0 \mapsto T_0]\Gamma \vdash e_0 : T \qquad [x_1 \mapsto T_1]\Gamma \vdash e_1 : T\end{array}}{\Gamma \vdash \texttt{control}\ e\ \texttt{with} \mid \texttt{inl}\ x_0 \Rightarrow e_0 \mid \texttt{inr}\ x_1 \Rightarrow e_1 : (T_0 + T_1) \times T}$$

$$\frac{[x \mapsto T]\Gamma \vdash e : T'}{\Gamma \triangleright \lambda x.e : T \to T'} \qquad \Gamma \triangleright g : \texttt{Bit} \to \texttt{Bit} \qquad \frac{\Gamma \triangleright f : T \to T' \qquad \Gamma \vdash e : T}{\Gamma \vdash f\ e : T'}$$

Here, $[x \mapsto T]\Gamma$ is an extension of Γ with x mapped to T (possibly overriding an earlier mapping).

Most of the rules are mostly standard since, as mentioned previously, QuantumOne does not strongly separate quantum and classical data. The interesting rule is the one for `control`. $\mathrm{FV}(e)$ is the set of free variables in e, defined in the usual way. The predicate discard-free(e) holds when e uses all of the variables it creates; it is formally defined in Appendix B. These premises effectively ensure that a `control` expression's subexpressions do not discard variables (i.e. they are free of measurement).

Other languages, such as Mingsheng Ying's QuGcl [10] have been designed to allow for measurement within quantum control statements. However, Ying's denotational semantics are non-compositional, meaning that equivalent subprograms cannot always be substituted to produce equivalent programs. We see this as a major hurdle for any kind of formal verification, and so we restrict `control` statements to disallow controlled measurement.

As an example, Deutsch's algorithm (see Section 2.1) can be typed like this:

$$\frac{\triangleright U_f : \texttt{Bit} \times \texttt{Bit} \to \texttt{Bit} \times \texttt{Bit}}{\vdash \texttt{hadamard}\ \big(\texttt{let}\ \langle x, y\rangle = U_f\ \langle \texttt{hadamard}\ 0, \texttt{hadamard}\ 1\rangle\ \texttt{in}\ x\big) : \texttt{Bit}}$$

QuantumOne does not include any recursion or looping mechanism, so it should be strongly normalizing, as described in the following conjecture. We are waiting to formally prove this until we have a verified compiler.

Conjecture 4.1 (Normalization). *Suppose* $\vdash e : T$ *is a valid typing judgment for some expression* e *and type* T*. Then* e *reduces in a finite number of steps to a value state of type* T*. That is,*

$$|e, []\rangle \longrightarrow^* |v_1, d_1\rangle + |v_2, d_2\rangle + \cdots + |v_n, d_n\rangle,$$

where every v_j *is a value such that* $\vdash v_j : T$*.*

5 FUTURE WORK

We hope that the semantics of QuantumOne will be easier to work with in a formal verification context than current tools, especially in programs that involve superpositions with more complicated structure where tagged unions could be helpful. The next step toward making this language more useful is constructing a certified compiler from this language to a lower-level circuit description language such as Sqir, as discussed in Appendix D. Formally proving this compiler correct would help prove Conjecture 4.1 because it would imply that the behavior of the semantics is equivalent to a finite quantum circuit.

Expressing and verifying more algorithms in this language would also be a good next step, particularly algorithms that combine classical and quantum computation or that can be naturally expressed with superpositions of tagged unions. We can also consider augmenting this language with additional features, like automatic oracle construction from classical functions (mentioned in Section 2.1) or a kind of "in-place" symmetric pattern matching [7].

ACKNOWLEDGMENTS

This material is based upon work supported 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 and the Air Force Office of Scientific Research under Grant No. FA95502110051.

REFERENCES

[1] Thorsten Altenkirch and Jonathan Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 249–258. https://doi.org/10.1109/LICS.2005.1

[2] Andris Ambainis, Andrew M Childs, Ben W Reichardt, Robert Špalek, and Shengyu Zhang. 2010. Any AND-OR formula of size N can be evaluated in time N^1/2+o(1) on a quantum computer. *SIAM J. Comput.* 39, 6 (2010), 2513–2530. https://doi.org/10.1137/080712167

[3] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

[4] 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

[5] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318

[6] 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

[7] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 348–364. https://doi.org/10.1007/978-3-319-89366-2_19

[8] Peter Selinger, Benoıt Valiron, et al. 2009. Quantum lambda calculus. *Semantic techniques in quantum computation* (2009), 135–172.

[9] André Van Tonder. 2004. A lambda calculus for quantum computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135. https://doi.org/10.1137/S0097539703432165

[10] Mingsheng Ying. 2016. *Foundations of Quantum Programming.* Elsevier Science, Chapter 6, 211–271.

A FULL OPERATIONAL SEMANTICS

The operational semantics is defined as two step relations. The first one defines how classical states step to quantum states. However, to account for discarded information, this step relation actually describes how a language expression steps to a pair containing an updated expression and a list of discarded expressions. In the inference rules below,

- "$e \implies \alpha|e_0, d_0\rangle + \beta|e_1, d_1\rangle$" can be interpreted as "e steps to a superposition of e_0 with d_0 discarded and e_1 with d_1 discarded, weighted by amplitudes α and β."
- *Values* are denoted with the letter v, and are expressions that are either unit, a tagged value, or a pair of values. Formally, $v ::= \langle\rangle \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \langle v_1, v_2\rangle$.
- "$[x := v]e$" is the expression e with every free occurrence of the variable x replaced with the value v.
- Assume that a gate g is represented by the matrix with scalars G_{00}, G_{01}, G_{10}, and G_{11}.

$$\frac{x \in \mathrm{FV}(e)}{(\lambda x.e)v \implies |[x := v]e, []\rangle} \qquad \frac{x \notin \mathrm{FV}(e)}{(\lambda x.e)v \implies |e, [v]\rangle}$$

$$\frac{x_0 \in \mathrm{FV}(e_0)}{(\texttt{case inl}\ v\ \texttt{of} \mid \texttt{inl}\ x_0 \Rightarrow e_0 \mid \texttt{inr}\ x_1 \Rightarrow e_1) \implies |[x_0 := v]e_0, [0]\rangle}$$

$$\frac{x_0 \notin \mathrm{FV}(e_0)}{(\texttt{case inl } v \texttt{ of } \mid \texttt{inl } x_0 \Rightarrow e_0 \mid \texttt{inr } x_1 \Rightarrow e_1) \Longrightarrow |e_0, [\texttt{inl } v]\rangle}$$

$$\frac{x_1 \in \mathrm{FV}(e_1)}{(\texttt{case inr } v \texttt{ of } \mid \texttt{inl } x_0 \Rightarrow e_0 \mid \texttt{inr } x_1 \Rightarrow e_1) \Longrightarrow |[x_1 := v]e_1, [1]\rangle}$$

$$\frac{x_1 \notin \mathrm{FV}(e_1)}{(\texttt{case inr } v \texttt{ of } \mid \texttt{inl } x_0 \Rightarrow e_0 \mid \texttt{inr } x_1 \Rightarrow e_1) \Longrightarrow |e_1, [\texttt{inr } v]\rangle}$$

$$(\texttt{control inl } v \texttt{ with } \mid \texttt{inl } x_0 \Rightarrow e_0 \mid \texttt{inr } x_1 \Rightarrow e_1) \Longrightarrow |\langle \texttt{inl } v, [x_0 := v]e_0\rangle, []\rangle$$

$$(\texttt{control inr } v \texttt{ with } \mid \texttt{inl } x_0 \Rightarrow e_0 \mid \texttt{inr } x_1 \Rightarrow e_1) \Longrightarrow |\langle \texttt{inr } v, [x_1 := v]e_1\rangle, []\rangle$$

$$\texttt{let } \langle x_1, x_2\rangle = \langle v_1, v_2\rangle \texttt{ in } e \Longrightarrow |(\lambda x_1.(\lambda x_2.e)\ v_2)\ v_1, []\rangle$$

$$g\ 0 \Longrightarrow G_{00}|0, []\rangle + G_{10}|1, []\rangle \qquad g\ 1 \Longrightarrow G_{01}|0, []\rangle + G_{11}|1, []\rangle$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\texttt{inl } e \Longrightarrow \alpha_1|\texttt{inl } e_1, d_1\rangle + \cdots + \alpha_n|\texttt{inl } e_n, d_n\rangle} \qquad \frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\texttt{inr } e \Longrightarrow \alpha_1|\texttt{inr } e_1, d_1\rangle + \cdots + \alpha_n|\texttt{inr } e_n, d_n\rangle}$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\langle e, e'\rangle \Longrightarrow \alpha_1|\langle e_1, e'\rangle, d_1\rangle + \cdots + \alpha_n|\langle e_n, e'\rangle, d_n\rangle} \qquad \frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\langle v, e\rangle \Longrightarrow \alpha_1|\langle v, e_1\rangle, d_1\rangle + \cdots + \alpha_n|\langle v, e_n\rangle, d_n\rangle}$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\begin{array}{c}(\texttt{case } e \texttt{ of } \mid \texttt{inl } x_0 \Rightarrow e' \mid \texttt{inl } x_1 \Rightarrow e'') \Longrightarrow \\ \alpha_1|\texttt{case } e_1 \texttt{ of } \mid \texttt{inl } x_0 \Rightarrow e' \mid \texttt{inl } x_1 \Rightarrow e'', d_1\rangle + \cdots + \alpha_n|\texttt{case } e_n \texttt{ of } \cdots, d_n\rangle\end{array}}$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\begin{array}{c}(\texttt{control } e \texttt{ with } \mid \texttt{inl } x_0 \Rightarrow e' \mid \texttt{inl } x_1 \Rightarrow e'') \Longrightarrow \\ \alpha_1|\texttt{control } e_1 \texttt{ with } \mid \texttt{inl } x_0 \Rightarrow e' \mid \texttt{inl } x_1 \Rightarrow e'', d_1\rangle + \cdots + \alpha_n|\texttt{control } e_n \texttt{ with } \cdots, d_n\rangle\end{array}}$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{\texttt{let } \langle x_1, x_2\rangle = e \texttt{ in } e' \Longrightarrow \alpha_1|\texttt{let } \langle x_1, x_2\rangle = e_1 \texttt{ in } e', d_1\rangle + \cdots + \alpha_n|\texttt{let } \langle x_1, x_2\rangle = e_n \texttt{ in } e', d_n\rangle}$$

$$\frac{e \Longrightarrow \alpha_1|e_1, d_1\rangle + \cdots + \alpha_n|e_n, d_n\rangle}{f\ e \Longrightarrow \alpha_1|f\ e_1, d_1\rangle + \cdots + \alpha_n|f\ e_n, d_n\rangle}$$

Finally, the main step relation can be defined by a single inference rule, sending quantum states to quantum states. We will use "$\longrightarrow$" do denote this step, and "$+\!\!+$" to denote list concatenation.

$$\frac{e_j \Longrightarrow \alpha'_1|e'_1, d'_1\rangle + \cdots + \alpha'_n|e'_n, d'_n\rangle}{\begin{array}{c}\alpha_1|v_1, d_1\rangle + \cdots + \alpha_{j-1}|v_{j-1}, d_j\rangle + \alpha_j|e_j, d_j\rangle + \cdots + \alpha_n|e_n, d_n\rangle \longrightarrow \\ \alpha_1|v_1, d_1\rangle + \cdots + \alpha_{j-1}|v_{j-1}, d_{j-1}\rangle + \alpha_j\left(\alpha'_1|e'_1, d_j +\!\!+ d'_1\rangle + \cdots + \alpha'_n|e'_n, d_j +\!\!+ d'_n\rangle\right) + \cdots + \alpha_n|e_n, d_n\rangle\end{array}}$$

This is the "main" step relation of a program. Running a program e can be seen as repeatedly applying this relation, starting with $|e, []\rangle$.

B DISCARD-FREE

Formally, the "discard-free" property is defined by the following inference rules:

$$\text{discard-free}(\langle\rangle) \qquad \frac{\text{discard-free}(e)}{\text{discard-free}(\texttt{inl}\ e)} \qquad \frac{\text{discard-free}(e)}{\text{discard-free}(\texttt{inr}\ e)}$$

$$\frac{\text{discard-free}(e_1) \qquad \text{discard-free}(e_2)}{\text{discard-free}(\langle e_1, e_2\rangle)} \qquad \text{discard-free}(x)$$

$$\frac{\text{discard-free}(e) \qquad \text{discard-free}(e_0) \qquad \text{discard-free}(e_1)}{\text{discard-free}(\texttt{control}\ e\ \texttt{with}\ |\ \texttt{inl}\ x_0 \Rightarrow e_0\ |\ \texttt{inr}\ x_1 \Rightarrow e_1)}$$

$$\frac{x_1 \in \mathrm{FV}(e) \qquad x_2 \in \mathrm{FV}(e) \qquad \text{discard-free}(e') \qquad \text{discard-free}(e)}{\text{discard-free}(\texttt{let}\ \langle x_1, x_2\rangle = e'\ \texttt{in}\ e)}$$

$$\frac{x \in \mathrm{FV}(e) \qquad \text{discard-free}(e) \qquad \text{discard-free}(e')}{\text{discard-free}((\lambda x.e)\ e')} \qquad \frac{\text{discard-free}(e)}{\text{discard-free}(g\ e)}$$

C COMPARISON WITH OTHER LANGUAGES

In this section, we will compare QuantumOne's features with that of a handful of other quantum programming languages, specifically QML[1], quantum lambda calculus [8], Quipper [4], QWIRE[6], QBRICKS[3], and SQIR[5]. (Note that there is more than one quantum lambda calculus [9], but we will discuss one of Selinger and Valiron's [8].)

C.1 Linear Types versus Sharing

The no-cloning theorem shows that there is no quantum circuit that can clone an arbitrary quantum state, so any quantum programming language with variables must handle variable re-use in some way. Some languages accomplish this using linear types, typing systems that disallow variables to be re-used. This is the approach taken in quantum lambda calculus and QWIRE, through typing relations and typechecking algorithms. Quipper is similar, but checks appear at runtime instead of compile time. SQIR does not have virtual qubit variables in the same way, but it includes a typing relation that prevents multi-qubit gates from being applied to a single location.

QML takes a different approach. It allows for variable duplication, but interprets it as *sharing* rather than *cloning*. We take this approach as well, as described in Section 2.2. One advantage to this approach is that it significantly simplifies the type system. Programs can combine classical and quantum operations in a single program, and the typing rules are no more complicated for the quantum data than the classical data.

QBRICKS seems to avoid the no-cloning problem altogether by limiting itself to describing whole circuits rather than applications to specific registers.

C.2 Formal Verification Oriented

SQIR and QBRICKS are two languages designed for formally verifying quantum programs. SQIR is implemented within the Coq proof assistant, and QBRICKS is implemented using the Why3 deductive verification platform. QBRICKS allows for a high degree of automation, and SQIR comes with VOQC, a verified optimizer for quantum circuits. Both have been used to verify interesting algorithms such as Grover's algorithm, and none of the other languages discussed were designed with formal verification in mind.

Given the difficulty in debugging quantum programs, we're planning to center the implementation of QuantumOne around formal verification. We have implemented the syntax, typing rules, and semantics of QuantumOne within the Coq proof assistant, and we have proven the correctness of small programs. A certified compiler from QuantumOne to SQIR would make it easy to optimize circuits with VOQC and would allow for formal verification using higher-level language features such as variables and tagged unions.

C.3 Superpositions of Tagged Unions

Any useful quantum programming language must allow for interactions between qubits, and thus must allow for some sort of collection of qubits. For most languages, this essentially means classical data structures containing qubits. Quipper and QWIRE allow for manipulating lists of qubits, but the length of the list must be known classically. (There is no notion of "superposition of lists of different length.") SQIR and QBRICKS both assume that all of the quantum data they deal with is a single list of qubits, or equivalently a superposition of bitstrings of a classically-known length. Quantum lambda calculus includes tagged unions that can contain qubits, but the tag is classical.

None of these languages allow for superpositions beyond qubits or collections of qubits. The only way to represent a *trit* in these languages would be with a pair of qubits, and a programmer would be forced to work with this two-bit representation.

QML and QuantumOne include superpositions of tagged unions. The "trit" type can be represented with "`Unit` + `Unit` + `Unit`," and language features are designed to work with these kinds of types. The kinds of case analysis that the two languages include are a bit different; QuantumOneonly allows for "`control`" expressions that keep the argument, while QML allows for a more powerful case analysis subject to orthogonality constraints.

D COMPILATION

We intend to write a compiler from QuantumOne to quantum circuits in SQIR [5], verified in Coq. It should be verified that the compiled circuit produces quantum states equivalent to those produced by the operational semantics. This will require us to represent all of our values with a series of qubits. The number of qubits needed to represent a value of type T is size(T), recursively defined by

$$\begin{aligned}\text{size}(\texttt{Unit}) &:= 0\\ \text{size}(T_0+T_1) &:= 1+\max\{\text{size}(T_0),\text{size}(T_1)\}\\ \text{size}(T_1\times T_2) &:= \text{size}(T_1)+\text{size}(T_2)\end{aligned}$$

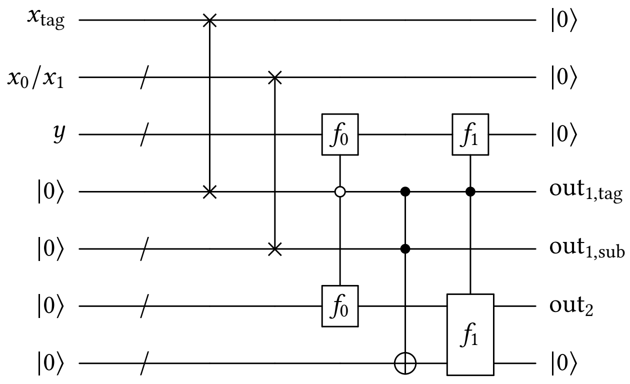
Tagged unions are represented with a single qubit for the tag and the rest of the qubits for the subvalues.

When compiling a QuantumOne expression, the compiler will be given a specific $|0\rangle$-initialized register on which to produce the desired value. In addition, there will need to be some collection of pointers associating variables with quantum registers, so that expressions with free variables can receive input from the correct register. Expressions that create new variables will need to allocate a new register, as will any expressions that duplicate variables. Duplication of variables will be accomplished with a series of CNOT gates applied to a newly-allocated register initialized to zero. SQIR does not have a mechanism for allocating and managing registers, so this mechanism will need to be implemented.

QuantumOne's "`control`" expressions will be implemented using controlled gates. As an example, consider this expression:

$$\texttt{control}\ x\ \texttt{with}\ |\ \texttt{inl}\ x_0 \Rightarrow f_0\ y\ |\ \texttt{inr}\ x_1 \Rightarrow f_1\ \langle y, x_0\rangle$$

This expression could be compiled into a circuit like this:



Note that the output registers of compiled expressions are distinct from the input registers containing the free variables used. This is done because in general the output type may be different from the input type. We are assuming that any discard-free expression compiles into a circuit that leaves its input register in the $|0\rangle$ state.

There is a lot of opportunity for optimization here, particularly in qubit reuse. It would save some qubits to allow for overlap between input and output registers, but this is more difficult to depict in a circuit diagram. For example, the f_1 gate in the above circuit could produce its output onto a register that includes the input y register, but the circuit diagram would look different depending on whether the type of $(f_1\ \langle y, x_0\rangle)$ is larger than the type of $\langle y, x_0\rangle$.