Deriving a Kronecker-Free Functional Quantum Simulator

MARTIN ELSMAN, University of Copenhagen, Denmark (mael@di.ku.dk)

The effect of a quantum circuit on a quantum state can be expressed by specifying the unitary complex matrix that the circuit denotes. This notion of denotation can be expressed compositionally by inductively defining the denotation of sub-circuits and by relating the denotation of a composed circuit by composing the denotation of sub-circuits. Concretely, the state of an n-qubit quantum computer can be represented by a 2^n complex vector and the denotation of an n-qubit quantum circuit can be expressed by a $2^n \times 2^n$ unitary complex matrix. Based on the denotational semantics of a quantum circuit, it is straightforward, on a classical computer, to simulate the effect of the quantum circuit, simply by applying the unitary matrix, denoted by the circuit, on the incoming state vector. For small n, this strategy works well whereas for larger n (e.g., $n > 8$), the size of the unitary matrix becomes a limiting factor. A question emerges. Can we do better?

In this paper we derive a purely functional interpreter for quantum circuits that operates directly on a circuit without first constructing the unitary matrix that the circuit denotes. The interpreter takes as input an n-qubit quantum circuit and a complex state vector of size 2^n and outputs a resulting state vector also of size 2^n. We demonstrate the performance benefits of the approach and highlight some of its promises.

1 INTRODUCTION

Most existing quantum-circuit simulation-frameworks are imperative in nature. They are centered around a programming model with operators that act on a set of qubits and implicitly update the underlying state space [2, 5, 6, 8, 12, 14, 22, 25, 26]. On the other hand, using a functional approach, it is possible to simulate the effect of a quantum circuit on the underlying state of a set of qubits by first establishing the unitary matrix that the circuit denotes and then applying that matrix to the state vector [24]. However, for general circuits, the space required by such a simulation approach is much larger than the space required to represent the state of the quantum computer. In general, the state of an n-qubit quantum computer can be represented in $O(2^n)$ space on a classical computer, whereas the unitary matrix denoted by the n-qubit circuit occupies $O(2^{2n})$ space.

We show that it is possible to derive (i.e., calculate) a functional quantum-state simulator for a quantum circuit without representing the unitary matrix that the circuit denotes. We further demonstrate the performance benefits of the functional interpreter compared to the unitary matrix approach by presenting running times with the different simulators for an implementation of Grover's algorithm [11]. The approach is surprisingly novel and may, for instance, be used for generating specialised quantum-circuit interpreters.

2 QUBITS, QUANTUM CIRCUITS, AND UNITARY MATRICES

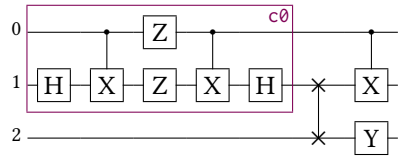
The basic building block of a quantum computer is a *qubit*, which can be modeled as a two-dimensional complex vector $\begin{bmatrix}\alpha & \beta\end{bmatrix}^{\mathrm{T}}$ specifying a linear combination $\alpha|0\rangle + \beta|1\rangle$ of the basis vectors $|0\rangle$ and $|1\rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$ (the *ket* $|0\rangle$ is defined as $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$ and $|1\rangle$ is defined as $\begin{bmatrix}0 & 1\end{bmatrix}^{\mathrm{T}}$).

A *single-qubit gate* operates on a single qubit and can be represented as a 2×2 *unitary* complex matrix U, meaning $U^\dagger U = UU^\dagger = I$ (norm-preserving and reversible).[1] Single-qubit gates include the *identity gate* I, the *Pauli-gates* X, Y, and Z, the *Hadamard gate* H, and the T-gate [17, 20, 29].

$$I = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \quad X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} \quad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \quad H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \quad T = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$$

[1] We use the notation $U^\dagger$ for the conjugated transpose of the unitary complex matrix U.

In 2nd Workshop on Quantum Software (WQS '2025). Extended Abstract. Seoul, Republic of Korea, June 2025.




```
val c0 = I**H++C(X)++Z**Z++C(X)++I**H
val c1 = c0**I++I**SW++C(X)**Y
```

$$\begin{bmatrix} -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 & 0 & 0 & 0 \\ 0 & -i & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & i \\ 0 & 0 & 0 & 0 & -i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & i & 0 \end{bmatrix}$$

Fig. 1. A quantum circuit, its functional specification, and the complex unitary matrix that it denotes.

The effect of "applying" the Hadamard gate H to a qubit in the $|0\rangle$ state (i.e., the vector $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$) puts the qubit in the *superposition* state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. A state of more qubits is modeled as a tensor product of the individual standard bases. Thus, a two-qubit state can expressed as a four-element complex vector $\begin{bmatrix}\alpha & \beta & \gamma & \theta\end{bmatrix}^{\mathrm{T}}$ that denotes a linear combination $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle$ of all combinations of basis vectors for the individual qubits, where $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\theta|^2 = 1$ (we write $|ab\rangle = |a\rangle \otimes |b\rangle$, for $a, b \in \{0, 1\}$, where $\otimes$ denotes tensor-product). The state space grows exponentially. For a three-qubit system, the state is expressed as an 8-element complex vector.

Single-qubit gates may be composed sequentially and in parallel to form quantum circuits. A circuit may be *drawn* using a simple diagram notation with each qubit represented by a horizontal line, single-qubit operations drawn as boxes, and so-called swaps drawn by connecting two lines with crosses. Sequential composition corresponds to multiplying unitary matrices and parallel composition corresponds to forming the tensor-product of two unitary matrices. In the diagram notation, tensors are implicit and represented as vertical composition whereas sequential composition is horizontal. So-called *control gates* provide the possibility that multiple qubits gets *entangled*, meaning that it becomes impossible to decompose the state of the involved qubits into states of the individual qubits. Control gates are drawn by vertically connecting a gate on one qubit line with a circle on another qubit line. Figure 1 shows a quantum circuit of *height* 3 (i.e., number of qubits) composed from a series of basic gates and with qubits connected using control gates and swaps.

Circuits can be represented using a functional language (e.g., Standard ML) type declaration:

```
datatype c = I | X | Y | Z | H | T | SW | C of c | Ten of c * c | Seq of c * c
```

Here `I` represents the identity gate, which together with `X`, `Y`, `Z`, `H`, and `T` are the basic one-qubit gates. Notice that, in diagrams, the identity gate is drawn as a horizontal line connecting the input directly to the output. The gate `SW` represents the two-qubit swap gate and `C` is a constructor for controlled circuits. Circuits may be composed sequentially using the `Seq` constructor, provided the sub-circuits operate on the same number of qubits, and in parallel using the `Ten` constructor. We use `**` as an infix-notation for `Ten`, which has higher precedence than the `++` infix-notation for `Seq`.

3 SEMANTICS

The semantics of well-formed circuits is defined inductively as a function that takes a circuit of height n and returns a unitary complex matrix of size $2^n \times 2^n$ denoting the circuit. The function, which makes use of libraries for complex numbers and matrices (Appendices A and B), is listed in Figure 2. The structure `C` provides operations on complex numbers. For instance, the function `C.fromIm` takes a `real` value and turns it into a complex imaginary number. Auxiliary functions on complex matrices are defined in Appendix C, including functions for complex matrix multiplication (`matmul`), complex tensor products (`tensor`), functions for converting integer lists to complex matrices (`fromIntM`), and a generalised `control` function, of type `mat → mat`, following [4]. We note here that the

```
fun sem (c:c) : mat =
    let val (c0, c1, ci) = (C.fromInt 0, C.fromInt 1, C.fromIm 1.0)
        val rsqrt2 = C.fromRe (1.0 / Math.sqrt 2.0)
        val eipi4 = C.exp(C.fromIm(Math.pi/4.0))
    in case c of I ⇒ fromIntM [[1,0], [0,1]]
               | X ⇒ fromIntM [[0,1], [1,0]]
               | Y ⇒ M.fromListList [[c0,C.~ ci], [ci,c0]]
               | Z ⇒ fromIntM [[1,0], [0,~1]]
               | H ⇒ M.fromListList [[rsqrt2,rsqrt2], [rsqrt2,C.~ rsqrt2]]
               | T ⇒ M.fromListList [[c1,c0], [c0,eipi4]]
               | SW ⇒ fromIntM [[1,0,0,0], [0,0,1,0], [0,1,0,0], [0,0,0,1]]
               | Seq(A,B) ⇒ matmul(sem B,sem A)
               | Ten(A,B) ⇒ tensor(sem A,sem B)
               | C A ⇒ control (sem A)
    end
```

Fig. 2. Semantics of circuits.

implementation of matrices is based on so-called *pull-arrays*, which use a functional representation, and which may be materialised by piping them (using the infix pipe operator `|>`) into row-major flat memory with the function `M.memoize`. The `tensor` operation forms the tensor-product $A \otimes B$ of two matrices A and B of dimensions $m \times n$ and $p \times q$, respectively:

$$(A \otimes B)_{ij} \quad = \quad A_{(i/p,j/q)} B_{(i\%p,j\%q)} \tag{1}$$

Here we write A_{ij} or $A_{(i,j)}$ to denote the zero-indexed (i, j)-element of the matrix A (row-major).

Based on the semantics, it is straightforward to define an evaluation function that, given a well-formed circuit of height h and a state vector of length 2^h returns a state vector (also of length 2^h) representing the result of "applying the circuit" to the input:

```
fun eval (c:c, v:vec) : vec = M.matvecmul_gen C.* C.+ (C.fromInt 0) (sem c) v
```

The `eval` function makes use of a generic function for matrix-vector multiplication, available in the matrix-library `M`. As an example, calling the `eval` function on the circuit `c1` and a state corresponding to the ket $|101\rangle$ (i.e., the state $\begin{bmatrix}0&0&0&0&0&1&0&0\end{bmatrix}^{\mathrm{T}}$) results in the state $\begin{bmatrix}0&0&0&0&(-i)&0&0&0\end{bmatrix}^{\mathrm{T}}$, which entails that there is a 100 percent probability of finding the resulting system in the basis state $|100\rangle$.[2]

4 KRONECKER-FREE CIRCUIT INTERPRETATION

We now demonstrate that it is possible to evaluate a circuit directly on an input state vector and return an output state vector without first creating the unitary complex matrix that the circuit denotes. We make use of the notion of *vectorisation* [18], which leads to the observation that

$$(A \otimes B)\mathsf{vec}(V) \quad = \quad \mathsf{vec}(BVA^T) \tag{2}$$

where $\mathsf{vec}(V)$ is the vector formed by "stacking" the columns of V. When V is a matrix of size $m \times n$, the *vectorisation* of V, written $\mathsf{vec}(V)$, is the vector of length mn and values $\mathsf{vec}(V)_i = V_{(i\%m,i/m)}$. For a vector v of length mn, the *unvectorisation* of v, written $\mathsf{unvec}(v)$, is the matrix V of size $m \times n$ such that $\mathsf{vec}(V) = v$. Concrete implementations appear in Appendix C.

We now derive a function `interp` that acts as an interpreter working directly on a circuit. To ease the reasoning below, when c is some circuit, we write $[\![c]\!]$ to denote the unitary complex matrix

[2]The high magnitude of basis state index 4 results in a measurement of the state $|100\rangle$. We use a big-endian encoding [17].

that the circuit denotes (as obtained with `sem c`). Semantically, the interpreter has the property that for any well-formed circuit c of height h and complex vector v of length 2^h, we have

$$\texttt{interp}\ c\ v \quad = \quad [\![c]\!]v \tag{3}$$

We derive the interpreter inductively on the structure of circuits, case by case, starting from 3 and from the property that $[\![c]\!]v = \texttt{eval}\ (c,v)$. The resulting function `interp` is listed in Figure 3.

The base cases are straightforward as the definitions for the base cases, including the swap gate, make directly use of the `eval` function. An exception is the case $c = \texttt{I}$ for which we simply return the input vector directly. For sequential composition (i.e., the `Seq` case), the derivation is as follows:

$$\begin{aligned}\texttt{interp}\ (A\texttt{++}B)\ v \quad &= \quad [\![A\texttt{++}B]\!]v \ = \ ([\![B]\!][\![A]\!])v \ = \ [\![B]\!]([\![A]\!]v) \\ &= \quad \texttt{interp}\ B\ (\texttt{interp}\ A\ v)\end{aligned} \tag{4}$$

We derive 4 from the definition of the `sem` function in Figure 2, from properties of matrix multiply, and by applying induction twice. For tensor composition (i.e., the `Ten` case), we have

$$\begin{aligned}\texttt{interp}\ (A\texttt{**}B)\ v \quad &= \quad [\![A\texttt{**}B]\!]v \\ &= \quad \texttt{vec}([\![B]\!]V[\![A]\!]^T) \text{ where } V = \texttt{unvec}\ v \qquad (5)\\ &= \quad \texttt{vec}(([\![A]\!]([\![B]\!]V)^T)^T) \qquad (6)\end{aligned}$$

Here we derive 5 from 2 and we derive 6 from the property that $(XY)^T = Y^T X^T$, for any compatible complex matrices X and Y. By induction and for any v' and v'' with appropriate lengths, we have

$$[\![A]\!]v' \quad = \quad \texttt{interp}\ A\ v' \tag{7}$$

$$[\![B]\!]v'' \quad = \quad \texttt{interp}\ B\ v'' \tag{8}$$

From 8 and the definition of matrix multiply, we have $[\![B]\!]V = (\texttt{map}\ (\texttt{interp}\ B)\ (V^T))^T$ and thus

$$([\![B]\!]V)^T \quad = \quad \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \tag{9}$$

Here `map` applies a function to each row (i.e., a vector) of a matrix. It now follows from 6, 9, and by introducing a variable W, that we have

$$\begin{aligned}\texttt{interp}\ (A\texttt{**}B)\ v \quad &= \quad \texttt{let}\ W = \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \\ &\quad\ \ \texttt{in}\ \texttt{vec}(([\![A]\!]W)^T)\end{aligned} \tag{10}$$

From 7 and the definition of matrix multiply, we have $[\![A]\!]W = (\texttt{map}\ (\texttt{interp}\ A)\ (W^T))^T$ and thus

$$([\![A]\!]W)^T \quad = \quad \texttt{map}\ (\texttt{interp}\ A)\ (W^T) \tag{11}$$

It now follows from 10, 11, and by introducing a variable Y, that we have

$$\begin{aligned}\texttt{interp}\ (A\texttt{**}B)\ v \quad &= \quad \texttt{let}\ V = \texttt{unvec}\ v \\ &\quad\ \ \texttt{let}\ W = \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \\ &\quad\ \ \texttt{let}\ Y = \texttt{map}\ (\texttt{interp}\ A)\ (W^T) \\ &\quad\ \ \texttt{in}\ \texttt{vec}(Y)\end{aligned} \tag{12}$$

For control gates (i.e., the `C` case), we leave it up to the reader to demonstrate that the result vector can be obtained by catenating the first half of the input vector with the result of interpreting the controlled circuit on the second half of the input vector, as shown in Figure 3.

```
fun interp (c:c) (v:vec) : vec =
   case c of I ⇒ v
           | Seq(A,B) ⇒ interp B (interp A v)
           | Ten(A,B) ⇒ let val V = unvec (pow2(height A),pow2(height B)) v
                            val W = mapRows (interp B) (M.transpose V)
                            val Y = mapRows (interp A) (M.transpose W)
                        in vec Y
                        end
           | C A ⇒ Vector.concat [vecTake (pow2(height A)) v,
                                  interp A (vecDrop (pow2(height A)) v)]
           | _ ⇒ eval (c,v)
```

Fig. 3. Kronecker-free quantum circuit interpreter. The functions pow2 and height are easily defined.

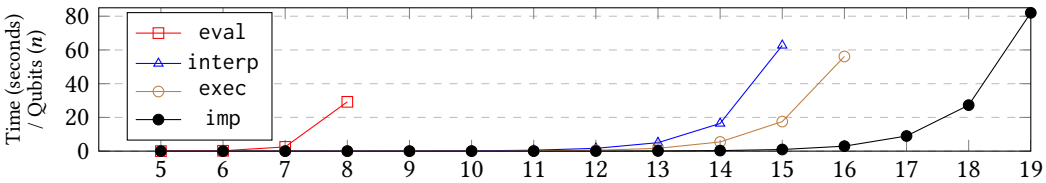


Fig. 4. Average execution times in seconds (over 10 runs with standard deviation less than two percent) for simulating Grover's algorithm with different simulators and different sizes of search problems (used qubits). Measurements are performed on a 2023 MacBook Pro, Apple M2 Max processor, 32GB of memory, and macOS 15.1.1. Programs are compiled with the MLton 20210117 Standard ML compiler [28].

5 BENCHMARKS AND OUTLOOK

We have implemented Grover's search algorithm [11] and simulated it with the different simulators and with different sizes of the search space (different number of qubits). Results are shown in Figure 4. We use `eval` to denote unitary simulation, `interp` to denote Kronecker-free interpretation, `exec` to denote `interp` with so-called *mutable array-views*, and `imp` to denote a classic imperative simulator [9], for which we have not established a relationship to the specified semantics. The circuit size grows with the size of the search space. With five qubits the search space is 2^5 and the number of basic gates used by the algorithm is 157, which include an encoding of the search oracle. With 16 qubits, the algorithm uses 25.342 basic gates and with 19 qubits and a 2^{19} search space, 85.369 basic gates are used. For the 19-qubit case, a state-vector for the full system occupies 2^{19+4} bytes (a complex number occupies 16 bytes), which amounts to about 8Mb. Using the `eval` function, execution time grows quickly due to the space used to hold large unitary matrices and the time used for matrix processing. The `interp` function performs much better. The refined simulator `exec`, which operates on a global state vector using a combination of one-dimensional and two-dimensional mutable array-views, provides a further three-times speedup (`exec`). We consider it future work to close the performance gap up to the full imperative version `imp`, which is most likely caused by the interpretative overhead of manipulating indexes through compositions of index functions.

There are plenty of possibilities for future work, including the possibility for combining the approach with parallelisation techniques [2, 5, 6, 8, 12, 26], with techniques for statically separating state spaces [3, 21], with techniques for gate fusion [13, 23], and with techniques for avoiding state space blow up when possible [1, 19]. Preliminary experiments show promising results with

respect to specialising the interpreter, using monadic GPU code generation [15], and with using the approach for deriving gate operations for a data-parallel language [7].

As a final remark, we are not claiming that the derived interpreter presented here is close in performance compared to state-of-art state vector simulators, such as qsim [27] and QuEST [16], which combine in-place state-vector updates with techniques for instruction-level vectorisation, gate fusion, and parallelism to achieve a performance that is magnitudes higher than the performance of the simple interpreter presented here.

REFERENCES

[1] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5. https://doi.org/10.1103/PhysRevA.70.052328

[2] A. Amariutei and Simona Caraiman. 2011. Parallel quantum computer simulation on the GPU. *15th International Conference on System Theory, Control and Computing* (2011), 1–6. https://api.semanticscholar.org/CorpusID:18896951

[3] Nicola Assolini, Alessandra Di Pierro, and Isabella Mastroeni. 2024. Abstracting Entanglement. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains* (Pasadena, CA, USA) *(NSAD '24)*. Association for Computing Machinery, New York, NY, USA, 34–41. https://doi.org/10.1145/3689609.3689998

[4] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical Review A* 52, 5 (Nov. 1995), 3457–3467. https://doi.org/10.1103/physreva.52.3457

[5] Daniel Claudino, Dmitry I. Lyakh, and Alexander J. McCaskey. 2024. Parallel quantum computing simulations via quantum accelerator platform virtualization. *Future Generation Computer Systems* 160 (2024), 264–273. https://doi.org/10.1016/j.future.2024.06.007

[6] Jun Doi and Hiroshi Horii. 2020. Cache Blocking Technique to Large Scale Quantum Computing Simulation on Supercomputers. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 212–222. https://doi.org/10.1109/qce49297.2020.00035

[7] Martin Elsman and Troels Henriksen. 2025. Gate Fusion is Map Fusion. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Seoul, Republic of Korea) *(ARRAY 2025)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3736112.3736143

[8] Jennifer Faj, Ivy Peng, Jacob Wahlgren, and Stefano Markidis. 2023. Quantum Computer Simulations at Warp Speed: Assessing the Impact of GPU Acceleration. arXiv:2307.14860 [cs.PF] https://arxiv.org/abs/2307.14860

[9] Bo Fang, M. Yusuf Özkaya, Ang Li, Umit V. Çatalyürek, and Sriram Krishnamoorthy. 2022. Efficient Hierarchical State Vector Simulation of Quantum Circuits via Acyclic Graph Partitioning. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 289–300. https://doi.org/10.1109/CLUSTER51413.2022.00041

[10] Emden R. Gansner and John H. Reppy. 2002. *The Standard ML Basis Library*. Cambridge University Press, USA.

[11] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[12] Eladio Gutierrez, Sergio Romero, Maria A. Trenas, and Emilio L. Zapata. 2008. Parallel Quantum Computer Simulation on the CUDA Architecture. In *Proceedings of the 8th International Conference on Computational Science, Part I* (Krakow, Poland) *(ICCS '08)*. Springer-Verlag, Berlin, Heidelberg, 700–709. https://doi.org/10.1007/978-3-540-69384-0_75

[13] Thomas Häner and Damian S. Steiger. 2017. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM. https://doi.org/10.1145/3126908.3126947

[14] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. https://doi.org/10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[15] Neil D. Jones. 2004. Transformation by interpreter specialisation. *Science of Computer Programming* 52, 1 (2004), 307–339. https://doi.org/10.1016/j.scico.2004.03.010 Special Issue on Program Transformation.

[16] T Jones, A Brown, I Bush, and S Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific Reports* 9, 2019 (2019).

[17] Phillip Kaye, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing*. Oxford University Press, Inc., USA.

[18] Hugo Daniel Macedo and José Nuno Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming* 78, 11 (2013), 2160–2191. https://doi.org/10.1016/j.scico.2012.07.012 Special section on Mathematics of Program Construction (MPC 2010) and Special section on methodological development of interactive

systems from Interaccion 2011.
[19] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (Jan. 2008), 963–981. https://doi.org/10.1137/050644756
[20] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, USA.
[21] Simon Perdrix. 2008. Quantum Entanglement Analysis Based on Abstract Interpretation. In *Static Analysis*, María Alpuente and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–282.
[22] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. 2024. Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers. arXiv:2406.14792 [quant-ph] https://arxiv.org/abs/2406.14792
[23] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The Quantum High Performance Software Testing Environment. arXiv:1601.07195 [quant-ph] https://arxiv.org/abs/1601.07195
[24] Robert Smith. 2023. A tutorial quantum interpreter in 150 lines of Lisp. https://www.stylewarning.com/posts/quantum-interpreter/ Blog post..
[25] 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 (RWDSL '18)*. ACM, 1–10. https://doi.org/10.1145/3183895.3183901
[26] Frank Tabakin and Bruno Juliá-Díaz. 2009. QCMPI: A parallel environment for quantum computing. *Computer Physics Communications* 180, 6 (2009), 948–964. https://doi.org/10.1016/j.cpc.2008.11.021
[27] Quantum AI team and collaborators. 2020. *qsim*. https://doi.org/10.5281/zenodo.4023103
[28] Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877
[29] Colin P. Williams. 2011. *Explorations in Quantum Computing*. Springer-Verlag London. https://doi.org/10.1007/978-1-84628-887-6

A APPENDIX: STANDARD ML SIGNATURE FOR COMPLEX NUMBERS

```
signature COMPLEX = sig
  type complex
  val mk       : real * real → complex
  val fromRe   : real → complex
  val fromIm   : real → complex
  val fromInt  : int → complex
  val conj     : complex → complex
  val re       : complex → real
  val im       : complex → real
  val mag      : complex → real
  val ~        : complex → complex            (* negation *)
  val +        : complex * complex → complex
  val -        : complex * complex → complex
  val *        : complex * complex → complex
  ...
  val sqrt     : complex → complex
  val exp      : complex → complex
  val abs      : complex → complex
  val pow      : complex * complex → complex
  val fmt      : StringCvt.realfmt → complex → string
  val toString : complex → string
end
```

B APPENDIX: STANDARD ML SIGNATURE FOR MATRICES

```
signature MATRIX = sig
  type α t
  val fromListList   : α list list → α t
  val dimensions     : α t → int * int
  val sub            : α t * int * int → α
  val map            : (α → β) → α t → β t
  val map2           : (α * β → γ) → α t → β t → γ t
  val listlist       : α t → α list list
  val transpose      : α t → α t
  val memoize        : α t → α t
  val tabulate       : int * int * (int*int → α) → α t
  val row            : int → α t → α vector
  val col            : int → α t → α vector
  val pp             : int → (α → string) → α t → string
  val ppv            : int → (α → string) → α vector → string
  val matmul_gen     : (α*α→α) → (α*α→α) → α → α t → α t → α t
  val matvecmul_gen  : (α*α→α) → (α*α→α) → α → α t → α vector → α vector
  ...
end
```

C APPENDIX: AUXILIARY COMPLEX MATRIX OPERATIONS

```
structure M = Matrix
structure V = Vector          (* Part of Standard ML Basis Library [10] *)

(* Matrix multiplication *)
fun matmul (t1:mat,t2:mat) : mat =
    M.matmul_gen C.* C.+ (C.fromInt 0) t1 t2 |> M.memoize

(* Tensor product of two complex matrices *)
fun tensor (a:mat,b:mat) : mat =
    let val (m,n) = M.dimensions a
        val (p,q) = M.dimensions b
    in M.tabulate(m * p, n * q,
                  fn (i,j) ⇒ C.* (M.sub(a,i div p,j div q),
                                  M.sub(b,i mod p,j mod q))
                 ) |> M.memoize
    end

(* Control matrix on square complex matrices *)
fun control (a:mat) : mat =
    let val n = M.nRows a
    in M.tabulate(2*n,2*n,                                      (* 1 0 0 0 *)
                  fn (r,c) ⇒ if r >= n andalso c >= n           (* 0 1 0 0 *)
                             then M.sub(a,r-n,c-n)              (* 0 0 a b *)
                             else if r = c then C.fromInt 1     (* 0 0 c d *)
                             else C.fromInt 0)
    end

(* List-to-matrix conversion *)
fun fromIntM (is:int list list) : mat = M.fromListList (map (map C.fromInt) is)

(* Row-major flattening and unflattening *)
fun flatten (m:mat) : vec =
    let val (r,c) = M.dimensions m
    in V.tabulate(r * c, fn i ⇒ M.sub(m,i div c,i mod c))
    end
fun unflatten (r,c) (v:vec) : mat =
    M.tabulate(r,c,fn (i,j) ⇒ V.sub(v,i*c+j))

(* Column-major flattening and unflattening *)
fun vec (m:mat) : vec = M.transpose m |> flatten
fun unvec (r:int,c:int) (v:vec) : mat = unflatten (r,c) v |> M.transpose

(* Take and drop elements from a vector *)
fun vecTake (n:int) (v:vec) = V.tabulate(n, fn i ⇒ V.sub(v,i))
fun vecDrop (n:int) (v:vec) = V.tabulate(V.length v - n, fn i ⇒ V.sub(v,i+n))

(* Map a function over the rows of a matrix *)
fun mapRows (f:vec→vec) (a:mat) : mat =
    List.tabulate(M.nRows a, fn i ⇒ f(M.row i a)) |> M.fromVectorList
```