

QuaFL: A Typed DSL for Quantum Programming

Andrei Lapets Marcus P. da Silva
Mike Thome Aaron Adler Jacob Beal
Raytheon BBN Technologies, 10 Moulton Street,
Cambridge, MA 02138, USA
alapets@bbn.com msilva@bbn.com
mthome@bbn.com aadler@bbn.com
jakebeal@bbn.com

Martin Rötteler
NEC Laboratories America, 4 Independence Way,
Suite 200, Princeton, NJ 08540, USA
mroetteler@nec-labs.com

Abstract

Quantum computers represent a novel kind of programmable hardware with properties and restrictions that are distinct from those of classical computers. We investigate how some existing abstractions and programming language features developed within the programming languages community can be adapted to expose the unique capabilities of quantum computers to programmers while at the same time allowing them to manage the new and unfamiliar constraints of programming a quantum device.

We introduce QuaFL, a statically typed domain-specific programming language for writing high-level definitions of algorithms that can be compiled into logical quantum circuits. The primary purpose of QuaFL is to support programmers in defining high-level yet physically realizable quantum algorithms and in helping them make informed decisions about implementation trade-offs. QuaFL allows programmers to use high-level data structures including integers, fixed point reals, and arrays within quantum algorithms, and to explicitly define superpositions and unitary transformations on data. The QuaFL type system allows programmers to distinguish between classical and quantum portions of a program, uses a variant of linear types and an orthogonality checking algorithm to ensure the quantum portions are physically realizable, and provides type size annotations that can facilitate automated computation of the quantities of quantum resources that will be necessary to run the compiled program (*i.e.*, a logical quantum circuit).

Categories and Subject Descriptors D.2.6 [Software Engineering]: Programming Environments; D.3.2 [Programming Languages]: Language Classifications—specialized application languages; D.3.3 [Programming Languages]: Language Constructs and Features—constraints

Keywords domain-specific languages; quantum programming languages; type systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPCDSL '13, September 22, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2380-2/13/09...\$15.00.
<http://dx.doi.org/10.1145/2505351.2505357>

1. Introduction

Quantum computers represent a novel kind of programmable hardware with properties and restrictions that are distinct from the properties of classical computers. For example, quantum computers make it possible to operate on data that is in *superposition* (*i.e.*, it is in multiple states at once). Also, the separation between storage and computation is handled differently in quantum computers; instead of a small, fixed number of CPUs that perform instructions on data loaded from memory, within a quantum computer operations can be applied to any qubit at any point in time. This means that naively applying the sequential, control-flow-based interpretation of algorithms used in some imperative programming languages for classical computers may lead to inefficient results when employed with a quantum computer. Computations performed by quantum computers must also be *reversible* in order to run on a quantum computer, and the costs associated with performing large computations (in terms of both data operated upon and time taken to perform a computation) may be significantly different from the costs of running a classical computer. For example, memory that can store data in superposition might be significantly more expensive than memory that can store data in classical computers. These properties of quantum computers, inherited from quantum mechanics, constitute an opportunity to deliver new programming abstractions that can be exploited by programmers to solve problems more efficiently. For example, with the right abstractions, programmers might find ways to effectively deal with the higher cost of maintaining data in superposition by exploiting the parallel nature of quantum computers (*e.g.*, if programmers have at their disposal the naturally parallelizable language constructs found in functional programming languages). However, building new programming abstractions is also a challenge because the properties and constraints of quantum computers may be unintuitive to programmers who are accustomed to working with classical computers.

In this work we view quantum computers as an alternative kind of programmable device (not unlike GPUs, FPGAs, motes, or entire services such as cloud storage or computation infrastructures, software-defined networks, and so on) that has its own unique characteristics and trade-offs. Guided by the known properties of quantum computers and the hypothesis that quantum computers are likely to be just one device in a larger toolbox, we identify some possible device-specific abstractions (*i.e.*, programming language features) a programmable quantum computer may be able to support. We also investigate how some of the tools developed by the programming languages community can help programmers navigate the unique cost-benefit trade-offs they face when choosing to employ a quantum computer, either in isolation (*i.e.*, without feed-

back of the quantum computation results into the classical controlling computer) or in concert with a classical computer.

We approach this problem by introducing a domain-specific language for programming quantum computers. The purpose of the language is not to investigate the theoretical properties of quantum computation or to provide a full-featured programming language for classical computers, but to use tools developed by the programming languages community to address the practical problems and trade-offs of using a quantum computer.

QuaFL is a statically typed domain-specific programming language for writing high-level definitions of algorithms intended to be compiled into quantum circuits and executed on quantum computing devices. The design of the language is utilitarian in nature: the primary goal is to support scalable compilation to logical quantum circuits of algorithms containing at least some of the features and abstractions commonly found in high-level programming languages.

QuaFL allows programmers to describe quantum computations that operate over “high-level” data types including integers, floating point numbers, and arrays. In other words, such high-level data types can be put into superposition within an algorithm implemented using QuaFL. The language also allows programmers to define unitary transformations on this data in an abstract way using pattern matching. The QuaFL type system is designed to help programmers manage their use of the capabilities of a quantum computer by distinguishing between quantum and classical data types and data flows (with quantum data types treated somewhat like linear types). The type system is also designed with the importance of the cost of memory in mind: explicit type size annotations for quantum types are required in order to support quantum circuit size estimation and synthesis capabilities.

The QuaFL programming language supports a functional programming style, though it also provides some features for writing imperative programs. The ability to program in a functional style lets programmers impose fewer restrictions on control flow during evaluation. This is a more faithful reflection of the target representation to which algorithms using quantum computers must be compiled: quantum circuits make it possible to apply mutually independent operations simultaneously to ranges of qubits. Because a QuaFL programmer is not directed to introduce into an algorithm implementation any control flow or sequential dependencies where none are required, it is less likely that opportunities to apply operations in parallel are lost at the logical quantum circuit level, and the QuaFL compiler can generate parallel (or, parallelizable through low-level optimization) quantum circuit descriptions where it is possible. The user is assumed to be a physicist, mathematician, or computer scientist with some mathematical background that would prefer to work on quantum algorithms at a more abstract level: quantum algorithms are *functions* (*i.e.*, unitary transformations).

The rest of the paper is organized as follows. Section 2 provides a high-level context for the work presented in this paper. Section 3 describes some of the salient features the QuaFL programming language and illustrates how it can be used to implement algorithms for quantum computers. Section 4 discusses the QuaFL kind and type system. In Section 5, related work is reviewed; we conclude and outline future work in Section 6.

2. Background and Motivation

Quantum computers are devices that can use the most general computational primitives allowed by the laws of physics as currently understood [6]. Various results in complexity theory indicate that quantum computers are strictly more powerful than the “classical” computers in use today [13]. Just as high-level programming languages facilitate the compact and correct description of complex algorithms in a classical computer, we expect that a high-level pro-

gramming language for quantum computers—a *quantum programming language*—will do the same for algorithms in a quantum computer.

However, due to the fundamental differences in how these devices operate, programming languages for classical computers cannot guarantee programmers access to the features of quantum computers that provide this additional power. For example, the *no-cloning theorem* [18] states that there is no general physical operation that can duplicate quantum values without disturbance, simply because physical operations that do not disturb the state must be linear transformations. This exemplifies that quantum languages require additional constraints in order to guarantee correct translation to physically realizable operations.

There exists earlier and ongoing work on quantum programming languages [3, 8–10, 14, 15]. The focus of these other efforts has been largely the low-level description of quantum circuits or linear transformations of quantum states for the purposes of, *e.g.*, simulation in a classical computer.

Our work is distinguished by a focus on the development of a domain-specific (as opposed to general-purpose) programming language that has a minimal but sufficient collection of features and can be compiled to quantum circuits with explicit tracking of quantum computational resources. The design of our language also deviates from these previous efforts by focusing on abstract descriptions of quantum computation, *i.e.*, without explicit reference to quantum circuits. Instead, computation is described as transformations of abstract data structures that must satisfy kind- and type-dependent constraints, which in turn are constructed to ensure compilation into physically realizable operations, as well as a clear separation of the classical and quantum parts of the program in order to facilitate the tracking and estimation of the resources required to realize the algorithm.

The target user base for QuaFL are physicists, mathematicians and computer scientists designing new and potentially complex high-level quantum algorithms that exploit the unique features of quantum computers. Abstracting away the low-level implementation details of the quantum computer will make such high level efforts more tractable for users; it will also make it possible to approach in a more modular manner the problem of algorithm and circuit compilation and optimization for different hypothetical quantum computer architectures still under research and development.

3. QuaFL Language

The QuaFL programming language is being developed as part of a larger quantum programming tool chain, including an interactive development environment, a compiler, a circuit description language and intermediate representation, optimization and resource estimation tools, and other components that support logical quantum circuit synthesis for different hypothetical quantum computer architectures. One of the back-end outputs of the QuaFL compiler is a logical quantum circuit description. Logical quantum circuits are the standard symbolic representation for reversible quantum computations that can be implemented on a quantum computer (although they still abstract away many of the realities and details of quantum computer architectures). The QuaFL compiler can also generate a rich, detailed, interactive HTML report describing the results of the static analysis algorithms applied to the abstract syntax of the input program. This interactive HTML report has multiple pages, each corresponding to different views and interpretations of the abstract syntax. In this paper, we focus on presenting the QuaFL programming language.

3.1 QuaFL Abstract Syntax

Table 1 provides a definition of the abstract syntax for programs, statements and expressions. A program is a collection of one or

natural	n	\in	\mathbb{N}
variable	x, f	\in	Variables
bit	b	\in	$\{0, 1\}$
unsign. int.	u	\in	\mathbb{Z}^+
integer	i	\in	\mathbb{Z}
fixed-pt.	w	\in	\mathbb{Q}
rational	q	\in	\mathbb{Q}
real	r	\in	\mathbb{R}
complex	v	\in	\mathbb{C}
constructor	δ	\in	UserConstants
type syn.	σ	\in	UserConstants
constant	c	$::=$	true false i pi e δ b u i w q r v
prefix op.	o_1	$::=$! - sqrt log hadamard not phase cnot toffoli fredkin paulix pauliy pauliz phasex phasey phasez
infix op.	o_2	$::=$	+ - * / ^ && ++ << >> or and mod == != < > <= >=
pattern	p	$::=$	_ c (p_1, \dots, p_n)
expression	e	$::=$	x c o_1 o_2 $e_1 e_2$ ket c ctrl x measure e discard e if e_1 then e_1 else e_2 def $(\bar{x} : \bar{\tau}) = e_1 ; e_2$ fun $f(\bar{x} : \bar{\tau}) : \tau \{ e_1 \} ; e_2$ cmb $f(\bar{x} : \bar{\tau}) : \tau \{ e_1 \} ; e_2$ (e_1, \dots, e_n) array (e_1, \dots, e_n) replacing $e_1 (e_2 \text{ as } x) \{ e_3 \}$ match $e_0 \{ p_1 \Rightarrow e_1 ; \dots ; p_n \Rightarrow e_n \}$
statement	s	$::=$	e enum $\delta \{ \delta_1 \dots \delta_n \}$ type $\sigma = \tau$ size $x = a$
module	M	$::=$	module $x s_1 ; \dots ; s_n$

Table 1. Abstract syntax for the QuaFL programming language. Note that τ and a are defined in Table 3.

more modules. Each module has its own namespace and consists of a collection of definitions of type synonyms, type size variables, user-defined enumerated types, values, and functions. Types are defined in Section 4.1. For concision, we adopt the following notations: $\bar{x} = x_1, \dots, x_n$, $\bar{\tau} = \tau_1, \dots, \tau_n$, $\bar{x} : \bar{\tau} = x_1 : \tau_1, \dots, x_n : \tau_n$, and $\bar{x} \mapsto \bar{\tau} = x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n$. Note also that τ and a which appear in Table 1, are defined in Table 3.

Within the implementation of the abstract syntax data structure in the QuaFL compiler, user-specified type annotations are explicitly associated with the abstract syntax constructs where applicable.

The actual implementation of the abstract syntax data structure also reserves space for any results of the type checking algorithms.

3.2 QuaFL Features and Examples

The features of the QuaFL language are designed to expose the capabilities and limitations of a quantum computing device to the user, and to help the user organize and make informed decisions about the quantum portions of the program.

Quantizable, quantum, and control value types. In a QuaFL program, values fall into two categories: those that can be stored in classical memory, and those that can be encoded in regions of qubits within a logical quantum circuit. QuaFL behaves like a functional language in terms of variable assignment and variable scope: any individual variable is not an alias for a region of qubits that persists during program execution; rather, a variable is a name for the *state* of some region of qubits at a certain point in time (or, more generally, at a certain point in the data flow within a simulation of a quantum circuit).

The distinction between classical and quantum values is that quantum values can be in superposition (corresponding to the fact that the qubits used to encode those values within a logical quantum circuit can be in superposition). The **measure** construct takes a quantum value, and returns a classical value non-deterministically. This function represents the measurement of a quantum state in the computational basis. As a simple example of how this feature can be used, we can consider the following random bit generator that can be run on a quantum computer (note that unsigned integers are represented in the QuaFL concrete syntax as bit strings, e.g., 0b001).

```
def superposition : qu bit = hadamard(0b0);
def random_bit : bit = measure(superposition);
```

As is well known in the quantum information community, quantum states cannot be cloned. This is a consequence of the linearity of quantum mechanics. Thus, multiple uses of a variable representing a quantum value must be forbidden under certain circumstances. For example, the following function cannot be compiled to a logical quantum circuit because a is used twice.

```
fun implicit_clone (a: qu bit, b: qu bit): qu bit {
  def c: qu bit = hadamard(a);
  cnot(b, a); // return
}
```

The **discard** construct allows programmers to adhere to the requirement that all quantum values must be used exactly once within the program. The following is allowed.

```
fun noise (a: qu bit): qu bit {
  def anc: qu bit = hadamard(0b0);
  def (junk: qu bit, result: qu bit) = cnot(anc, a);
  discard(junk);
  result; // return
}
```

However, it is not always the case that quantum values must be used exactly once. In particular, if the way in which the value is used is such that it corresponds to the way a control qubit is used in a logical quantum circuit, a value can be used more than once. In the example in Figure 1, a is used twice. The first time it is used, it appears in the condition of an **if** expression (implicitly, as

a **control**), which is why it is not consumed, and may be consumed by the second use when a Hadamard transform is applied to it.

```
fun bell_basis_change (a: qu bit, b: qu bit)
: qu bit * qu bit {
  defq b =
    if (a == 0b1) {
      not(b);
    } else {
      b;
    }
  defq a = hadamard(a);
  (a, b);
}
```

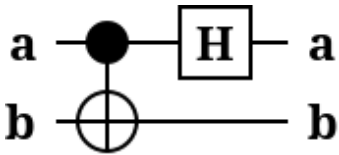


Figure 1. Example QuaFL implementation of a transformation from the Bell basis into the computational basis (with image of compiled quantum circuit below). This implementation is presented only to illustrate QuaFL features; much more compact implementations are possible. This image was generated using QCViewer [1].

For another example, Figure 3 presents a QuaFL implementation of Deutsch’s Algorithm [5], a common example used to demonstrate how quantum computers can provide significant speedup over classical computers under specific circumstances.

Constants and data types. QuaFL supports a variety of numerical types that can be stored in classical memory and can be used in classical computations (including complex numbers with rational components), as well as several types that can be quantum: bits, boolean values, unsigned integers (which can be represented in the QuaFL concrete syntax as bit strings, *e.g.*, 0b001), signed integers, and fixed-point reals. The language also supports quantum arrays of any values that can be quantum (including arrays); this means that in a QuaFL program, arrays can be put into superposition. Special features have been introduced for updating quantum arrays. In particular, a **replacing** construct makes it possible to update individual entries within quantum arrays.

The ket and match constructs. The **ket** construct makes it possible to put values that have quantum types into superpositions. For example, we can put the values 0b0 and 0b1 into superposition explicitly (this corresponds to applying a Hadamard transform to 0b0).

```
def superposition : qu bit = ket(0b0) + ket(0b1);
```

The pattern-matching notation found in popular functional languages such as Standard ML and Haskell has been adapted in QuaFL for use in defining explicit unitary transformations. The QuaFL **match** construct makes it possible to explicitly list a collection of patterns against which to match a value, and to specify how the value of the overall **match** expression should be computed for each branch. The difference between classical variants of this construct and the QuaFL **match** construct is that it is possible to use **match** expressions to specify superpositions, and thus, unitary transformations (as long as all the branches of the **match** expres-

sion are orthogonal to one another).¹ The following is an implementation of the Hadamard transform on a single qubit.

```
fun hadamard ( x: qu bit ) : qu bit {
  match x {
    0b0 => ket(0b0) + ket(0b1);
    0b1 => ket(0b0) - ket(0b1);
  }
}
```

Because the output of the Hadamard corresponds to a computational basis state of the input in a superposition, it is necessary to wrap the superposition values with the **ket** construct to make it clear that + and - here signify the superposition of two quantum states, and not arithmetic over the values representing those states.

For more examples, Figure 2 lists implementations of several standard logical quantum gates using the QuaFL **match** and **ket** constructs.² Figure 5 provides yet another QuaFL implementation of a transform found in the literature [4].

```
fun phase (x: qu bit): qu bit {
  match x {
    0b0 => 0b1;
    0b1 => i * ket(0b0);
  }
}

fun fredkin (c1: ctrl bit, t1: qu bit, t2: qu bit)
: qu bit * qu bit {
  match (c1, t1, t2) {
    (0b1, _, _) => (t2, t1);
    -           => (t1, t2);
  }
}

fun pauliy (b : qu bit): qu bit {
  match (b) {
    0b0 => i*ket(0b1);
    0b1 => -i*ket(0b0);
  }
}
```

Figure 2. Implementations of some common logical quantum gates using the QuaFL **match** and **ket** constructs.

Classical combinational functions. QuaFL allows programmers to define a classical function that can then be applied to quantum data (*i.e.*, data that may be in superposition).³ The only restriction is that all the input and output types must be quantizable (an example of a combinational function definition is provided in Figure 4).

¹ While QuaFL makes it possible to express any unitary as a **match** expression, it can be highly expensive and impractical to compile some unitary transformations defined in this way (especially unitaries of high dimension) into a finite sequence of logical quantum gates. This construct is intended to be used to express relatively sparse or small unitaries.

² Though it is possible to use the **match** construct to define these gates, they are provided as built-in constants within QuaFL (see Table 1). This facilitates direct compilation of these operations into logical quantum gates.

³ The QuaFL compiler automatically transforms classical combinational function definitions defined in QuaFL into reversible quantum circuits.

```

// Deutsch's algorithm: determines whether
// a boolean function f (wrapped inside the
// quantum function uf) is constant.

fun is_constant (
  uf: qu bit * ctrl bit -> qu bit
) : bool {
  def a : qu bit = hadamard(0b0);
  def ancilla : qu bit = hadamard(0b1);

  def ancilla : qu bit = uf(ancilla, control(a));

  def a : qu bit = hadamard(a);
  discard(ancilla);
  measure(a) == 0b0; // return
}

```

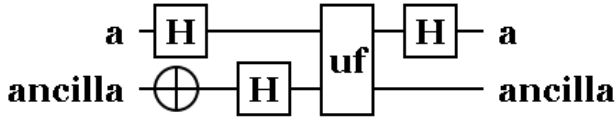


Figure 3. Implementation of Deutsch’s algorithm in QuaFL (with image of compiled quantum circuit below). This image was generated using QCViewer [1].

```

cmb main (x: uint[4]) : uint[1] {
  def c1: uint[4] = 0b0001;
  def c3: uint[4] = 0b0011;
  def u0: uint[4] = x;
  def u1: uint[4] = (c3 * u0);
  def u2: uint[4] = u1[0] ++ u1[3] ++ u1[2] ++ u1[1];
  def u3: uint[4] = c3 * (u2 + c1);
  def u4: uint[4] = u3[0] ++ u3[3] ++ u3[2] ++ u3[1];
  u4[3];
}

```

Figure 4. Combinational implementation of an oracle for the Deutsch-Jozsa algorithm.

4. Type System

The features of the QuaFL language’s type system serve three specific goals: (1) to allow programmers to distinguish between quantum and classical resources and computations within a program; (2) to ensure that quantum portions of programs adhere to constraints that guarantee the physical realizability of the program on a quantum computer; and (3) to allow the compiler to transform QuaFL programs into quantum circuit descriptions with full information about the resources necessary to represent high-level data structures as regions of qubits within the logical quantum circuit.

Towards these ends, the QuaFL type system has a unique kind system that imposes restrictions on the types of data that can be represented as quantum data. It also supports a variant of linear types that makes it possible to use quantum values under **control** constructs an arbitrary number of times while still requiring that they are used exactly once in **non-control** contexts.⁴

⁴Linear types are often used for data instances with persistent state. In this context, they are used for a slightly different purpose, which necessitates some deviation from the usual typing rules for linear types.

```

def reflection_internal (dir: qu uint[2])
: qu uint[2] {
  match dir {
    0b00 => (-1/3) * ket(0b00) + (2/3) * ket(0b01)
          + (2/3) * ket(0b11);

    0b01 => (2/3) * ket(0b00) - (1/3) * ket(0b01)
          + (2/3) * ket(0b11);

    0b11 => (2/3) * ket(0b00) + (2/3) * ket(0b01)
          - (1/3) * ket(0b11);

    _ => dir;
  }
}

def reflection_root (dir: qu uint[2])
: qu uint[2]{
  match dir {
    0b00 => (2 - sqrt(N)) * ket(0b00)
          + (2*sqrt(sqrt(N)-1)) * ket(0b01);
    0b01 => (2*sqrt(sqrt(N)-1)) * ket(0b00)
          + (sqrt(N) - 2) * ket(0b01);
    0b11 => - ket(0b11);
    _ => dir;
  }
}

```

Figure 5. More examples of unitary transformations defined using QuaFL **match** expressions.

Currently, the QuaFL type system is *monomorphic* and the compiler does only very limited type inference of constants (thus requiring explicit type annotations for all variables in input programs). The absence of polymorphic features or type inference is not necessarily due to any inherent infeasibility; rather, it is a reflection of the focus of the work being presented: distinguishing between quantum and classical data instances and operations, and requiring explicit type size annotations. Potential extensions to the type system are discussed in Section 6.

4.1 QuaFL Kinds and Types

Table 2 presents the abstract syntax for QuaFL kinds, and Table 3 presents the abstract syntax for QuaFL types.

kind κ	::=	type quantizable ket quantum control
---------------	-----	---

Table 2. Abstract syntax for QuaFL kinds.

4.2 QuaFL Kind and Type Inference Rules

Table 4 presents the kind inference rules for QuaFL types. We define Δ to be a context mapping variables to type size terms (defined in Table 3), and we define Σ to be a context mapping user-defined type synonyms to actual types. Let Δ_\emptyset and Σ_\emptyset represent the empty contexts.

In QuaFL, a type can be of kind **quantizable** (meaning values of that type can be represented both as data in the memory of classical computers and as qubits within logical quantum circuits), **ket** (meaning the value of that type is a symbolic representation of a superposition), **quantum** (meaning values of that type are data within a logical quantum circuit at that point in the program),

natural	n	\in	\mathbb{N}
variable	x	\in	Variables
infix op.	\oplus	$::=$	$+ \mid - \mid * \mid / \mid \wedge$
size	a	$::=$	$n \mid x \mid a_1 \oplus a_2 \mid \log a$
enum. type	δ	\in	UserConstants
base type	β	$::=$	bit \mid bool \mid δ
			int \mid rational \mid real \mid complex
			uint [a] \mid sint [a] \mid fixedreal [a_1, a_2]
type syn.	σ	\in	UserConstants
type	τ	$::=$	$\beta \mid \sigma \mid \mathbf{array}[\tau, a]$
			$\tau_1 * \dots * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \leftrightarrow \tau_2$
			qu $\tau \mid$ ctrl $\tau \mid$ ket $\tau \mid (\tau)$

Table 3. Abstract syntax for QuaFL types.

quantum **control** (meaning the values at that point in the program are represented by qubits that are being used as control qubits by some gate or gates), or **type** (meaning values of that type are represented as data or operations only within a classical computer).

The kind system imposes restrictions on what types can exist. For example, function types cannot be quantum reflecting that functions cannot be represented as qubits within a logical quantum circuit (e.g., **qu** (**bool** \rightarrow **bool**) has no kind, so it is not a valid type). Also notice that the kind system enforces canonical representations of quantizable and quantum data types. For example, the type **array**[**qu bit**, n] is not a valid type; the correct way to indicate that a value is an array of qubits is to assign it the type **qu array**[**bit**, n]. Type size parameters can contain variables that are defined globally in the module using the **size** statement.

Table 5 contains a selection of the type inference rules for QuaFL expressions.⁵ We define Γ to be a context mapping *all* variables to types (including those of a quantum type), and we define Φ to be a context mapping quantum variables to types. Let Γ_\emptyset and Φ_\emptyset be the empty contexts. Let \uplus represent disjoint union of contexts. We introduce two separate contexts for variables in order to accommodate the possibility that a quantum resource is used more than once as a control.

Note that in the rule APPFUN in Table 5, the function expression e_1 that is being applied is constrained to contain no quantum resources since the premise for e_1 has context Φ_\emptyset . The FUN rule does not allow variables with quantum types that might be in the surrounding context to be used inside the body of a function definition; thus, the premise for the function body e_0 extends an empty context Φ_\emptyset with the local variables that have quantum types. The CMB rule for defining classical combinational functions requires that all argument types and the return type must be quantizable.

Each bitwise, boolean, arithmetic, and relational operator constant in QuaFL can be implemented in multiple ways as a logical quantum circuit. In particular, it is often possible to allow operations that store results either *in place* or *out of place* (i.e., the result of the operation can be stored either in the same qubits in which a part of the input was stored, or in a “fresh” region of qubits). Thus, each operator constant in QuaFL may have multiple types.

A large variety of linear type systems and related variants have been proposed in the literature [2, 7, 11, 16, 17]. But the QuaFL type system must impose a constraint that is arguably distinct from the constraints linear type systems are typically designed to impose.

⁵ While there exists a prototype implementation of a type-checking QuaFL compiler, it is still under development, so there is no definitive or complete collection of type inference rules at this time.

In particular, variables that have quantum types must be used exactly once, but may be used as a form of “control” (corresponding to *control qubits* in logical quantum circuits) arbitrarily many times without restrictions. Thus, the QuaFL type system treats values of quantum types as resources that must be used exactly once in a non-control context. However, in the rule VARCTRL, all mappings in the context Γ that map a variable to a control quantum type are available to be used; this ensures that variables of a quantum control type can be used any number of times.⁶ Note also that in the IFELSEQ rule, all branches must use the same quantum variables.

It should be possible to define a semantics for QuaFL in terms of logical quantum circuits and/or unitary transformations (i.e., matrices) of quantum states; it would then be possible to formally specify a soundness property for the type system that corresponds to the physical realizability of a QuaFL program as a quantum computation. A formalization of the semantics to support the definition and proof of such a property is the subject of ongoing work.

The problem facing a QuaFL programmer is not just strict adherence to usage of quantum resources that ensures physical realizability, but also the management of trade-offs in situations where resources may be used in different ways that lead to different efficiency consequences. For example, there exist variants of circuits for arithmetic operations that store results in place when compiled, and so do not require a separate collection of qubits for the result, and variants that store results out of place when compiled, requiring a greater number of qubits. A programmer may choose to use an in-place operator, but this means that the second input to the operator is consumed and cannot be reused. Programmers can choose to make trade-offs between in-place and out-of-place operators by specifying whether the quantum data to which the operators are applied is of a type that has kind **control** or **quantum**. The interactive HTML report generated as output by the QuaFL compiler indicates explicitly all locations in which the out-of-place operator appears.⁷

4.3 Type Error Reporting and the End-user Interface

Type systems are designed to detect type errors in a program. These errors must then be communicated to the programmer. Effective error reporting can be challenging, especially when type systems support features such as linear types. These challenges arise due to the *variety* of distinct type errors that can be reported, as well as the *quantity* of errors that may be detected in a program. The QuaFL compiler addresses this problem by incorporating techniques and interface features being developed and investigated in work on accessibility and integration of formal analysis and verification tools and techniques [12]. Figure 6 presents an example of an annotated, interactive HTML report containing type checking results.

5. Related Work

There exists earlier and ongoing work on quantum programming languages [3, 8–10, 14, 15]. The focus of these other efforts has been largely the explicit description of quantum circuits or linear transformations of quantum states for the purposes of, e.g., simulation in a classical computer.

Ömer [14] developed the first real quantum programming language, the C-inspired imperative language QCL, which focused on

⁶ The QuaFL compiler output includes among its analysis results a view of the abstract syntax that indicates all locations where variables are used in multiple locations as controls. While multiple uses of a quantum control variable is allowed, it can sometimes be costly to compile such QuaFL programs into logical quantum circuits.

⁷ Determining how in-place and out-of-place operators are *compiled* (e.g., to logical quantum circuits), and how qubits should be managed under these two possible circumstances, is the task of a QuaFL compiler; such low-level details of memory management are not exposed to QuaFL users.

$\text{SIZEN} \frac{n \in \mathbb{N}}{\Delta \vdash \llbracket n \rrbracket \in \mathbb{N}}$	$\text{SIZEV} \frac{\Delta(x) = n}{\Delta \vdash \llbracket x \rrbracket \in \mathbb{N}}$	$\text{SIZEPRE} \frac{\Delta \vdash \llbracket a \rrbracket \in \mathbb{N}}{\Delta \vdash \log \llbracket a \rrbracket \in \mathbb{N}}$	$\text{SIZEIN} \frac{\Delta \vdash \llbracket a_1 \rrbracket \in \mathbb{N} \quad \Delta \vdash \llbracket a_2 \rrbracket \in \mathbb{N} \quad \oplus \in \{+, -, *, /, \wedge\}}{\Delta \vdash \llbracket a_1 \oplus a_2 \rrbracket \in \mathbb{N}}$
$\text{SMPQ} \frac{\tau \in \{\text{bit}, \text{bool}, \delta\}}{\vdash \tau : \text{quantizable}}$	$\text{SMPT} \frac{\tau \in \{\text{int}, \text{rational}, \text{real}, \text{complex}\}}{\vdash \tau : \text{type}}$	$\text{SYNT} \frac{\Delta \vdash \Sigma(\sigma) : \kappa}{\Sigma; \Delta \vdash \sigma : \kappa}$	$\text{ARRT} \frac{\Sigma; \Delta \vdash \tau : \text{quantizable} \quad \Delta \vdash \llbracket a \rrbracket \in \mathbb{N}}{\Sigma; \Delta \vdash \text{array}[\tau, a] : \text{quantizable}}$
$\text{UNST} \frac{\Delta \vdash \llbracket a \rrbracket \in \mathbb{N}}{\Sigma; \Delta \vdash \text{uint}[a] : \text{quantizable}}$	$\text{SGNT} \frac{\Delta \vdash \llbracket a \rrbracket \in \mathbb{N}}{\Sigma; \Delta \vdash \text{sint}[a] : \text{quantizable}}$	$\text{FXT} \frac{\Delta \vdash \llbracket a \rrbracket \in \mathbb{N} \quad \Delta \vdash \llbracket b \rrbracket \in \mathbb{N}}{\Sigma; \Delta \vdash \text{fixedreal}[a, b] : \text{quantizable}}$	
$\text{PRDT} \frac{i \in \{1, \dots, n\} \quad \Sigma; \Delta \vdash \tau_i : \kappa_i}{\Sigma; \Delta \vdash \tau_1 * \dots * \tau_n : \text{type}}$	$\text{FNT} \frac{\Sigma; \Delta \vdash \tau_1 : \kappa_1 \quad \Sigma; \Delta \vdash \tau_2 : \kappa_2}{\Sigma; \Delta \vdash \tau_1 \rightarrow \tau_2 : \text{type}}$	$\text{CMBT} \frac{\Sigma; \Delta \vdash \tau_1 : \text{quantizable} \quad \Sigma; \Delta \vdash \tau_2 : \text{quantizable}}{\Sigma; \Delta \vdash \tau_1 \leftrightarrow \tau_2 : \text{type}}$	
$\text{CLAT} \frac{\Sigma; \Delta \vdash \tau : \text{quantizable}}{\Sigma; \Delta \vdash \tau : \text{type}}$	$\text{KETT} \frac{\Sigma; \Delta \vdash \tau : \text{quantizable}}{\Sigma; \Delta \vdash \text{ket } \tau : \text{ket}}$	$\text{QUAT} \frac{\Sigma; \Delta \vdash \tau : \text{quantizable}}{\Sigma; \Delta \vdash \text{qu } \tau : \text{quantum}}$	$\text{CTRTR} \frac{\Sigma; \Delta \vdash \tau : \text{quantum}}{\Sigma; \Delta \vdash \text{ctrl } \tau : \text{control}}$

Table 4. Kind inference rules for QuaFL programming language types.

$\text{BIT} \frac{b \in \{0, 1\}}{\vdash b : \text{bit}}$	$\text{BOOL} \frac{b \in \{\text{true}, \text{false}\}}{\vdash b : \text{bool}}$	$\text{INT} \frac{i \in \mathbb{Z}}{\vdash i : \text{int}}$	$\text{RATIONAL} \frac{q \in \mathbb{Q}}{\vdash q : \text{rational}}$	$\text{REAL} \frac{r \in \mathbb{R} \uplus \{\text{pi}, \text{e}\}}{\vdash r : \text{real}}$	$\text{CPLX} \frac{v \in \mathbb{C} \uplus \{\text{i}\}}{\vdash v : \text{complex}}$
$\text{UNS} \frac{u \in \mathbb{Z}^+ \quad u < 2^n}{\vdash u : \text{uint}[n]}$	$\text{SGN} \frac{u \in \mathbb{Z}^+ \quad -2^n + 1 < i < 2^n}{\vdash i : \text{sint}[n+1]}$	$\text{FX} \frac{w \in \mathbb{Q} \quad w < 2^n \quad w \cdot 2^m \in \mathbb{Z}}{\vdash w : \text{fixedreal}[n, m]}$	$\text{KET} \frac{\vdash c : \tau \quad \Sigma; \Delta \vdash \tau : \text{quantizable}}{\Sigma; \Delta; \Gamma; \Phi \vdash \text{ket } c : \text{ket } \tau}$		
$\text{VAR} \frac{\Gamma(x) = \tau \quad \Sigma; \Delta \vdash \tau : \kappa \quad \kappa \in \{\text{type}, \text{quantizable}\}}{\Sigma; \Delta; \Gamma; \Phi \vdash x : \tau}$		$\text{VARQUANTUM} \frac{\Phi = \{x \mapsto \text{qu } \tau\}}{\Sigma; \Delta; \Gamma; \Phi \vdash x : \text{qu } \tau}$		$\text{VARCTRL} \frac{\Gamma(x) = \text{ctrl } \tau}{\Sigma; \Delta; \Gamma; \Phi \vdash \text{ctrl } x : \text{ctrl } \tau}$	
$\text{KETQUANTUM} \frac{\Sigma; \Delta; \Gamma; \Phi \vdash e : \text{ket } \tau}{\Sigma; \Delta; \Gamma; \Phi \vdash e : \text{qu } \tau}$		$\text{QUANTUM} \frac{\Sigma; \Delta; \Gamma; \Phi \vdash e : \tau \quad \Sigma; \Delta \vdash \tau : \text{quantizable}}{\Sigma; \Delta; \Gamma; \Phi \vdash e : \text{qu } \tau}$		$\text{MEASURE} \frac{\Sigma; \Delta; \Gamma; \Phi \vdash e : \text{qu } \tau}{\Sigma; \Delta; \Gamma; \Phi \vdash \text{measure } e : \tau}$	
$\text{DEF} \frac{\Sigma; \Delta; \Gamma; \Phi_1 \vdash e_0 : (\tau_1, \dots, \tau_n) \quad \Sigma; \Delta; \Gamma \uplus \{x_i \mapsto \tau_i \mid \Sigma; \Delta \vdash \tau_i : \text{type}\} \uplus \{x_i \mapsto \text{ctrl } \tau'_i \mid \tau_i = \text{qu } \tau'_i\}; \Phi_2 \uplus \{x_i \mapsto \tau_i \mid \tau_i = \text{qu } \tau'_i\} \vdash e : \tau}{\Sigma; \Delta; \Gamma; \Phi_1 \uplus \Phi_2 \vdash \text{def } (x_1 : \tau_1, \dots, x_n : \tau_n) = e_0 ; e : \tau}$					
$\text{FUN} \frac{\Sigma; \Delta; \Gamma \uplus \{x_i \mapsto \tau_i \mid \Sigma; \Delta \vdash \tau_i : \text{type}\} \uplus \{x_i \mapsto \text{ctrl } \tau'_i \mid \tau_i = \text{qu } \tau'_i\} \uplus \{f \mapsto (\tau_1, \dots, \tau_n) \rightarrow \tau_0\}; \Phi_0 \uplus \{x_i \mapsto \tau_i \mid \tau_i = \text{qu } \tau'_i\} \vdash e_0 : \tau_0 \quad \Sigma; \Delta; \Gamma \uplus \{f \mapsto (\tau_1, \dots, \tau_n) \rightarrow \tau_0\}; \Phi \vdash e : \tau}{\Sigma; \Delta; \Gamma; \Phi \vdash \text{fun } f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau_0 \{e_0\} ; e : \tau}$					
$\text{CMB} \frac{\Sigma; \Delta; \Gamma \uplus \{\bar{x} \mapsto \bar{\tau}\}; \Phi_0 \vdash e_0 : \tau_0 \quad \Sigma; \Delta; \Gamma \uplus \{f \mapsto (\tau_1, \dots, \tau_n) \rightarrow \tau_0\}; \Phi \vdash e : \tau \quad \forall i, \Sigma; \Delta; \vdash \tau_i : \text{quantizable}}{\Sigma; \Delta; \Gamma; \Phi \vdash \text{cmb } f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau_0 \{e_0\} ; e : \tau}$					
$\text{APPFUN} \frac{\Sigma; \Delta; \Gamma; \Phi_0 \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Sigma; \Delta; \Gamma; \Phi \vdash e_2 : \tau_2}{\Sigma; \Delta; \Gamma; \Phi \vdash e_1 e_2 : \tau_1}$			$\text{APPCMB} \frac{\Sigma; \Delta; \Gamma; \Phi_0 \vdash e_1 : \tau_2 \leftrightarrow \tau_1 \quad \Sigma; \Delta; \Gamma; \Phi \vdash e_2 : \text{ctrl } \tau_2}{\Sigma; \Delta; \Gamma; \Phi \vdash e_1 e_2 : \text{qu } \tau_1}$		
$\text{IFELSE} \frac{\Sigma; \Delta; \Gamma; \Phi_1 \vdash e_1 : \text{bool} \quad \Sigma; \Delta; \Gamma; \Phi_2 \vdash e_2 : \tau \quad \Sigma; \Delta; \Gamma; \Phi_3 \vdash e_3 : \tau}{\Sigma; \Delta; \Gamma; \Phi_1 \uplus \Phi_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$			$\text{IFELSEQ} \frac{\tau_1 \in \{\text{qu bool}, \text{ctrl bool}\} \quad \Sigma; \Delta; \Gamma; \Phi_2 \vdash e_2 : \text{qu } \tau_2 \quad \Sigma; \Delta; \Gamma; \Phi_1 \vdash e_1 : \tau_1 \quad \Sigma; \Delta; \Gamma; \Phi_3 \vdash e_3 : \text{qu } \tau_2}{\Sigma; \Delta; \Gamma; \Phi_1 \uplus \Phi_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{qu } \tau_2}$		

Table 5. Selected type inference rules for QuaFL programming language expressions.

simulation of quantum computers, and included a complete classical sublanguage for the post-processing of classical data generated by the quantum part.

There has been significant work in the development of functional languages for quantum computers, with two distinct flavours: one focusing on classical control and quantum data, and one allowing for quantum control and data. The first variety includes the work on QPL and Quipper [10, 15], while the second revolves around Alternkirch and Grattage’s QML [3]. Formal semantics and compilers for both these languages have been developed, targeting concrete quantum circuits in a random access memory model of quantum computation and resource-unconstrained quantum circuits. However, QPL remains relatively low-level in its descriptions of quantum circuits, while QML fails to address some of the more

common patterns in quantum algorithms, such as the automatic translation of classical irreversible functions into reversible computation to be performed by a quantum computer on quantum data.

6. Conclusions and Future Work

We have presented the QuaFL programming language and its type system, which helps programmers exploit some of the unique features of quantum computers while navigating the constraints imposed by these devices. There exist many directions for future work. These include the introduction into QuaFL of more programming language abstractions and features found in other programming languages, improvement of the QuaFL compiler (including optimizations), and further development of the QuaFL type system.

```

func DOWELD1 {
  treeBit: ctrl_bit
  a: ctrl_uint[N]
  b: qu_uint[N]
  f: qu_uint[N]
}: qu_uint[N] {
  defq tmp =
    if ( not (treeBit) ) {
      a + f
    } else {
      a - f
    }
  ctrl (tmp) xor ctrl (b)
}

```

```

procedure DOWELD1 [] ( treeBit, a, b, f ) <
  ...
> {
  NOT (treeBit);
  control (treeBit) {
    call *adderUnsignedInPlaceModulus [4] ( a, f );
  };
  control not (treeBit) {
    call *subtractUnsignedInPlaceModulus [4] ( a, f );
  };
  NOT (treeBit);
  tmp := (f);
  _v_36 := wire [4];
  call *xorOutOfPlace [4] ( tmp, b, _v_36 );
};

```

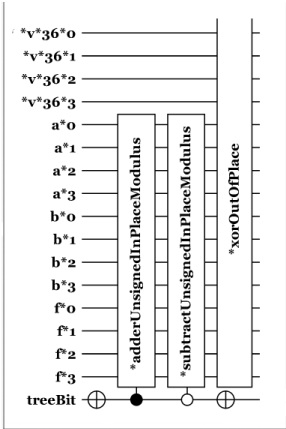


Figure 6. Screen capture of interactive HTML output from the QuaFL compiler, with syntax highlighting, interactive static analysis results annotations, and compiled logical quantum circuit output with circuit diagram generated using QCViewer [1].

Adding user-defined recursive data types found in functional languages such as Haskell and Standard ML would allow programmers to work with high-level data structures such as trees in superposition. Adding a library of common functions on existing types (such as transcendental functions for quantizable fixed-point real numbers) would provide more options for programmers who wish to implement quantum algorithms that apply these functions to data that is in superposition. It is also possible to make QuaFL a full-featured functional programming language for classical computers, such as Standard ML or Haskell.

Polymorphic or dependent types can be introduced into the QuaFL type system in order to eliminate the need to define static integer type parameters. However, whenever a user wishes to generate a logical quantum circuit, it would still be necessary to either specify the static integer type parameters, or to provide explicit inputs to the algorithm defined by the programmer so the size parameters can be determined at the time of circuit synthesis. A more powerful symbolic algorithm for validating that different branches of **match** expressions are orthogonal can be constructed by invoking existing SMT solvers or other third-party tools. It may also be of interest to investigate how existing linear type system variants such as quasi-linear types [11] might relate to the physical realizability constraints the QuaFL type system imposes on programs.

Acknowledgments

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center Contract number DIIIPC20166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC or the U.S. Government.

We thank Richard Lazarus for helpful discussions.

References

- [1] QCViewer: a tool for displaying, editing, and simulating quantum circuits, 2012. Available at <http://qcirc.iqc.uwaterloo.ca/>.
- [2] P. Achten and R. Plasmeijer. The ins and outs of clean i/o. *Journal of Functional Programming*, 5:81–110, 0 1995. ISSN 1469-7653. .
- [3] T. Altenkirch and J. J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Science Press, 2005.
- [4] A. M. Childs, B. W. Reichardt, R. Spalek, and S. Zhang. Every nand formula of size n can be evaluated in time $n^{(1/2)+o(1)}$ on a quantum computer. arXiv:quant-ph/0703015v3, 2007.
- [5] D. Deutsch and R. Jozsa. Rapid Solution of Problems by Quantum Computation. *Royal Society of London Proceedings Series A*, 439: 553–558, Dec. 1992. .
- [6] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6,7):467–488, 1982.
- [7] M. Fluet, G. Morrisett, and A. J. Ahmed. Linear regions are all you need. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 7–21. Springer, 2006. ISBN 3-540-33095-X.
- [8] S. J. Gay. Quantum programming languages: Survey and bibliography. *Math. Struct. in Comp. Sci.*, 16(4):581–600, 2006.
- [9] J. Grattage and T. Altenkirch. Qml: Quantum data and control. submitted for publication, February 2005.
- [10] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 333–342. ACM, 2013. ISBN 978-1-4503-2014-6.
- [11] N. Kobayashi. Quasi-linear types. In A. W. Appel and A. Aiken, editors, *POPL*, pages 29–42. ACM, 1999. ISBN 1-58113-095-3.
- [12] A. Lapets, R. Skowrya, A. Bestavros, and A. Kfoury. Towards Accessible Integration and Deployment of Formal Tools and Techniques. In *Proceedings of the 3rd Workshop on Developing Tools as Plug-ins (TOPI 2013)*, San Francisco, CA, USA, May 2013.
- [13] M. Mosca. Quantum algorithms. In *Encyclopedia of Complexity and Systems Science*. Springer, 2009.
- [14] B. Ömer. *Structured Quantum Programming*. PhD thesis, Technical University of Vienna, 2003.
- [15] P. Selinger. Towards a quantum programming language. *Math. Struct. in Comp. Sci.*, 14:527–586, 2004.
- [16] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, Oct. 2006. ISSN 0167-6423. . URL <http://dx.doi.org/10.1016/j.scico.2006.02.003>.
- [17] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990. URL <http://citeseer.nj.nec.com/wadler90linear.html>.
- [18] W. K. Woiters and W. H. Zurek. A single quantum cannot be clones. *Nature*, 299:802–803, 1982.