quAPL: Modeling Quantum Computation in an Array Programming Language

Santiago Núñez-Corrales
NCSA/IQUIST
UIUC
Urbana IL, United States
nunezco2@illinois.edu

Marcos Frenkel
NCSA
UIUC
Urbana IL, United States
marcosf2@illinois.edu

Bruno Abreu
NCSA
UIUC
Urbana IL, United States
babreu@illinois.edu

2023 IEEE International Conference on Quantum Computing and Engineering (QCE) | 979-8-3503-4323-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/QCE57702.2023.00114

***Abstract*—Most contemporary quantum programming languages describe computation as circuits, using a host classical counterpart to drive the execution of quantum programs. However, the circuit model adds expensive complexity to quantum algorithmic development and decreases the transparency of connections between syntax and formal semantics in quantum programs. We argue that producing a high-level quantum programming language without reference to circuits is possible and necessary. We summarize desirable features in future high-level quantum programming languages and provide evidence supporting array programming languages as a natural paradigm for quantum algorithmic expression at the circuit level and beyond. We highlight why APL is a profitable host programming language to attain this goal progressively. In particular, we demonstrate how features provided by APL, such as native support of complex numbers and matrix operations, naturally capture quantum operations while bringing a less cluttered syntax that encodes and encapsulates the linear character of quantum circuit execution. We discuss implementation details of quAPL, an APL library for quantum circuit specification, simulation, and execution intended to provide a gradual ramp toward developing composable procedural abstractions. Finally, we discuss the broader implications of our work and the next steps in our research program.**

***Index Terms*—APL, array programming languages, quAPL, quantum computing, quantum programming**

I. Introduction

Programming languages are, primarily, notational tools of thought [1]. As computing technologies evolve, programming practices drive language evolution toward an equilibrium between two competing forces: *broadening generality* –the ability to express a wider range of problems- and *increasing complexity* –the growth in diversity of vocabulary and grammar available to software developers. Both are modulated by the theoretical richness and availability of abstraction and sophistication brought forth by hardware advances. Research and practice, with time, separate languages into those with greater affinity for algorithmic expression and those where proximity to the hardware facilitates program verification and optimization. As the specification of a programming language stabilizes through repeated use, exploring new problems often translates into extending that language or developing new languages at higher levels.

Mounting evidence [2]–[4] indicates that quantum programming languages have not yet followed such an evolutionary trend. Despite the rapid increase in the number of quantum programming languages available today, most utilize a host programming language to allow the specification of quantum circuits. These specifications are then transpiled into some intermediate form –e.g., OpenQASM [5], QIR [6], MLIR [7]- that software stacks provided by various hardware vendors can understand, optimize (possibly at the pulse level), and execute. The host language then serves the function of a program driver, interfacing between classical and quantum entities and operations. Python [8], C++ [9] and Haskell [10] are notable examples.

Several elements differentiate the evolutionary trajectories of quantum and classical programming languages so far. First, quantum computers possess a vastly larger set of exploitable physical resources than classical ones [11]. The intuitiveness in the use of classical resources seems not to be present in their quantum counterparts, to a large extent due to very limited phenomenological experience with quantum logic. While the complexity of classical algorithms is measured in terms of space and (discrete) time, quantum algorithms also must account for their use of superposition, coherence, interference, and entanglement. Second, the fundamental constructs provided by classical Turing machines, random access machines, and lambda calculus translate to abstract functions over certain algebraic number fields; in contrast, their quantum counterparts [12]–[16] make explicit references to states and circuit elements, thereby still constraining program expressiveness to gates and registers for the most part. Third, the problem of building fault-tolerant quantum hardware is yet to be solved, albeit several platforms have demonstrated increased numbers of usable qubits [17]: NISQ architectures force mixing low-level concerns such as quantum error correction with algorithm design and implementation, which are conceptually separate matters. Recent work simulating fault-tolerant versions of variational quantum eigensolvers [18] also suggests that post-NISQ algorithms will likely admit adaptive versions, or

979-8-3503-4323-6/23/$31.00 ©2023 IEEE
DOI 10.1109/QCE57702.2023.00114

even replacement by other fault-tolerant methods. Fourth, as this number grows, the complexity of circuit building will exceed human specification capabilities.

We report in this article preliminary work tackling these four problems simultaneously through the principled development of a quantum programming language library, **quAPL**. We seek to minimally capture the semantics of quantum circuits to then abstract them away into *quantum motifs* that make little to no reference to circuit elements. Our guiding principle is that *quantum circuits distract from actual problem solving*, since –as their classical counterparts- their expressiveness scales poorly with problem size and thus precludes an intellectually profitable level of expression. In short, we hypothesize that finding an appropriate host programming language can significantly facilitate the transition from circuits to composable abstractions, leading progressively to a more abstract quantum programming language that makes little to no reference to the underlying hardware. By *appropriate*, we mean being capable of naturally revealing the universality and executability of new and existing constructs found across quantum programs, encapsulating complexity, or exposing it as close to the mathematical definitions as needed. To achieve this, we have found in APL –a historical representative for array programming languages- a choice that suits the aforementioned needs.

The structure of this article is as follows. We first enumerate the requirements of a high-level quantum programming language and how these should percolate across a putative software development stack down to the level of circuit specification. Next, we analyze the features of APL [19] that make it an ideal host language to produce libraries starting at the circuit level leading to new abstractions going upwards or producing code in some intermediate representation executable on real quantum hardware. We then describe the features of *quAPL* from states, gates, measurements, and circuits up to three simple examples: quantum random number generation, and the Deutsch-Jozsa and quantum teleportation algorithms.

II. Requirements for High-level Quantum Programming Languages

The primary purpose of a high-level language for any class of computations is to capture precisely and expressively the intended meaning of programs [20]. Meaning, in the context of programming languages, pertains to the study of formal semantics in three main varieties: denotational, operational, and axiomatic. Denotational semantics maps syntactic program entities and their transformations to a set of abstract mathematical entities. Operational semantics describes how syntactic program constructs are evaluated and executed in some abstract machine. Axiomatic semantics formalizes logic constraints used to locate (or rather *fix*) the intended meaning of a program and verify its correctness; this type of semantics has proven substantially harder to achieve. Pragmatically, a programming language should facilitate solving problems with computers. This section addresses both types of requirements, recognizing from the start that quantum computation models differ significantly from digital ones; analogies between both must be performed carefully.

A. Formal requirements

In the case of classical programming languages, all types of semantics rest upon a solid understanding of computation as functions on a finite arithmetic model later mapped onto Turing machines, random access machines, or abstract λ-calculus machines. Materializing these abstract computing devices into actual hardware –simplifying out historical details- resulted from the tripartite match involving how Boolean algebra satisfied the representational requirements of finite arithmetic models and how bistable electronic elements could be used to implement Boolean circuits; Boolean algebra itself arose as a consequence of intense meta-mathematical work.

Performing the same kind of work to create new quantum programming languages as intended here departs from much more austere, scant, and still unclear foundations. One may be tempted to take the view that quantum information is at the core of quantum mechanics [21] and, therefore, that it provides a denotational semantics of quantum computation. While quantum information specifies constraints for quantum resources present in devices implementing computation, it shares the same limitations with classical information theory preventing it from becoming an adequate foundation for algorithmic expression: they do not naturally produce composable abstractions readily interpretable as computation.

Clifford algebras [22] and spinor formulations [23] suffer from a similar limitation when constrained to the description of states and gates. However, their direct connection to the abstract spaces underlying quantum field theory [24], the fact that abstract geometrical algebra machines with substantial computational power can be specified [25], and the existence of symmetries in connection to quantum automata derived from discrete quantum walks [26] that could lead to high-level combinators of the sort found in functional languages [27] suggests they can play a substantial role in the denotational semantics of quantum programs if streamlined. Higher algebra formulations of quantum field theory constitute much stronger candidates in this direction [28]–[30] due to the connection between computation and processes [31]. Determining an adequate abstract formulation for the denotational semantics of quantum programs remains an open problem. Linear logic and categorical semantics have already led to uncovering denotational semantics of quantum lambda calculus [32].

Regarding operational semantics, we remain limited again by the circuit model of quantum computation and, in consequence, by the lack of an abstract quantum machine operating at a higher-level of abstraction [33]. From the initial formulations of quantum Turing machines [12], [34]

to the most recent QRAMs [14], [16] and quantum lambda calculi [13], [15], gates pervade their formal description. Classical RAM and functional abstract machines, on the contrary, are specified in terms of instructions or functions applied to entities belonging to a given finite field instead of concerning themselves with the manipulation of bits through gates. These programming constructs are composable procedural abstractions in the sense that (a) compositionality ensures that all possible programs executable on equivalent abstract machines can be written using a finite number of these abstractions, (b) no reference to specific hardware properties is needed to understand the execution of instructions (although one may construct and enact hardware models through them), and (c) these abstractions provide actionable descriptions of how they harness the resources available for computation. Except for Silq [35], Cavy [36], and Proto-Quipper [37], existing quantum programming languages and frameworks only encapsulate and synthesize circuits depending on hardware capabilities rather than departing from composable procedural abstractions and later satisfying their execution at a lower level through circuit synthesis as a separate process. Thus, classical host languages must partially lend their operational semantics, obscuring the process of discovering abstract quantum expressions.

Finally, quantum logic models have quickly emerged in connection to quantum mechanics and computation [38], [39]. As expected, these have begun to provide an axiomatic basis for quantum program correctness at the circuit level [40] and even type safety through automated theorem proving [41]. ZX calculus provides a substantial logic basis with intuitive graphic semantics beyond a device for circuit simplification [42]. Despite these advances, being constrained by the circuit model also hampers the development of high-level logic models required once more sophisticated abstractions lead the way to operational semantics with purely abstract instructions.

Bluntly, the preponderance of the circuit model in the initial formalization of quantum computing –explained by fundamental unknowns in quantum science- has recently become an obstacle to achieving true algorithmic expression as quantum hardware moves from lab experiments to hardware platforms of increasing size and complexity. Drawing general lessons from the evolution of classical computing, it appears that existing quantum programming approaches fail to produce composable procedural abstractions of the sort described here. Hence, the first requirement for quantum programming languages aspiring to achieve high-level algorithmic expressiveness is to help clarify quantum program semantics by (a) identifying suitable candidate abstract theories that fulfill denotational semantic requirements, (b) producing composable procedural abstractions as either instructions or functions for a new class of abstract machine that makes no reference to quantum circuits, thus separating computation in the abstract domain identified prior from hardware that satisfies the requirements of instructions, and (c) prompting the development of new logics ensuring program correctness and validity.

At the same time, and given the substantial investment in the circuit model and the state of quantum hardware, pursuing such a research program still necessitates modeling quantum circuits without making them the center of attention. Thus, a second overarching requirement is to keep the intellectual effort devoted to circuit building at a minimum by providing abstractions as early as possible by means of language tools that quickly encapsulate recurrent patterns, *quantum motifs*. When combined appropriately, features of circuits blur and signatures of composable procedural abstractions should emerge naturally from them. This approach substantially constrains the space of possible host programming languages to those whose syntax has a small footprint, those including combinators, and those where functions and operators are separate entities. A bonus includes host programming languages where the syntactic cost of expressing facts about states and operators in Hilbert spaces is as low as possible (i.e., native support of complex vector and matrix arithmetic).

B. Pragmatic requirements

Quantum programming languages are expected to facilitate problem-solving in the presence of computing platforms bearing quantum resources. The desirable features for programming notation laid out by Kenneth Iverson in his 1979 Turing Award Lecture [1] provide a surprisingly illuminating list for the current state of affairs in quantum computing. We reinterpret them here in the context of quantum programming languages.

1) Ease of expressing constructs arising in problems: A high-level quantum programming language will lend itself to conveniently express patterns of computation found across a wide variety of problems, whether these arise as a consequence of analysis, generalization, or specialization. We must exercise care in giving preference to combinators, whether those provided by the host language or new ones, due to the limited number of algorithms and thus an equally limited number of known quantum motifs so far; we remind ourselves of the significantly larger set of resources present in quantum computers in contrast to classical counterparts whose expressiveness we have just started to explore. Ideally, the use of high-level expressions for circuit descriptions must produce new motifs that, when composed in clever ways, can become instructions in a language at a higher level.

2) Suggestivity: A significant goal for quantum programming is to suggest motif reuse across different programs that solve similar problems. Even more successfully, language users should recognize a common motif after using the abstraction capabilities present in the language. This association is critical to connecting problems and sub-problems, not just programs. A more challenging and rewarding situation is when a problem cannot be (at least

partially) covered by existing motifs, but its organization suggests a quantum solution. In such cases, creating new motifs is likely connected to fundamental properties of quantum computation enabled by new uses of quantum resources. We want to enable programmers to expand the vocabulary of motifs, which we expect to produce new instructions in a QRAM model with no reference to circuits. Substantial gains can be reaped with these motifs give rise to new combinators or operators.

3) Subordination of detail: The syntax of a quantum programming language should translate the separation of responsibilities into procedural clarity, emphasizing the effects of the computation over the details of how they are achieved; this goes beyond existing work on quantum program modules [43]. For instance, phase kickback connects higher-order interference to reversible, controlled phase transformations [44]. Accessing higher-order phases depends, in turn, on properties such as causality, purification, and strong symmetry; these details are subordinate and should not be directly exposed in order to make use of a generalized phase kickback. A similar detail occurs with the motif of controlled rotation R_n matrices in the Quantum Fourier Transform: we are interested in the periodicities they introduce across a given state, but the details involved are not directly interesting to program builders once a verified implementation exists. Subordination of detail does not translate to obscuring underlying mechanisms: they must remain accessible, only at a different level of visibility.

4) Economy: The utility of a quantum programming language is directly proportional to the number of problems that can be tackled and inversely proportional to the number of syntactic constructs required to write programs for them. With a more extensive set of computational resources, the number of constructs required to express quantum computation is poised to grow. Hence, deliberate choices should be made to ensure a lean syntax from circuits to composable procedural abstractions toward a quantum language with sufficiently new and distinguishable denotational semantics. In classical computing, functional languages possess substantially simpler syntactic constructions. Array programming languages go a step further by providing entire algorithms that operate on a small number of classes of data structures.

5) Amenability to Formal Proofs: Finally, correctness and verification guarantees are necessary to ensure programmer productivity and maximize useful computing time. While quantum algorithm verifiers at the circuit level exist, the types of errors these can identify in current quantum programs is limited. Having composable procedural abstractions will bring new syntactic and semantic expressiveness, with the caveat that greater expressive power often comes with limits about what can be proved correct.

III. Array Programming Languages for Quantum Computing

We wish to suggest that array programming languages constitute the most appropriate hosts for quantum computation, particularly those which support combinators and have a strong functional orientation. Several facts support our hypothesis. First, vector and matrix arithmetic rest at the core of their denotational semantics. Second, many of these languages –or language libraries- provide native support for complex numbers, enabling a transparent manipulation of Hilbert spaces. Third, our search for syntactic economy narrows the selection to Iversonian programming languages in which characters –i.e., *glyphs*- encode entire algorithms of frequent use directly connected to mathematical entities. Of these, APL [19], J[1] and BQN[2] provide the most succinct syntax, yet the communities around J and BQN are substantially less developed than that of APL. Fourth, the need for combinators further constrains the search within APL dialects to Dyalog APL[3], despite not being a fully functional language as BQN. Figure 1 conveniently depicts the relations between the programming languages mentioned above.

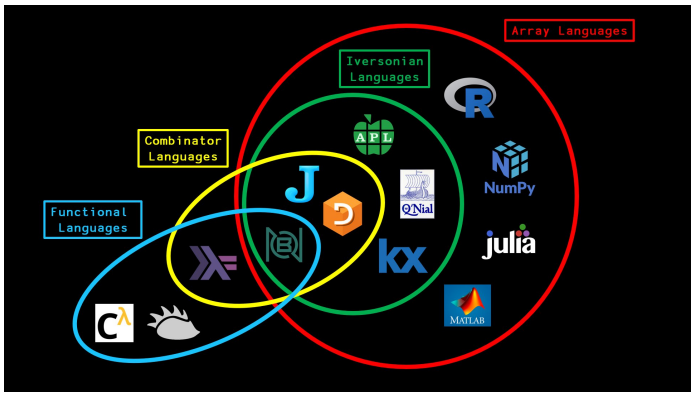

Figure 1. Array programming language Venn diagram. See https://twitter.com/code_report/status/1569808096654163969/photo/1.

APL programming practice bears a rich and informative history in our search for an expressive and economic host language for quantum computation. APL started as an attempt by Kenneth Iverson to regularize existing mathematical notation and rapidly found a place in the description of finite sequential processes [45]. Iverson's work rapidly evolved into a programming language [46] with defining the impact of formal systems design [47] –at the time microarchitecture driven-, manifesting in its application to the design of the IBM SYSTEM/360 [48]. At the same time, APL informed programming language design theory early on [49], bringing to the forefront the role of dualities and identities in program execution, becoming a programming language for applications as well

[1]See: https://www.jsoftware.com/#/README.
[2]See: https://mlochbaum.github.io/BQN/.
[3]See: https://dyalog.com/.

[50]. Part of the evolution of the language resulted in the distinction between functions and operators, paving the way for higher-order constructs of substantial sophistication [51] interpretable as spanning a calculus of operators [52] whose operational semantics leads to tacit expressions with right-to-left execution [53]. Despite decreased use and interest spanning two decades, recent advances in compiler techniques and GPU availability [54] have made it an attractive language for various tasks, including implementing neural network models [55], [56]. A comprehensive contemporary review of APL can be found in [19].

It has not escaped our notice that the account we have provided of the evolution of APL directly suggests a strong parallel to the evolution of hardware and software observed in quantum computing.

IV. quAPL: a Quantum Language Library

We present the features of **quAPL**, the first stage in our journey to clarify open questions about quantum computation and simultaneously bring greater intellectual productivity to quantum programming. The language at the current stage is implemented as a library aiming to fully capture the semantics of quantum circuits with a code basis as minimal as possible, maximizing the use of constructs already present in Dyalog APL. In the exposition below, we have removed the namespace structure to improve readability. A preliminary, experimental version of the code is available on GitHub[4].

Prior to delving into elements of the library itself, a few preliminary observations are necessary to understand the design choices behind **quAPL**. In particular, we must explain why it remains conspicuously close to quantum circuits despite our prior discussion, with the apparent disadvantage of being symbolically heavy for new users.

Most quantum software stacks, when compared to classical ones, appear to have some misplaced elements. For instance, the role OpenQASM, QIR and MLIR play is properly that of hardware specification language instead of specifying how instructions execute in a machine whose architectural components are implementable satisfiable by many possible circuits, none of which should be exposed to end users. Quantum compilation [57], the process of translating quantum logic elements to actual hardware, is substantially closer to a mixture between signal-level hardware control, digital design synthesis in FPGAs and nanoprogramming [58] than to compilation in classical systems, in which a high-level language is translated into a machine language agnostic to physical details. Since we have no quantum abstract machine with instructions, there is no analogue to microprogramming yet.

It is tempting to argue that thinking about higher architectural details is premature, or that there is no *a priori* reason why a similar hierarchy should emerge in quantum computing and that the current abstraction frameworks provide will suffice. Our view is as follows. Classical and quantum Turing machines are equivalent in their Turing-computability, thus making it possible to think in terms of quantum automata of various kinds [59] and, by extension, their corresponding formal languages. Different formal languages have different expressive power. Programming languages are formal languages, and different ones which specify virtual machines arranged in a hierarchy of simulatability resulting from their differences in expressive power [60]. As an example, multiple assembly languages can be implemented at the microprogramming level, and multiple high-level languages compile to the same assembly code. Some of these virtual machines will be implemented closer to the hardware, others to the software, and others will mediate between both.

Even when quantum mechanics introduces new aspects to formal languages, the rise of a hierarchy of quantum virtual machines according to their expressive power sharing properties an a structure similar to that in classical computing should be expected, and even more, welcomed. Substantial separation of responsibilities for hardware and software implementation constitutes the first byproduct of delineating the boundaries between virtual machines and their languages. The second one, most relevant for the purpose behind our research, is the discovery and implementation of composable procedural abstractions described above. Figure 2 places **quAPL** in the larger context of this goal. In consequence, a detailed retrospective look at the evolution of classical computing architectures should help us bypass known mistakes: quantum ontogeny should not have to recapitulate all stages of classical phylogeny, especially the failed ones. Following theoretically-informed prescriptions from the start will, in all likelihood, produce quantum architectures with higher standardization, lower implementation friction, and in a different class of algorithmic expressiveness.

A. APL Crash Course

We provide here the rudiments of APL used in the construction of **quAPL**. The APL syntax is described in detail in [19]. Succinctly, elements in APL belong to two distinct categories: those corresponding to values, and meta-elements. Assignment is represented by ←, and parentheses group expressions as in other languages.

1) Arrays: Multi-dimensional arrays (i.e., tensors) are first-class citizens in APL. They can be specified by extension, using generator functions on integer sequences created using the iota operator (⍳), by creating arrays based on a pre-defined shape (⍴), or by re-shaping an existing array. Complex values are supported by default as array entries, which can be accessed via square bracket notation ([]) or positional access functions.

```
⍝ Arrays by extension
    1 2 3 4 5
1 2 3 4 5
```

[4]See: https://github.com/nunezco2/quAPL.

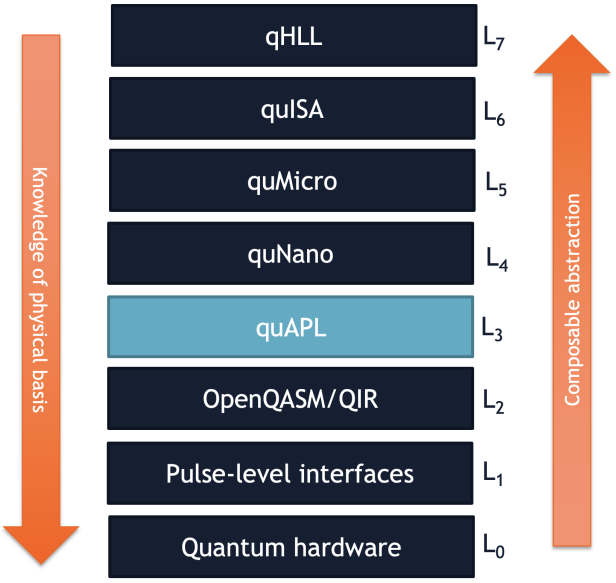

Figure 2. Location of **quAPL** in relation to quantum stack components inferred from the relations between abstract (virtual) machines and the expressive power of languages they enable. Presently, no analogues to classical nano- and microprogramming exist, critical for the emergence of instructions with no reference to logic or hardware elements; we ultimately seek the development of quantum high-level languages. Languages are the bottom specify virtual machines implemented closer to the hardware, while those at the top preferentially build upon composable procedural abstractions.

```
⍝ Arrays through iotas
    ⍳5
1 2 3 4 5

⍝ Arrays through shaping
    3 3 ⍴ 0
0 0 0
0 0 0
0 0 0

⍝ Arrays through shaping an existing vector
    3 3 ⍴ ⍳9
1 2 3
4 5 6
7 8 9
```

2) Functions: Functions operate on arrays. In this exposition, we center our attention on direct (dynamic) functions (i.e., dfns). A function can be *monadic* (i.e., taking a right argument ⍵), dyadic (i.e., taking both a left argument ⍺ and a right argument ⍵) or niladic (i.e., returns a constant value). Pre-defined functions include the usual arithmetic functions (e.g., +, ÷), structural manipulation of arrays (e.g., shape – ⍴, enclose and pick – ⊂, ⊃), manipulation of array contents (e.g., grade up and down – ⍋, ⍒), enclose and pick – ⊂, ⊃), and other system functions. New functions are defined using curly braces (⍺, ⍵), where multiple statements are separated by ⋄. Functions are right-associative to values without establishing individual precedence rules.

```
    ⍝ f(⍺,⍵) = (⍵×5) + ⍺
    2 {⍺ + ⍵×5} 3
17
    ⍝ Compute and reverse 2^i, i=0..⍵
    {⌽2*(0,⍳6)} 6
64 32 16 8 4 2 1

    ⍝ A named function for e^(i⍵)
    phase ← {*0J1×⍵}
    phase 1
0.54J0.841
```

3) Operators: Operators in APL implement higher-order functions in the sense usually found in functional languages such as Haskell and OCaml. These implement structure- or syntactic-dependent computation patterns. Reduction-based patterns (e.g., replicate – /), and combinators (e.g., map and commute – ¨, ⍨) exemplify some of these. Custom operators, both monadic and dyadic can be specified similar to functions by means of ⍺⍺ and ⍵⍵ as left and right parameters, correspondingly. Composition (i.e., .) and outer product (∘) are frequently used in matrix algorithms. Operators are left associative to functions.

```
    ⍝ Inner product between two 1D vectors
    2 4 6 +.× 1 3 5
44

    ⍝ Solve the matrix equation A(⍺)×x = ⍵
    solve ← {(⌹⍺) +.× ⍵}  ⍝ ⌹: matrix inverse
    A ← 2 2 ⍴ 2 1 1 2
    b ← 2 1 ⍴ 1 0  ⍝ Column vector
    A solve b
 0.667
¯0.333

    ⍝ Sum from of 1/2^i, i=0..6
    inv_pow_2 ← {÷2*⍵}
    +/inv_pow_2¨ 0,⍳6
1.98
```

We refer the reader to several concise introductions to APL available online[5].

B. States

Single-qubit basis states $|0\rangle$ (ground) and $|1\rangle$ (excited) can be correspondingly specified **quAPL** by means of the `q0` and `q1` functions

```
q0 ← 2 1 ⍴ 1 0
q1 ← 2 1 ⍴ 0 1
```

which produce column vectors. An arbitrary qubit state can be specified in one of two ways. The first one operates by specifying the complex amplitudes directly and computing the resulting normalized linear superposition

```
qx ← {normalize ⊃+/(q0 q1) ×.+ ⍵}
```

where `normalize` corresponds to

```
normalize ← {⍵÷0.5*⍨(dagger+.×⊢)⍵}
```

[5]See: https://mastering.dyalog.com/README.html

In the preceding code, converting a ket $|\varphi\rangle$ to a bra $\langle\varphi|$ is performed through the **dagger** function

```
dagger ← { ⍉+⍵ }
```

which applies to a quantum vector state of arbitrary length and compactly encodes the well-known mathematical definition. Another way to specify a qubit state is by means of the azimuthal and polar angles θ (**⍵**) and ϕ (**⍺**):

```
bloch ← {(2○(⍵÷2)),(*(0J1×⍺))×(1○(⍵÷2))}
```

These definitions become easily extensible to multiqubit states. To do so, we define the Kronecker product function using tacit programming.

```
kpr ← ⊃ (,/ (⍪⌿ (⊂[3 4] ∘.×)))
```

This definition showcases APL's advantages for quantum computation in two major ways. First, it unveils the Kronercker product as a reorganization of a higher dimensional object, in this case, a 4-dimensional tensor. In effect, its definition through the outer product **∘** reveals underlying connections to geometrical algebra. Second, it is extremely compact, notationally speaking. Defining a 5-qubit register initialized in the ground state can be expressed thus as

```
reg_5 ← q0 kpr q0 kpr q0 kpr q0 kpr q0
```

Fortunately, we can make use of APL's replicate (**/**) alongside enclose (**⊂**) and pick (**⊃**) to avoid unnecessary repetition. The same can be written as

```
reg_5 ← ⊃ kpr/ 5⍴⊂q0
```

With this in mind, one can readily define an addressable quantum register by means of the function

```
reg ← {(⍵ 1⍴(⍳⍵)-1),(⍵ 1⍴⊂q0)}
```

and, conveniently, translate into a state vector by means of *threading* the second column of the register containing a given number of individually initialized qubits on the ground state.

```
thread_reg ← {⊃ kpr/⍵[;2]}
```

Finally, we also provide two functions that facilitate translating between indices and state vectors. This is particularly necessary when applying circuit stages to vector states upon conversion from register notation. The first of such functions is **subregister**, which given a state register and a set of indices obtains only that part of the state. The condition ensures all indices are present in the register, or returns empty (**⍬**) otherwise.

```
subregister ← { ∧/(⍺ ∈ ⍵[;1])=0: ⍬
    (⍺+1)[]⍤⍋∘⊂⍤⍋⊢⍵
}
```

To observe its effect, consider the following 4-qubit register

```
    r4
0  1
   0
1  0
   1
2  1
   0
3  0
   1
```

corresponding to the quantum state $|\varphi\rangle = |0101\rangle = |\varphi_0\varphi_1\varphi_2\varphi_3\rangle$. To select only excited states **(3 1)** –in that order- it suffices to compute

```
      (3 1) subregister r4
3  0
   1
1  0
   1
```

More generally, **subregister** also provides a simple interface to perform qubit permutations to match gate inputs as needed (e.g., controlled gates). A complete state permutation **(3 1 0 2)** is readily realized by

```
      (3 1 0 2) subregister r4
3  0
   1
1  0
   1
0  1
   0
2  1
   0
```

Another common task, particularly when working between the circuit and vector state representations, is to perform a similar permutation on individual qubit indices and have it reflected as a permutation. To achieve this, we note that mapping a permutation from circuits to vector states is equivalent to computing the table of indices for a given number of qubits, reordering the columns resulting from their binary encoding, and finally reinterpreting the values back as integer indices. Take the function

```
tnsidx ← {⍉(⍵ ⍴ 2) ⍉ ((2*⍵) ⍴ (⍳(2*⍵)) - 1)}
```

and apply it, for instance, to the state $|\varphi_0\varphi_1\varphi_2\rangle$. The corresponding 2^3 indices for vector state entries in a 3-qubit system (in binary) are

```
    tnsidx 3
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Reordering the state into, say, $|\varphi_2\varphi_0\varphi_1\rangle$ changes the table above as

```
0 0 0
1 0 0
0 0 1
1 0 1
0 1 0
1 1 0
0 1 1
1 1 1
```

which can then be interpreted into integer values by means of decode (⊥) with the function specified below.

```
qrdtotrd←{ 1 + 2⊥⍉ω[⍋[2]⍤∘⊂⍤tnsidx α}
```

We observe that, despite the small number of primitives, **quAPL** conveniently captures vector states in Hilbert spaces of arbitrary finite dimension.

C. Gates

Quantum gates are naturally specified in **quAPL** as complex matrices of size 2^n for n qubits. We classify gates in two main dimensions: depending on whether these are parametric or not, and depending on whether they are atomic or composite. Non-parametric gates are implemented through numerical matrices directly, such as the Fredkin gate,

```
      FRK
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
```

while parametric gates correspond to functions that receive parameters that modulate the definition of the physical operator being implemented. For instance, the $XX(\phi)$ interaction gate with $\phi = \pi/3$ becomes

```
      XX ○÷3
0.866       0           0           0.000J¯0.5
0           0.866       0.000J¯0.5  0
0           0.000J¯0.5  0.866       0
0.000J¯0.5  0           0           0.866
```

Atomic gates can be transparently combined into larger (multiquibit) gates using the Kronecker product directly. Creating a 2-qubit Hadamard gate can be achieved by

```
      ⊃ kpr/ 2ρ⊂H
0.5  0.5  0.5  0.5
0.5 ¯0.5  0.5 ¯0.5
0.5  0.5 ¯0.5 ¯0.5
0.5 ¯0.5 ¯0.5  0.5
```

which then can be encoded as a gate-dependent idiom or *motif* templated by the dimensionality of a vector state, thus generalizing it to an arbitrary number of qubits to store it for repeated use in a quantum algorithm. Then, defining

```
template ← {⊃ kpr/ (1⌷2⊛(ρα)) ρ⊂ω}
```

leads to a generalized superposition instruction with initial state vector `iv`

```
superpose ← {
    stage ← ω template H
    stage ω
}
```

that makes abstract, not concrete reference to state vectors regardless of their dimension. This generalization applies, for instance, to creating controlled gates with an arbitrary number of control qubits. Consider the controlled phase gates R_n that appear in the Quantum Fourier Transform, which can be implemented by the function

```
Rn ← {1∘gCTR P (2×○÷2*ω)}
```

It now becomes possible to easily generate the sequence $R_2, \cdots R_n$ gates required for the QFT algorithm by constructing a function that uses the dimension of the vector state in combination with the map function (¨). The resulting function for a vector state `vs` becomes

```
qft_cphase_set ← Rn¨ (1 ↓ ⍳ 2 ⊛ 1⌷⍴ vs)
```

Note, once more, that producing the corresponding set of controlled phase gates becomes a motif itself. The function makes use of the traditional (idiomatic) `ID` function to then embed the unitary gate into an adequate control structure spanned by `α` qubits,

```
gCTR ← {
    (n _) ← ⍴ ω
    ID ← {ω ω ⍴ 1, ω⍴0}
    gate ← ID 2*(α + 2⊛n)
    ((-⍴ω)↑gate) ← ω
    gate
}
```

a function that can be used to generate arbitrary control structures, essential to patterns of uncomputation. We preserve standard controlled gates to avoid unnecessary computation, but can readily verify the correctness of implementation by testing that

```
    CNOT ≡ 1 gCTR X
1
```

and

```
    FRK ≡ 1 gCTR SWAP
1
```

for instance.

D. Circuits

Circuits are implemented in **quAPL** by means of stages that are evaluated right-to-left, in contraposition to circuit diagrams that follow left-to-right conventions. A stage constitutes a function that takes a collection of indices `idx`, a collection of gates `gtx` and applies them to a vector state `vs` to obtain a new vector state `nvs`.

```
nvs ← (idx gtx) stage vs
```

The number and sequence of indices must match the number of inputs for gates in **gtx**. For instance, in

```
st_1 ← ((0 1 2)) (H H H))∘stage
st_2 ← ((2 0) (CNOT))∘stage
st_3 ← ((3) (Rx (○÷16)∘stage
```

the stage **st_1** defines a stage in which three Hadamard gates are applied to a vector state. We have used jot (**∘**) to bind the data structure and curry the function and ensure it is monadic and right-associative. We note in **st_2** that only two-qubit indices have been specified for **CNOT**: qubit 2 controls the outcome of qubit 0. To ensure greater generality, each stage uses its vector state input to calculate the dimension of the corresponding Hilbert space and add identity gates (**I**) to unspecified qubits. This is an essential step to remove unnecessary hardware details from the description of quantum programs and achieve generality. In **st_3**, we implement $R_x(\pi/16)$ on qubit 3. Using the power operator in APL, we can define a new rotation in x with k $\pi/16$ increments and compute $R_x(5\pi/16)$

```
Rx16_unit ← ((3) (Rx ○÷16))∘stage
Rx16_k ← {(Rx16⍣⍺) ⍵}
nvs ← 5 Rx16k vs
```

Thanks to right-to-left associativity of function application in APL and tacit (i.e. fixed-point) expressions, a full circuit with all three stages becomes

```
circuit ← st_3 st_2 st_1
nvs ← circuit nv
```

Finally, we provide a single measurement function **measure** that collapses the state of a qubit (or group of qubits) upon usage. Measurements are treated as special circuit stages, yet information is not directly converted to classical bits. We make use of **qrdtotrd** to implement the addressable qubit measurement effects in qubit measurements. When used in monadic, right-associative form, **measure** collapses all qubits in the register. When a list of qubit indices is supplied to the left argument, the corresponding partial measurement is performed. We assume non-demolition measurements are available.

E. Examples

We present below three paradigmatic cases that exemplify the use of the facilities developed above. While small, these cases contain measurement, superposition, and entanglement, essential across quantum algorithms. In the first two cases, we explore the current capabilities and limitations of APL and suggest extensions. For the third case, we show directly the possibilities with both existing primitives and with the proposed extensions to **quAPL**.

1) Quantum random deviates: Generating quantum deviates requires generating a uniform superposition across a quantum register of size n initialized at ground level and then performing a full state measurement. Given that this case is composed of a uniform superposition and a uniform measurement, no specialized circuit stage is required. To convert to a (classical) integer value, we make use of decode (**⊥**). In this example, we generate an 8-bit random integer.

```
vs ← thread_reg reg 8
rand8 ← 2 ⊥ measure superpose vs
```

2) 2-qubit Deutsch-Jozsa: We implement the Deutsch-Jozsa algorithm [61] for the 2-qubit XOR function $f(x_0, x_1) = x_0 \oplus x_1$. We first construct the oracle function using two different stages. Start by defining a 3-qubit register that includes the ancilla,

```
n ← 3
vs ← thread_reg reg n
```

First, note that the resulting circuit contains two CNOT gates where the index of control qubits decreases while using the same ancilla for controlled outcome. That is, qubits 0 to n become controls for qubit $n+1$. Such a pattern is easy to capture (in reverse order to facilitate stage application). Since circuit stages can use the corresponding vector to infer which qubits require identity gates, we can therefore write the qubit mappings for resulting substages as the function

```
stage_ctrls ← {⍵ {(⍵ ⍺) CNOT}¨ ¯1↓((⍳⍵)-1)}
```

which we make use to create the oracle implementing f

```
ctrls ← stage_ctrls 2⍴1⌷⍴ vs
oracle_a ← (1 ⌷ ctrls)∘stage
oracle_b ← (2 ⌷ ctrls)∘stage
oracle ← oracle_b oracle_a
```

We now produce a superposition over all except the ancilla qubits by setting

```
Hna ← (n - 1) template H
sup_na_st ← ((¯1↓(⍳n) - 1) Hna)∘stage
```

and finally, we produce two stages corresponding to the sequences **XH** and **HX** which are ancilla rotations,

```
an_rot_st ← ((n-1) H)∘stage ((n-1) X)∘stage
an_rot_i_st ← ((n-1) X)∘stage ((n-1) H)∘stage
```

The last intermediate step requires assembling matching 'superpose and rotate' blocks

```
an_rot_st ← ((n-1) H)∘stage ((n-1) X)∘stage
an_rot_i_st ← ((n-1) X)∘stage ((n-1) H)∘stage
```

Finally, we can assembly the preparation-unpreparation blocks and perform the final measurement on all but the ancilla qubit

```
prep ← an_rot_st sup_na_st
unprep ← sup_na_st an_rot_i_st
```

```
vs ← (¯1↓(⍳n)-1) measure unprepare
       oracle prepare vs
```

The code above is lacking in terms of expressive economy and readability. While the code contains a substantial number of instances where symmetry is mapped resulting in lists of curried functions and sequential right-to-left function applications, these facts are not exploited fully. This use case suggests the need for left and right maps (⍤) to admit beside and bind (∘) for algorithmically producing partial functions, an extension to most contemporary APL implementation. Additionally, having a new compositional application operator

```
apply ← { ⍺⍺ ⍵⍵ ⍵}
```

for two functions `f` and `g` as well as a second-order operator interpretation of reductions (represented here by `//`) using it. Supposing for a moment that such features are available, the code could then be stated in terms of lists of partial functions. We can even become more succinct by explicitly leaving stage separation to the end.

```
n ← 3
vs ← thread_reg reg n

stage_ctrls ← {⍵ {(⍵ ⍺) CNOT}⍤ ¯1↓((⍳⍵)-1)}
oracle ← stage_ctrls 2⊛1⌷⍴ vs

Hna ← (n - 1) template H
prep ←  (
            ((n-1) H)
            ((n-1) X)
            ((¯1↓(⍳n)-1) Hna)
        )

DJ ← apply // ⊃,/((⍣¯1prep) oracle prep)∘⍤stage
measure_but ← {(¯1↓(⍳⍺)-1)∘measure ⍵}
nvs ← (¯1↓(⍳n)-1) measure_but DJ vs
```

Note that (a) we have separated measurement from the core of the Deutsch-Jozsa algorithm which now acquires the form of a kernel, (b) that the symmetries in the circuit become clear through the application of reverse (⍣¯1), and (c) that we now have lists of curried functions combined into a single function by means of fixed point notation.

3) Quantum teleportation: Now, using all the above, we will produce code for Bennett's quantum teleportation algorithm [62]. Note that the first two stages of the circuit (Hadamard gate followed by a CNOT gate) are reversed and translated one qubit upwards in the circuit. This suggests the function

```
translate ← {
    s ← ⍵
    (1⌷s) ← ⍺ + (1⌷s)
    s
}
```

which then leads to the full solution below.

```
n ← 3
vs ← thread_reg reg n

EPR ← (
    ((1 2) CNOT)
    ((1) H)
)

cROT ← (
    ((1 2) (1 gCTR Z))
    ((0 2) (1 gCTR X))
)

QTel ←  cROT∘stage measure_but
        apply // ⊃,/(
            (⍣¯1 translate EPR)
            EPR
        )∘⍤stage

nvs ← QTel vs
```

V. Conclusions

The ability to productively construct programs at an adequate level of abstraction is an unavoidable precondition for the continued success of any computing technology. One of the consequences of higher-level languages, as put by Alan J. Perlis in his Foreword to the Structure and Interpretation of Computer Programs [63], is to create a semantic context where "*a programmer should acquire good algorithms and idioms.*" In its current form, the circuit model for quantum programmers makes existing algorithms poorly accessible, new ones hard to envision, and idioms somewhat of an impossibility. We believe that the success of quantum computing is intricately tied to how programming language design will facilitate algorithmic expression, and how the formal properties of resulting languages provide guarantees that prevent unproductive or incorrect software development paths. Also, contributing to the context in which new standards for quantum hardware and software arise constitutes a major motivation in quantum hardware-software co-design; current quantum programming trends are still driven by the diverse offerings of quantum platform vendors, limiting scalability of adoption. The prevailing mismatch between the increasing sophistication of quantum hardware platforms, our formal understanding of what quantum computation *means* at a level above circuits, and the persistence of the circuit model underscores the relevance of our work.

In this manuscript, we presented compelling evidence in favor of array programming languages with APL as the alternative satisfying theoretical and pragmatic needs arising in quantum programming. We described two different kinds of demands on future high-level programming languages: (a) formal demands that, when satisfied, reveal deeper theoretical semantics beyond qubits and gates, and (b) pragmatic demands to ensure broader accessibility and productivity of quantum programming. The intent of our work resonates with Marvin Minsky's assertion that "*programming is a good medium for expressing poorly understood and sloppily formulated ideas*" [64]: building programming languages more prescriptively may prove

useful to unveil the extent of what doing quantum computation means in more abstract terms in the first place, and then how to harness it to solve practical and exciting problems. For the pragmatic demands, Iverson's Turing Award Lecture seems prescient of our needs and expectations for better means of algorithmic expression in quantum computing. Our preliminary work on **quAPL** demonstrates a promising exploratory path in this sense.

Our future work will concentrate on four main tasks: (a) complete and ensure the robustness of **quAPL** for the generalized description of quantum circuits, (b) implement known quantum algorithms directly, (c) aggressively abstract common computation patterns across these algorithms to construct a new library of *quantum motifs*, (d) systematically search for arrangements of quantum motifs capable of giving rise to composable procedural abstractions in the form of instructions for an abstract quantum machine that makes no reference to the circuit model, and (e) possibly build a new programming language on top of it. In parallel, we plan to implement additional functionality to connect our research with existing quantum hardware platforms. One of them is transpiling from **quAPL** to OpenQASM and QIR intermediate representation. Another project is exploring the construction of realistic hardware simulators with various noise models where **quAPL** can run natively. By implementing increasingly large quantum algorithms and comparing the resulting code with other programming frameworks, we will be able to draw relevant comparisons beyond the simple examples explored here. Our goal is two-fold regarding software implementations: pushing the expressive range of APL to test its adequacy across the quantum computing stack, and at all points maintaining a code base as small as possible. Due to the inherent complexity of the research problems found in quantum computing, care must be exercised to avoid it on the software front at all costs.

As with any application of existing technologies to new kinds of problems, we believe the code and resulting examples shown here illustrate some of the tensions and resolutions quantum computing will induce on programming languages in general, and array programming languages specifically as they serve as host mediating the interplay between classical and quantum computing resources. For APL in particular, we foresee the strategic introduction of broader functional programming semantics, extending the applicability of existing operators to fulfill higher-order specifications, and conservatively introducing new glyphs that encode purely quantum composable procedural abstractions. Substantially more research and practice are needed in this direction.

Acknowledgment

This work was partially funded by the Illinois Quantum Applications Program, supported by NSF #2016136, the IBM-Illinois Discovery Accelerator Institute, and the National Center for Supercomputing Applications via the Health Care Innovation Team, University of Illinois Urbana-Champaign. We wish to thank Patrick Snyder and Brian de Marco at the Illinois Quantum Science and Technology Center for their support, and Ulises Agüero-Arroyo at Universidad Cenfotec (Costa Rica) for offering his expert views on the potential impact of nano and micro programming on future quantum architectures. We also thank Richard Park, Aaron Hsu, and Morten Kromberg from Dyalog Ltd for their valuable interactions regarding Dyalog APL, and particularly to Adám Brudzewsky for his expert suggestions via the APL Orchard, yet any code errors and inaccuracies remain our own. We declare no competing interest. Finally, we thank our reviewers for sharp, probing questions that raised the quality of our manuscript substantially.

References

[1] K. E. Iverson, "Notation as a tool of thought," in *ACM Turing Award Lectures*, 2007, p. 1979.

[2] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore, "Quantum programming languages," *Nature Reviews Physics*, vol. 2, no. 12, pp. 709–722, 2020.

[3] S. Garhwal, M. Ghorani, and A. Ahmad, "Quantum programming language: A systematic review of research topic and top cited languages," *Archives of Computational Methods in Engineering*, vol. 28, pp. 289–310, 2021.

[4] S. S. Gill, A. Kumar, H. Singh, M. Singh, K. Kaur, M. Usman, and R. Buyya, "Quantum computing: A taxonomy, systematic review and future directions," *Software: Practice and Experience*, vol. 52, no. 1, pp. 66–114, 2022.

[5] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta *et al.*, "OpenQASM 3: A broader and deeper quantum assembly language," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–50, 2022.

[6] J. Luo and J. Zhao, "Formalization of quantum intermediate representations for code safety," *arXiv preprint arXiv:2303.14500*, 2023.

[7] A. McCaskey and T. Nguyen, "A MLIR dialect for quantum assembly languages," in *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2021, pp. 255–264.

[8] A. Cross, "The IBM Q experience and QISKit open-source quantum computing software," in *APS March meeting abstracts*, vol. 2018, 2018, pp. L58–003.

[9] S. Stanwyck, H. Bayraktar, and T. Costa, "cuquantum: Accelerating quantum circuit simulation on GPUs," in *APS March Meeting Abstracts*, vol. 2022, 2022, pp. Q36–002.

[10] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: a scalable quantum programming language," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 333–342.

[11] E. Chitambar and G. Gour, "Quantum resource theories," *Reviews of modern physics*, vol. 91, no. 2, p. 025001, 2019.

[12] D. Deutsch, "Quantum theory, the church–turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.

[13] P. Selinger, B. Valiron *et al.*, "Quantum lambda calculus," *Semantic techniques in quantum computation*, pp. 135–172, 2009.

[14] S. Guerrini, S. Martini, and A. Masini, "Quantum turing machines: computations and measurements," *Applied Sciences*, vol. 10, no. 16, p. 5551, 2020.

[15] N. Botö and F. Forslund, "The zeta calculus," *arXiv preprint arXiv:2303.17399*, 2023.

[16] Q. Wang and M. Ying, "Quantum random access stored-program machines," *Journal of Computer and System Sciences*, vol. 131, pp. 13–63, 2023.

[17] C. Q. Choi, "Ibm's quantum leap: The company will take quantum tech past the 1,000-qubit mark in 2023," *IEEE Spectrum*, vol. 60, no. 1, pp. 46–47, 2023.

[18] H. Sayginel, F. Jamet, A. Agarwal, D. E. Browne, and I. Rungger, "A fault-tolerant variational quantum algorithm with limited t-depth," *arXiv preprint arXiv:2303.04491*, 2023.

[19] R. K. Hui and M. J. Kromberg, "APL since 1978," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–108, 2020.

[20] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[21] J. Bub, "Quantum mechanics is about quantum information," *Foundations of Physics*, vol. 35, no. 4, pp. 541–560, 2005.

[22] J. Hrdina, A. Návrat, and P. Vašík, "Quantum computing based on complex clifford algebras," *Quantum Information Processing*, vol. 21, no. 9, p. 310, 2022.

[23] M. A. Trindade, S. Floquet, and J. D. M. Vianna, "A general formulation based on algebraic spinors for the quantum computation," *International Journal of Geometric Methods in Modern Physics*, vol. 17, no. 14, p. 2050206, 2020.

[24] J. Baugh, D. R. Finkelstein, A. Galiautdinov, and H. Saller, "Clifford algebra as quantum language," *Journal of Mathematical Physics*, vol. 42, no. 4, pp. 1489–1500, 2001.

[25] R. Schott and G. Stacey Staples, "Reductions in computational complexity using Clifford algebras," *Advances in applied Clifford algebras*, vol. 20, pp. 121–140, 2010.

[26] P. Arnault, "Clifford algebra from quantum automata and unitary wilson fermions," *Physical Review A*, vol. 106, no. 1, p. 012201, 2022.

[27] C. Hoekstra, "Combinatory logic and combinators in array languages," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, 2022, pp. 46–57.

[28] H. Halvorson and M. Müger, "Algebraic quantum field theory," *arXiv preprint math-ph/0602036*, 2006.

[29] G. Basti, A. Capolupo, and G. Vitiello, "Quantum field theory and coalgebraic logic in theoretical computer science," *Progress in Biophysics and Molecular Biology*, vol. 130, pp. 39–52, 2017.

[30] M. Benini, A. Schenkel, and L. Woike, "Homotopy theory of algebraic quantum field theories," *Letters in Mathematical Physics*, vol. 109, no. 7, pp. 1487–1532, 2019.

[31] J. Baez and M. Stay, *Physics, topology, logic and computation: a Rosetta Stone*. Springer, 2011.

[32] A. van Tonder and M. Dorca, "Quantum computation, categorical semantics and linear logic," *arXiv preprint quant-ph/0312174*, 2003.

[33] S. Núñez-Corrales, "Quantum abstract machines without circuits: the need for higher algorithmic expressiveness," 2023.

[34] R. P. Feynman, "Quantum mechanical computers." *Found. Phys.*, vol. 16, no. 6, pp. 507–532, 1986.

[35] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 286–300.

[36] C. M. McNally, "Practical modern quantum programming," Ph.D. dissertation, Massachusetts Institute of Technology, 2021.

[37] D. Lee, "Formal methods for quantum programming languages," Ph.D. dissertation, Université Paris-Saclay, 2022.

[38] J. M. Dunn, L. S. Moss, and Z. Wang, "Editors' introduction: the third life of quantum logic: quantum logic inspired by quantum computing," *Journal of Philosophical Logic*, vol. 42, pp. 443–459, 2013.

[39] E. Rowell and Z. Wang, "Mathematics of topological quantum computing," *Bulletin of the American Mathematical Society*, vol. 55, no. 2, pp. 183–238, 2018.

[40] J. Bergfeld, *Quantum logics for expressing and proving the correctness of quantum programs*. University of Amsterdam, 2019.

[41] L.-J. Dandy, E. Jeandel, and V. Zamdzhiev, "Type-safe quantum programming in idris," in *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. Springer, 2023, pp. 507–534.

[42] R. Duncan, A. Kissinger, S. Perdrix, and J. Van De Wetering, "Graph-theoretic simplification of quantum circuits with the zx-calculus," *Quantum*, vol. 4, p. 279, 2020.

[43] P. Sánchez and D. Alonso, "On the definition of quantum programming modules," *Applied Sciences*, vol. 11, no. 13, p. 5843, 2021.

[44] C. M. Lee and J. H. Selby, "Generalised phase kick-back: the structure of computational algorithms from physical principles," *New Journal of Physics*, vol. 18, no. 3, p. 033023, 2016.

[45] K. E. Iverson, "The Description of Finite Sequential Processes," in *Proceedings of the Fourth London Symposium on Information Theory*, C. Cherry, Ed., 1960, pp. 447–457.

[46] ——, *A Programming Language*. John Wiley & Sons, Incorporated, 1962.

[47] ——, "Programming notation in systems design," *IBM Systems Journal*, vol. 2, no. 2, pp. 117–128, 1963.

[48] A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A formal description of SYSTEM/360," *IBM Systems Journal*, vol. 3, no. 2, pp. 198–261, 1964.

[49] K. E. Iverson, "Formalism in programming languages," *Communications of the ACM*, vol. 7, no. 2, pp. 80–88, 1964.

[50] A. D. Falkoff and K. E. Iverson, *APL/360: User's manual*. International Business Machines Corporation, 1968.

[51] K. E. Iverson, "Operators," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 161–176, 1979.

[52] K. E. Iverson, R. Pesch, and J. H. Schueler, "An operator calculus," *ACM SIGAPL APL Quote Quad*, vol. 14, no. 4, pp. 213–218, 1984.

[53] R. K. Hui, K. E. Iverson, and E. E. McDonnell, "Tacit definition," *ACM SIGAPL APL Quote Quad*, vol. 21, no. 4, pp. 202–211, 1991.

[54] A. W. Hsu, "Co-dfns: Ancient language, modern compiler," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2014, pp. 62–67.

[55] A. Šinkarovs, R. Bernecky, and S.-B. Scholz, "Convolutional neural networks in apl," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, 2019, pp. 69–79.

[56] A. Hsu and R. Girão-Serrão, "U-net CNN in APL," *International Workshop on Libraries, Languages and Compilers for Array Programming – ARRAY 23. Orlando FL, USA.*, 2023 (accepted).

[57] M. Maronese, L. Moro, L. Rocutto, and E. Prati, "Quantum compiling," in *Quantum Computing Environments*. Springer, 2022, pp. 39–74.

[58] G. F. Casaglia, "Special feature: Nanoprogramming vs. microprogramming," *Computer*, vol. 9, no. 1, pp. 54–58, 1976.

[59] C. Moore and J. P. Crutchfield, "Quantum automata and quantum grammars," *Theoretical Computer Science*, vol. 237, no. 1-2, pp. 275–306, 2000.

[60] M. Felleisen, "On the expressive power of programming languages," *Science of computer programming*, vol. 17, no. 1-3, pp. 35–75, 1991.

[61] R. Jozsa, "Entanglement and quantum computation," *arXiv preprint quant-ph/9707034*, 1997.

[62] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels," *Physical review letters*, vol. 70, no. 13, p. 1895, 1993.

[63] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

[64] M. Minsky, "Why programming is a good medium for expressing poorly understood and sloppily formulated ideas," *Design and planning II-computers in design and communication*, pp. 120–125, 1967.