Qimaera: Type-safe (Variational) Quantum Programming in Idris

Liliane-Joy Dandy
Ecole Polytechnique/EPFL
France/Switzerland

Emmanuel Jeandel
Université de Lorraine
France

Vladimir Zamdzhiev
Inria
France

ABSTRACT

Variational Quantum Algorithms are hybrid classical-quantum algorithms where classical and quantum computation work in tandem to solve computational problems. These algorithms create interesting challenges for the design of suitable programming languages. In this paper we introduce Qimaera, which is a set of libraries for the Idris 2.0 programming language that enable the programmer to implement (variational) quantum algorithms where the full power of the elegant Idris language works in synchrony with quantum programming primitives that we introduce. The two key ingredients of Idris that make this possible are (1) dependent types which allow us to implement unitary (i.e. reversible and controllable) quantum operations; and (2) linearity which allows us to enforce fine-grained control over the execution of quantum operations that ensures compliance with the laws of quantum mechanics. To the best of our knowledge, this is the first programming language that is suitable for variational quantum programming in the sense that it provides first-class high-level support for both classical and quantum programming and that is moreover type-safe.

KEYWORDS

variational quantum programming, dependent types, linear types

1 INTRODUCTION

Quantum Computing is a new and emerging computational paradigm whose main idea is to use quantum mechanical phenomena, such as entanglement and superposition, in order to perform computation. A quantum computer can solve problems which are out of reach for classical computers (e.g. factorisation of large numbers [26], solving large linear systems [18]) and this has caused a major surge of interest into the development of quantum technologies. A major breakthrough was recently achieved by Google which demonstrated quantum computational advantage on existing quantum computers [9]. The development of quantum computing technologies is rapidly accelerating and has recently benefited from major investment from technological companies with dedicated quantum research teams, such as Google, Microsoft, IBM, and many other quantum startup firms. Quantum technologies are also one of the main focus areas for the European Research Council [6].

Variational Quantum Algorithms [22, 25] are becoming increasingly important for quantum computation. The main idea behind this computational paradigm is to use hybrid classical-quantum algorithms that work in tandem to solve computational problems. The classical part of the algorithm is performed by a classical processor and the quantum part of the algorithm is executed on a quantum device. During the computation process, intermediary results produced by the quantum device are obtained with probabilities determined by the laws of quantum mechanics, and then are manipulated by the classical processor, which performs further computation on them that is used to tune the parameters of the quantum part of the algorithm, which therefore has an effect on the quantum dynamics. The hybrid classical-quantum back and forth process repeats until a satisfactory result has been obtained.

This hybrid classical-quantum computational paradigm opens up interesting and important challenges for the design of suitable programming languages. It is clear that if we wish to program within such computational scenarios, we need to develop a language that correctly models the manipulation of *quantum resources*. In particular, quantum measurements give rise to *probabilistic computational effects* that are inherited by the classical side of the language. Another issue that has to be accounted for by a suitable language is that quantum information behaves very differently from classical information. As an example, quantum information cannot be copied in a uniform way [27], unlike classical information, which may be freely copied without restriction. Therefore, if we wish to avoid runtime errors, the quantum fragment of the language needs to be equipped with features for fine-grained control, such as for example, having a *substructural typing discipline* [11, 13, 14, 17, 21] where contraction (i.e. copying) is restricted. On the other hand, when doing classical computation, such restrictions are unnecessary and often inconvenient. One solution to this problem is to design a language with a classical (non-linear) fragment together with a quantum (linear) one, both of which interact nicely with each other. In fact, this can be achieved within an existing language that has a sufficiently advanced type system, as we show in this paper.

1.1 Our Contributions

In this paper, we describe *Qimaera* (named after the hybrid creature Chimaera from Greek mythology), which is a set of libraries for the Idris 2 programming language [15] that allow the programmer to implement (variational) quantum algorithms in Idris in a *type-safe* way. Idris 2 is an elegant functional programming language that is equipped with an advanced type system based on Quantitative Type Theory [11, 21] that brings many useful features to the programmer, most notably *dependent types* and *linearity*. These two features of Idris are crucial for the development of Qimaera and, in fact, are the reason we chose Idris in the first place (we do not know of any other language that supports both features simultaneously). Dependent types are used throughout our entire development in order to correctly represent and formalise the compositional nature of quantum operations. Linearity is used in order to enforce the proper consumption of quantum resources (during execution) in a way that is admissible with respect to the laws of quantum mechanics. The combination of dependent types and linearity allows us to *statically* detect and reject erroneous quantum programs and this ensures the type-safety of our approach to variational quantum programming.

In our intended computational scenario, we should have access to both a classical computer and a quantum computer. Since we

cannot directly observe quantum information, we directly interact with the classical computer which sends instructions to, and receives data from, the quantum device via a suitable interface that makes use of the IO monad. In our view, this is an adequate representation of a realistic computational environment for variational quantum programming. However, since we do not personally have any quantum hardware, we instead simulate the relevant quantum operations on our classical computers by using the proper linear-algebraic formalism, but while still using an IO monad to accurately generate probabilistic effects as prescribed by quantum mechanics. From a high-level programming perspective, the simulation approach addresses the same programming challenges that arise from the realistic classical-quantum device scenario.

We emphasise that we can achieve type-safe (variational) quantum programming in an *existing* classical programming language by implementing suitable libraries. This is important for *variational* quantum programming, because in most variational quantum algorithms, the classical part of the algorithm is considerably larger, more complicated and more difficult to implement, compared to the quantum part of the algorithm. Therefore, it is important for the programming language to have good support for classical programming features. Our chosen language, Idris, is definitely such a language. The advanced type system of Idris allows us to elegantly mix quantum and classical programming primitives and therefore allows us to get the best of both worlds. To showcase this, we implement several (variational) quantum algorithms and we show that (high-level) quantum and classical computation live in synchrony within Idris and may be mixed in an elegant way.

To the best of our knowledge, our paper describes the first language that is suitable for variational quantum programming in the sense that it provides first-class high-level support for *both* classical and quantum programming and that is moreover *type-safe*.

1.2 Overview

The paper is organised as follows:

- We begin by providing some background on quantum computation (§2.1) and the Idris programming language (§2.2).
- We then explain how we represent unitary (i.e. reversible and controllable) quantum operations in Idris and we provide some important and non-trivial examples (§3). We do *not* make use of linearity in order to achieve this, but our development makes extensive use of dependent types.
- In §4 we describe how we represent arbitrary (non-unitary, effectful) quantum operations and we present some simple examples of effectful quantum programs and algorithms. The linear features of Idris are crucial for achieving this.
- We discuss why Qimaera is suitable for variational quantum programming and we provide a prototype implementation of the Variational Quantum Eigensolver algorithm in §5.
- Finally, we discuss related work in §6 and we provide concluding remarks in §7.

The Idris source code for Qimaera is on Github at the following URL: https://github.com/zamdzhiev/Qimaera. It is licensed under the MIT license and it may be freely used by anyone.

2 BACKGROUND

In this section we introduce the relevant background and we fix notation so that readers may follow our subsequent development.

2.1 Quantum Computation

Readers interested in a detailed introduction to quantum computing may consult [23]. In this section we summarise the basic notions that are relevant for our development.

2.1.1 Qubits. The simplest non-trivial quantum system is the *quantum bit*, often abbreviated as *qubit*. Qubits are of fundamental importance in quantum computation and quantum information and they may be thought of as the quantum counterparts of the bit from classical computation. A qubit $|\psi\rangle$ is represented as a normalised vector in $\mathbb{C}^2$. The *computational basis* is given by the pair of vectors

$$|0\rangle \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \text{and} \qquad |1\rangle \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

which may be seen as representing the classical bits 0 and 1. An arbitrary qubit can therefore be described by $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ with $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$.

2.1.2 Superposition. A qubit may be in (uncountably) many different states, whereas a classical bit is either 0 or 1. When the linear combination $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ is non-trivial, then we say that $|\psi\rangle$ is in *superposition* of $|0\rangle$ and $|1\rangle$. Superposition is a very important quantum resource which is used by many quantum algorithms.

2.1.3 Composite Systems. The state space that describes a system of n qubits is the Hilbert space $\mathbb{C}^{2^n}$. Notice that the dimension of the state space grows exponentially with the number of qubits. If $|\psi\rangle$ and $|\phi\rangle$ are two states of n and m qubits respectively, then the composite $n+m$ qubit state $|\psi\phi\rangle \stackrel{\text{def}}{=} |\psi\rangle \otimes |\phi\rangle$ is described by the Kronecker product $\otimes$ of the original states.

2.1.4 Unitary Quantum Operations. A quantum state $|\psi\rangle \in \mathbb{C}^{2^n}$ may undergo a *unitary evolution* described by a unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$ in which case the new state of the system is described by the vector $U\,|\psi\rangle$. Unitary operations (and matrices) are closed under sequential composition (described by matrix multiplication $\circ$) and under parallel composition (described by Kronecker product $\otimes$). Sequential composition of unitary operations is used to describe the temporal evolution of quantum systems, whereas the parallel composition is used to describe the action of several unitary transformations acting simultaneously on different parts of composite quantum systems.

The unitary quantum operations are also often called *unitary gates*. One typically chooses a *universal gate set* which is a small set of unitary operations that suffices to express all other unitary operations via (parallel and sequential) composition. The universal gate set that we choose for our development is standard and we specify these unitary operations next by giving their action on the computational basis (which uniquely determines the operations).

The *Hadamard Gate*, denoted H, is the 1-qubit unitary map whose action on the computational basis is given by

$$H\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \qquad H\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

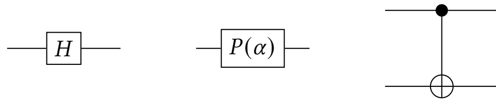


Figure 1: The Hadamard, Phase Shift and CNOT gates.

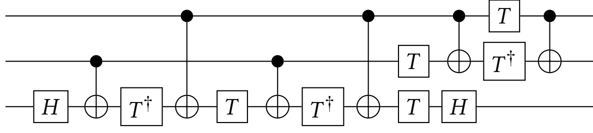


Figure 2: The Toffoli gate.

and its primary purpose is to generate superposition. The *Phase Shift Gate*, denoted $P(\alpha)$, for $\alpha \in \mathbb{R}$, is a 1-qubit unitary map whose action on the computational basis is given by:

$$P(\alpha)\left|0\right\rangle = \left|0\right\rangle \qquad P(\alpha)\left|1\right\rangle = e^{i\alpha}\left|1\right\rangle$$

and its primary purpose is to modify the phase of a quantum state. The family of Phase Shift Gates is parameterised by the choice of $\alpha \in \mathbb{R}$ and important special cases include the unitary gates $T \stackrel{\text{def}}{=} P(\pi/4)$ and $Z \stackrel{\text{def}}{=} P(\pi)$. The *Controlled-Not Gate* (CNOT), is a 2-qubit unitary map whose action on the computational basis is given by

$$\begin{array}{ll} \mathrm{CNOT}\left|00\right\rangle = \left|00\right\rangle & \mathrm{CNOT}\left|01\right\rangle = \left|01\right\rangle \\ \mathrm{CNOT}\left|10\right\rangle = \left|11\right\rangle & \mathrm{CNOT}\left|11\right\rangle = \left|10\right\rangle \end{array}$$

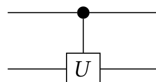
and this unitary map may be used to generate quantum entanglement (see §2.1.7).

Unitary gates admit a diagrammatic representation as *quantum circuits*. The atomic unitary gates we described above are shown in Figure 1. Composite unitary gates may also be described as circuits (see Figure 2): sequential composition amounts to plugging wires of subdiagrams left-to-right and parallel composition amounts to juxtaposition of circuits top-to-bottom.

2.1.5 Controlled Unitary Operations. The CNOT gate is the simplest example of a *controlled unitary gate*. Given a unitary gate $U : \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, the controlled-$U$ unitary gate is the unitary gate $CU : \mathbb{C}^{2^{n+1}} \to \mathbb{C}^{2^{n+1}}$ whose action is determined by the assignments

$$CU(\left|0\right\rangle \otimes \left|\psi\right\rangle) = \left|0\right\rangle \otimes \left|\psi\right\rangle \qquad \text{and} \qquad CU(\left|1\right\rangle \otimes \left|\psi\right\rangle) = \left|1\right\rangle \otimes (U\left|\psi\right\rangle).$$

Controlled unitary operations are ubiquitous in quantum computing and they are graphically depicted as



using a similar notation to that of the CNOT gate.

2.1.6 Inverse Unitary Operations. Every unitary operation U is *reversible* with the inverse operation given by the conjugate transpose, denoted $U^\dagger$, which is again a unitary matrix. Applying the inverse operation (also known as the *adjoint*) of a given unitary is also ubiquitous in quantum computing.

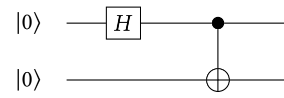


Figure 3: Preparation of the Bell state using atomic gates.

2.1.7 Quantum Entanglement. A quantum state $\left|\psi\right\rangle \in \mathbb{C}^{2^n}$, with $n > 1$, is said to be *entangled* when there exists no non-trivial decomposition $\left|\psi\right\rangle = \left|\phi\right\rangle \otimes \left|\tau\right\rangle$. Quantum entanglement is a very important resource in quantum computation which is exhibited by many quantum algorithms. Because of the possibility of entanglement, we cannot, in general, break down quantum systems into smaller components and we are often forced to reason about such systems in their entirety. The most important example of an entangled state is the *Bell state* given by $\left|\mathrm{Bell}\right\rangle \stackrel{\text{def}}{=} \frac{\left|00\right\rangle + \left|11\right\rangle}{\sqrt{2}}$.

2.1.8 Preparation of Quantum States. Preparing a new qubit in state $\left|0\right\rangle$ is an admissible physical operation. This, together with application of unitary gates as part of the computation, allows us to prepare arbitrary quantum states. For example, the Bell state can be prepared by taking $\left|\mathrm{Bell}\right\rangle = (\mathrm{CNOT} \circ (H \otimes I))\left|00\right\rangle$. See Figure 3 for a diagrammatic representation. Notice that the circuit in Figure 3 is applied to the initial state $\left|00\right\rangle$ and therefore describes a state, whereas the circuits in Figures 1 and 2 are unitary operations and do not describe states, because no input state is specified.

2.1.9 Measurements. Quantum information cannot be directly observed without affecting the state of the underlying system. In order to extract information from quantum systems, we need to perform a *quantum measurement* on (parts of) our systems. For example, when performing a quantum measurement on a qubit in the state $\left|\psi\right\rangle = a\left|0\right\rangle + b\left|1\right\rangle$, there are two possible outcomes: either the quantum system will collapse to state $\left|0\right\rangle$ and we obtain the classical bit 0 as evidence of this event, or, the quantum system will collapse to state $\left|1\right\rangle$ and we obtain the classical bit 1 as evidence of this event. The first outcome (corresponding to bit 0) occurs with probability $|a|^2$ and the second outcome (corresponding to bit 1) occurs with probability $1 - |a|^2 = |b|^2$. In general, when we measure n qubits simultaneously, we obtain a bit string of length n which determines the event that occurred and the quantum system collapses to a corresponding state with some probability, both of which are determined via the Born rule of quantum mechanics. Therefore, quantum measurements induce evolutions which are *probabilistic* and *irreversible* (or *destructive*), which distinguishes them from unitary evolutions, which are *deterministic* and *reversible.*

2.1.10 No-Cloning Theorem. Unlike classical information, quantum information cannot be uniformly copied. This is made precise by the *no-cloning* theorem of quantum mechanics [27]: there exists no unitary operation $U : \mathbb{C}^4 \to \mathbb{C}^4$, such that for every qubit $\left|\psi\right\rangle$:

$$U(\left|\psi\right\rangle \otimes \left|0\right\rangle) = \left|\psi\right\rangle \otimes \left|\psi\right\rangle.$$

This means that copying of quantum information is a *physically inadmissible* operation. Therefore, quantum programming languages should be designed so that these kinds of errors are detected by the language, ideally during type checking and not at runtime.

2.2 The Idris Language

In this section, we give a short overview of the Idris 2 programming language and its most crucial features for the development of Qimaera.

Idris 2 is a pure functional language with a syntax influenced by that of Haskell. It introduces two powerful constructions, linearity and dependent types, that are the cornerstone of our implementation of quantum primitives.

2.2.1 Dependent Types. In Idris, types can be manipulated like any other construct of the language. This allows to have more expressive types that can depend on a value, and hence it enables to make some properties and program invariants explicit. The type of vectors is the most common example of a dependent type: a vector is a list with a fixed length, which is part of its type. It can be declared as follows, where S is the constructor for the successor of a natural number, and a is a polymorphic type:

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect 0 a
  (::) : a -> Vect k a -> Vect (S k) a
```

Here, the type Vect has two constructors. The first one constructs the empty vector, of length zero. The second one constructs all non-empty vectors: a vector of size $k + 1$ with elements of type a is obtained by combining an element of type a and a vector of size k.

Some correctness and mathematical properties are then ensured during type-checking. For instance, we can define an append function that concatenates two vectors. In this case, the size of the output vector is the sum of the sizes of the input vectors.

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

If we make a mistake in the definition, for example if we forget the recursive call, an error will be raised at type-checking.

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x::xs) ys = x::ys

Error: While processing right hand side of append.
Can't solve constraint between: m and plus k m.
```

Type dependency may also be used to express some constraints on the parameters of a function. Here, we write a pop function that cannot be applied to an empty vector.

```
pop : Vect (S k) a -> Vect k a
pop (x :: xs) = xs
```

Writing `pop [ ]` will fail at compile time rather than at runtime.

However, using dependent types makes type-checking undecidable. Indeed, the equivalence between types has sometimes to be proved inside the code. This is the case, for example, for the types `Vect (n + k) Nat` and `Vect (k + n) Nat`: the compiler needs a proof that addition is commutative to unify these types.

2.2.2 Linearity. The type system of Idris 2 is based on Quantitative Type Theory. Every function argument is associated with a multiplicity that states the number of times the variable is used at runtime. This multiplicity can be 0, 1 or ω. A parameter with a multiplicity 0 is only used at compile time and is erased at runtime. A linear argument, with multiplicity 1, is used exactly once at runtime. If the argument is a variable, it is used when it is pattern matched against, and if it is a function, it is used when it is run. Finally, the multiplicity ω, the default one, means that the usage of the argument is unrestricted.

In Qimaera, when manipulating quantum information, linearity will be enforced in order to consume properly quantum resources and comply with the no-cloning theorem.

In our libraries, we define the data type `LVect` (which stands for linear vector). The definition is similar to that of `Vect`, but the parameters of the constructor are linear (linearity is specified with the multiplicity 1 in front of each argument). We also use the linear pairs (`LPair`) that are already defined in Idris 2.

```
data LVect : Nat -> Type -> Type where
  Nil : LVect 0 a
  (::) : (1 _ : a) -> (1 _ : LVect k a) ->
         LVect (S k) a

data LPair : Type -> Type -> Type
  (#) : (1 _ : a) -> (1 _ : b) -> LPair a b
```

We can illustrate linearity with the standard example of the duplication function. In this function, we are trying to use twice a linear resource, and an error is reported at compile time.

```
duplication : (1 _ : a) -> LPair a a
duplication x = x # x

Error: While processing right hand side of
duplication. There are 2 uses of linear name x.
```

3 UNITARY GATES IN QIMAERA

As we saw in §2.1, unitary transformations have a special role in quantum computation. In fact, in most non-variational quantum algorithms, the vast majority of the programming effort consists in implementing the required unitary gates. In this section, we describe our representation of unitaries transformations in Qimaera.

3.1 The Unitary Data Type

Quantum unitary operations admit a compositional and algebraic representation based on the gates from the universal gate set. Our idea for the representation of quantum unitaries is based on this, and more specifically, on how quantum unitaries may be expressed in terms of quantum circuit diagrams. Because of these reasons, linearity is not required for our formalisation of quantum unitaries.

Our reasoning above shows that quantum unitary operations may be represented as an algebraic data type within Idris. The code which does this is listed in Figure 4 and we now describe this formalisation in greater detail.

Given a natural number `n : Nat`, the type of quantum unitaries acting on `n` qubits is given by `Unitary n`. Therefore `Unitary` is an algebraic data type with a simple type dependency on the arity of the desired operation. The `Unitary` type has four different introduction rules which we describe next.

The first constructor, `IdGate`, represents the identity unitary on `n` qubits. We can see this as constructing a circuit of `n` wires, without any other unitary gates applied on any of the wires. Its

```
data Unitary : Nat -> Type where
  IdGate   : {n : Nat} -> Unitary n
  H        : {n : Nat} -> (j : Nat) ->
             {auto prf : (j < n) = True} ->
             Unitary n -> Unitary n
  P        : (p : Double) ->
             {n : Nat} -> (j : Nat) ->
             {auto prf : (j < n) = True} ->
             Unitary n -> Unitary n
  CNOT     : {n : Nat} -> (c : Nat) -> (t : Nat) ->
             {auto prf1 : (c < n) = True} ->
             {auto prf2 : (t < n) = True} ->
             {auto prf3 : (c /= t) = True} ->
             Unitary n -> Unitary n
```

Figure 4: The Unitary data type

only argument, `n`, is an implicit argument – it can be omitted when calling the constructor and it will be often inferred by Idris.

The second constructor, `H`, should be understood as applying the Hadamard gate H to the `j`-th wire of some previously constructed unitary gate which is specified as the last argument. The first implicit argument, `n`, is simply the arity of the resulting unitary gate. The second implicit argument, `prf`, is a proof obligation that `j` is smaller than `n`. This ensures that the argument `j` identifies an existing wire of the previously constructed unitary (last argument) and therefore the overall definition is algebraically and physically sound. We note that the implicit argument `prf` may be removed from our implementation if we change the type of `j` to `Fin n`, the type of integers less than `n`. However, in our experience, Idris has better support for `Nat` than for `Fin` and for this reason we chose to keep the `prf` argument.

The third constructor, `P`, should be viewed as applying the $P(p)$ gate, where the real number $p \in \mathbb{R}$ is approximated by the term `p : Double`.[1] The remaining arguments serve the same purpose as those for `H`.

The fnal constructor, `CNOT`, should be understood as applying the CNOT gate, where `c` identifies the wire used for the control (the small black dot in Figure 1), `t` identifies the wire of the target (the crossed circle in Figure 1) and the last (unnamed) argument is the previously constructed unitary circuit on which we are applying CNOT. The remaining arguments are implicit and often do not have to be provided by the users: the argument `n` is the arity of the unitary; `prf1` and `prf2` ensure that `c` and `t` identify valid wires of the unitary circuit; `prf3` ensures that the control and target wires are *distinct* and therefore the overall application of CNOT is physically admissible.

In our representation of quantum unitary gates, we make use of the dependently-typed features of Idris to impose proof obligations on some of our constructors in order to guarantee that the representation makes sense in physical and algebraic terms. On first glance, this might seem like a big burden for the users of the library. However, in our experience Idris can often automatically discover the required proofs (without any assistance from the user) and we had to do very little manual theorem proving. This is discussed in detail in the next subsection.

[1]This approximation is not a serious limitation – in fault-tolerant quantum computing one usually replaces the $P(p)$ gate family with a single $T = P(\pi/4)$ gate and the resulting gate set suffices to approximate any unitary with *arbitrary* precision. So we may replace the `P` constructor with a simple `T` constructor.

3.2 Constructing Unitary Transformations

The `Unitary` type from Figure 4 comes equipped with four basic introduction rules that allow us to define high-level functions in Idris that can be used to construct complex unitary gates out of simpler ones. We discuss this here and we show that the proof obligations from Figure 4 are not severe and can be easily ameliorated.

First, we point out that auto-implicit arguments may often be inferred by Idris via suitable search. For example, if all the arguments are known statically, the required proofs will be discovered by Idris and the users do not have to manually provide them.

Example 3.2.1. The unitary gate depicted in the circuit from Figure 3 may be constructed in the following way:

```
toBellBasis : Unitary 2
toBellBasis = CNOT 0 1 (H 0 IdGate)
```

In this example, Idris is able to infer all the implicit arguments and there is no need to provide any proofs. If we do not satisfy one of the constraints, for example if we write `CNOT 1 1` above (which does not make physical sense), then we get the following error during type checking:

```
Error : While processing right hand side of
toBellBasis. Can't find an implementation for
not (== 1 1) = True.
```

An error is also reported if we provide a wire number larger than 1.

It is also useful to define standalone unitary gates (not algebraic type constructors) for the $H, P(r)$ and CNOT gates as follows:

```
HGate : Unitary 1
HGate = H 0 IdGate

PGate : Double -> Unitary 1
PGate r = P r 0 IdGate

CNOTGate : Unitary 2
CNOTGate = CNOT 0 1 IdGate
```

3.2.1 Composing Unitary Gates. Our libraries provide functions for sequential composition (`compose`) and parallel composition (`tensor`) of unitary gates and they have the following types:

```
compose : Unitary n -> Unitary n -> Unitary n
tensor : {n : Nat} -> {p : Nat} ->
         Unitary n -> Unitary p -> Unitary (n + p)
```

Notice that both functions do not impose any proof obligations on the user. This means that the primary algebraic way of composing unitary transformations may be done in Qimaera without any need for theorem proving.

Example 3.2.2. The `toBellBasis` gate from Example 3.2.1 may be equivalently expressed in the following way:

```
toBellBasis : Unitary 2
toBellBasis = CNOTGate `compose`
              (HGate `tensor` IdGate)
```

However, there are situations where the two functions above do not suffice. For instance, this can occur if we are given a unitary circuit D on n qubits and we wish to apply another unitary gate U on $k \leq n$ qubits to a set of the wires of D which are not consecutive but separated by some wires. For this reason, Qimaera provides the function `apply` whose type is as follows:

```
apply : {i : Nat} -> {n : Nat} ->
        Unitary i -> Unitary n ->
        (v : Vect i Nat) ->
        {auto _ : isInjective n v = True}
```

The `apply` function is used to apply a smaller unitary gate of size `i` to a bigger one of size `n`, giving the vector `v` of wire indices on which we wish to apply the smaller gate. It needs one auto-implicit proof which enforces the consistency requirement that all indices of the wires specified by `v` are pairwise distinct and smaller than `n`.

In fact, the `apply` function implements the most general notion of composition and both sequential and parallel composition can be realised as special cases using it. The importance of the vector `v` is that it determines how to apply the smaller unitary gate of arity `i` to *any selection* of `i` wires of the larger unitary circuit, and moreover, it also allows us to *permute* the inputs/outputs of the smaller unitary gate while doing so. More specifically, if the k-th entry of the vector `v` is the natural number p, then the k-th input/output of the smaller unitary gate will be applied to the p-th wire of the larger unitary circuit. This is best understood by example.

Example 3.2.3. Consider the following code sample:

```
U : Unitary 3
U = HGate `tensor` IdGate `tensor` (P pi)

apply_example : Unitary 3
apply_example = apply toBellBasis U v
```

where `v` is a vector of length two. Here `toBellBasis` is given in Example 3.2.1 and represents the circuit given below left and `U` represents the circuit given below right:

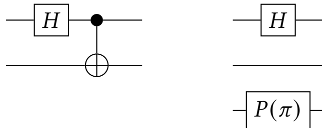


Table 1 shows what unitary gate is specified under different values of `v`. In these examples, Idris can automatically infer the required proofs and the user does not have to provide them.

Remark 3.2.4. Instead of using `apply`, there is another possible approach, in the spirit of *symmetric monoidal categories* [20, §XI], where we could add one extra introduction rule to the `Unitary` type in Figure 4 for representing *permutations* of wires. However, in our view, this approach is somewhat awkward, because one does not usually think of permutations (induced by the symmetric monoidal structure) as physical gates.

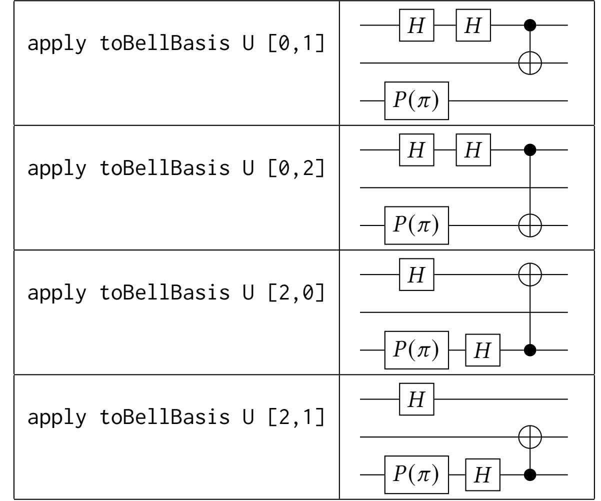


Table 1: Examples illustrating the `apply` function.

3.2.2 Adjoints of Unitary Gates. Qimaera also provides a function

```
adjoint : Unitary n -> Unitary n
```

which computes the adjoint (i.e. inverse) of a given unitary gate. As explained previously, one often has to apply the inverse of a given unitary gate, so having a high-level method such as this is useful. Our implementation uses the obvious algorithm for synthesising the adjoint. This may be used, for example, to automatically uncompute operations that we perform on ancilla qubits, which is often required by many quantum algorithms.

3.2.3 Controlled Unitary Gates. We also implement a function

```
controlled : {n : Nat} -> Unitary n ->
             Unitary (S n)
```

which given a unitary gate U constructs the corresponding controlled unitary gate. Our implementation uses the obvious algorithm for doing this, but more sophisticated algorithms may also be implemented in the future.

3.2.4 Optimisation of Unitary Gates. Unitary gates are represented in a scalable and compositional way in Qimaera. It is therefore possible to use Idris to define an optimisation function

```
optimise : Unitary n -> Unitary n
```

which optimises a given (very large) unitary gate with respect to some criterion. We have not done this yet, because this is not the focus of our work. The point we wish to make is that unitary gates in Qimaera may be analysed and manipulated like any other algebraic data type using the full capabilities of Idris.

3.3 Example: The Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a unitary transformation which may be seen as the quantum analogue of the classical Discrete Fourier Transform. This is a very important unitary operator and it is used in Shor's algorithm for integer factorisation. The unitary circuit which realises QFT on n qubits is shown in Figure 5, where $R_n \stackrel{\text{def}}{=} P\left(\frac{2\pi}{2^n}\right)$. The Qimaera code which realises this unitary

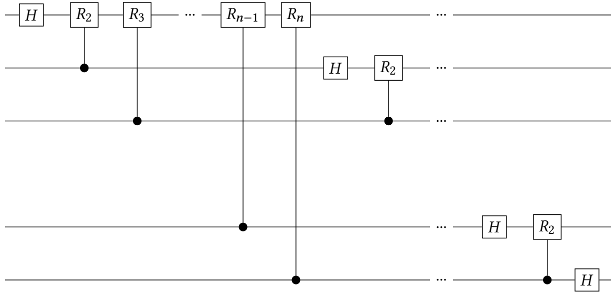


Figure 5: The QFT unitary gate on n qubits.

```
Rm : Nat -> Unitary 1
Rm m = let m' = cast m
       in PGate (2 * pi / (pow 2 m'))

cRm : Nat -> Unitary 2
cRm m = controlled (Rm m)

recursivePattern : (n : Nat) -> Unitary n
recursivePattern 0 = IdGate
recursivePattern 1 = HGate
recursivePattern (S (S k)) =
  let t = tensor (recursivePattern (S k)) IdGate
  in rewrite sym $ lemmaplusOneRight k
  in apply (cRm (S k)) t [S k,0]
           {prf = lemmaInj1 k}

qft : (n : Nat) -> Unitary n
qft Z = IdGate
qft (S k) =
  let g = recursivePattern (S k)
      h = tensor (IdGate {n = 1}) (qft k)
  in compose h g
```

Figure 6: Qimaera code for QFT.

gate is shown in Figure 6. Notice that here we make use of the `controlled` function from §3.2.3 in the function `cRm`, so that we can automatically implement the many controlled R_n gates that are required for the QFT gate. The required recursive pattern and recursive calls are easily expressed within Idris.

In this example, all the parameters are universally quantified, so we need a few very short proofs inside the code for using the `apply` function and one for correctly unifying the size of the circuit. These proofs are shown (in their entirety) in Figure 7 and they are very easy and simple. These proofs state the following simple facts (when translated mathematically): (1) $\forall k \in \mathbb{N}. k < k+1$; (2) if a is true and b is true, then $a \& b$ is true; (3) $\forall k \in \mathbb{N}. k+1 \neq 0$; (4) $\forall n \in \mathbb{N}. n+1 = S(n)$, where S is the successor function. Currently Idris cannot automatically discover these proofs, but we hope in the future its capabilities for proof search would improve to the point where it could. If this happens, then the users would not have to manually provide these proofs.

```
kLTSucc1 : (k : Nat) -> k < (k + 1) = True
kLTSucc1 0 = Refl
kLTSucc1 (S k) = kLTSucc1 k

lemmaAnd : {a : Bool} -> {b : Bool} ->
           (a = True) -> (b = True) ->
           (a && b = True)
lemmaAnd {a = True} {b = True} p1 p2 = Refl

lemmaInj1 : (k : Nat) ->
            isInjective (S (k + 1)) [S k, 0] = True
lemmaInj1 k = let p1 = kLTSucc1 k
              in lemmaAnd (lemmaAnd p1 Refl) Refl

lemmaplusOneRight : (n : Nat) -> n + 1 = S n
lemmaplusOneRight n =
  rewrite plusCommutative n 1 in Refl
```

Figure 7: All the lemmas needed for the QFT function

4 EFFECTFUL QUANTUM COMPUTATION

In the previous section we showed how unitary gates are represented in Qimaera. This suffices to capture the pure, deterministic and reversible fragment of quantum computation. However, as we explained in §2.1, we need to consider effectful, probabilistic and irreversible quantum processes (e.g. measurements) in order to recover the full power of quantum computation. In this section we show how this is implemented in Qimaera. In particular, we make heavy use of monads, linearity and dependent types in order to achieve this is a type-safe way.

Remark 4.0.1. As we mentioned in §1, since we do not personally have access to quantum hardware, all the quantum operations are simulated using the linear-algebraic representation of quantum states. However, from a high-level programming point of view, we believe our implementation addresses the same problems as the more realistic quantum-classical device scenario, which also uses the IO monad.

4.1 Representation of Quantum States

We now explain how the quantum program dynamics are represented in Qimaera in a type-safe way. We are (roughly) inspired by representing the notion of a *quantum configuration* as it appears in [19, 24], which is in turn used to formally describe the operational semantics of quantum type systems.

4.1.1 Qubits in Qimaera. Because of the possibility of quantum entanglement (see §2.1.7), we cannot describe the state of an individual qubit which is part of a larger composite system – we have to describe the state of the entire system. On the other hand, we wish to be able to refer to *parts* of the whole system by identifying specific qubit positions. In Qimaera, we introduce the following type declaration:

```
data Qubit : Type where
  MkQubit : (n : Nat) -> Qubit
```

The argument of type `Nat` is used as a *unique identifier* for the constructed qubit. The constructor `MkQubit` is *private* and users of our libraries cannot access it. Instead, our libraries provide functions (discussed later) that ensure that terms of type `Qubit` are always created with a fresh (i.e. unique) natural number that serves as its identifier. In fact, these functions are the only way users can access or manipulate qubits and, moreover, our users cannot access these unique identifiers. This allows us to formulate a representation where terms of type `Qubit` unambiguously refer to the relevant parts of larger composite systems. Therefore, a term of type `Qubit` should be understood as a pointer, or as a unique identifier, of a 1-qubit subsystem of some larger quantum state. Terms of type `Qubit` should *not* be understood as representing any sort of state, because they do not carry such information.

4.1.2 Quantum States in Qimaera. We represent quantum states in Qimaera via the following code:

```
data QuantumState : Nat -> Type where
  MkQuantumState : {n : Nat} ->
                   Matrix (power 2 n) 1 ->
                       Vect n Nat -> Nat ->
                       QuantumSTate n
```

The constructor is *private* and users cannot access it. Because of this, quantum states may be only be created and manipulated by a few functions that we provide and discuss later. The argument `n` is the number of qubits of the quantum state; the argument of type `Matrix` is a linear-algebraic column vector that describes the quantum state; the third argument contains the qubit identifiers and their positions in the quantum state; the final argument of type `Nat` represents the highest qubit identifier seen so far, and we use it to ensure that newly created qubits have unique identifiers.

4.1.3 Probabilistic Effects. As we discussed in §2.1.9, quantum measurements induce probabilistic computational effects which are inherited by the classical language. In order to incorporate these probabilistic effects, we follow the standard approach and use the IO monad. Furthermore, the use of the IO monad is also necessary in the more realistic classical-quantum device scenario.

Example 4.1.1. A monadic program which generates a random number, prints it on the screen, and then returns it as a result is given below:

```
printRandomNumber : IO Double
printRandomNumber = do
    nb <- randomIO ()
    putStrLn (show nb)
    pure nb
```

However, when representing quantum program dynamics, we also require *linearity*, but all the functions provided by the IO monad (e.g. `pure`) are *not* linear in any of their arguments. This creates a problem which may be solved by using the LIO library, which extends the IO monad with linearity. For more simplicity, we define `R` to be our linear IO monad:

```
R : Type -> Type
R = L IO {use = Linear}
```

```
data QStateT : Type -> Type -> Type -> Type where
  MkQST :
    (1 _ : (1 _ : initialType) ->
           R (LPair finalType returnType)) ->
    QStateT initialType finalType returnType

runQStateT : (1 _ : initialType) ->
             (1 _ : QStateT initialType finalType
                    returnType) ->
             R (LPair finalType returnType)

pure : (1 _ : a) -> QStateT t t a

(>>=) : (1 _ : QStateT i m a) ->
        (1 _ : ((1 _ : a) -> QStateT m o b)) ->
        QStateT i o b

QuantumOp : Nat -> Nat -> Type -> Type
QuantumOp n m t =
    QStateT (QuantumState n) (QuantumState m) t
```

Figure 8: The type of (effectful) quantum operations.

Then, by using `R` we can combine IO effects (and thus also probabilistic effects) and linearity in a suitable way.

4.1.4 Quantum States and Effects. Quantum computation is *effectful* and as we saw in §2.1 it depends on *quantum states*. Because of this, we define a *quantum state transformer* by combining several different concepts: *indexed state monads* [10][2], linearity and IO (and thus also probabilistic) effects. Our representation of these ideas in Qimaera is shown in Figure 8, where we omit some of the function definitions for brevity.

The type of biggest interest is the type `QuantumOp n m t`, which should be understood as a *quantum operation* from `n` qubits to `m` qubits which also produces a value of type `t` as a result to the user. This type allows us to specify (in Qimaera) all quantum operations that we need to recover full effectful quantum computation. In Figure 9, we show how the most important quantum operations are represented in Qimaera.

The function `newQubits` is used to prepare `p` new qubits in state $|0\rangle$ and the function returns a linear vector of length `p` with the qubit identifiers of the newly created qubits.

The function `applyUnitary` is used to apply a unitary operation of arity `i` to the qubits specified by the argument `LVect` (which also determines the order of application) and the operation returns an `LVect` which serves the same purpose – it identifies the qubits which were just modified by the unitary operator.

The `measure` function is used to measure `i` qubits identified by the `LVect` argument and it returns a (non-linear) value of type `Vect i Bool` that represents the result of the measurement.

Finally, the function `run` is used to *run* or to *execute* quantum operations (on the quantum device) which start and finish with the trivial quantum state (on zero qubits) and which produce some number of classical bits as a return result. This may be used to run

[2] See [2] for a Haskell implementation of this idea.

```
newQubits : (p : Nat) ->
            QuantumOp n (n+p) (LVect p Qubit)

applyUnitary : {n : Nat} -> {i : Nat} ->
               (1 _ : LVect i Qubit) -> Unitary i ->
               QuantumOp n n (LVect i Qubit)

measure : {n : Nat} -> {i : Nat} ->
          (1 _ : LVect i Qubit) ->
          QuantumOp (n+i) n (Vect i Bool)

run : QuantumOp 0 0 (Vect n Bool) ->
      IO (Vect n Bool)
```

Figure 9: Effectful quantum operations.

quantum algorithms: in a typical situation, we start with the trivial quantum state (on zero qubits), we prepare n qubits in state $|0\rangle$, we apply some unitary operations on them, and we finally measure all the qubits, thereby destroying all the qubits and producing n bits of classical information. This quantum algorithm is then represented as a value of type `QuantumOp 0 0 (Vect n Bool)`. Running it, however, produces a classical value of type `IO (Vect n Bool)`, because the execution is probabilistic and because our classical computer (on which we are running Idris) has to perform IO actions to communicate with the quantum device.

In fact, all of the above operations *implicitly modify quantum state* and may cause IO effects, because of the need to communicate with the quantum device. This is indeed reflected by our implementation. Observe, that `QuantumOp` is defined in terms of the `QStateT` monad transformer which does incorporate IO effects (via the `R` monad we discussed previously).

The two following examples are small functions that use the QStateT monad.

```
quantumOperation : QuantumOp 0 2 (LVect 2 Qubit)
quantumOperation = do
    [q1,q2] <- newQubits 2
    [s1] <- applyUnitary [q1] HGate
    [r1,r2] <- applyUnitary [s1,q2] CNOTGate
    pure [r1,r2]

quantumOperation2 : (1 _ : LVect 2 Qubit) ->
                    QuantumOp 2 2 (LVect 2 Qubit)
quantumOperation2 [q1,q2] = do
    [s1] <- applyUnitary [q1] HGate
    [r1,r2] <- applyUnitary [s1,q2] CNOTGate
    pure [r1,r2]
```

In this example, as indicated by the type of `quantumOperation`, we are performing a quantum operation from a state with no qubits to a state with two qubits. The user is returned a linear vector of two qubit pointers. Indeed, the function creates two qubits, applies some circuits on them, and returns their pointers. For the second function, `quantumOperation2`, we operate on a quantum state with already two qubits. Their pointers are given as input.

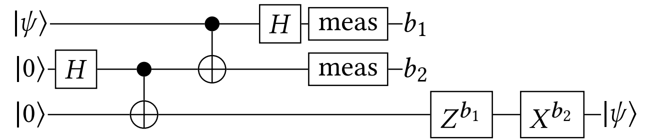


Figure 10: The teleportation protocol.

We provide the user with a `run` function which initializes a quantum state with no qubits, runs all operations within the State monad and finally discards the context once all qubits have been measured. Within this run function, the user is enforced to use only linear operations with quantum states and qubit pointers. We can for example run the previous functions :

```
run_example : IO (Vect 2 Bool)
run_example = run
  (do
    [q1,q2] <- quantumOperation
    [r2,r1] <- quantumOperation2 [q2,q1]
    measure [r1,r2]
  )
```

If we try to copy a qubit, or if we do not declare the qubits pointers to be linear in our functions, an error is raised by the compiler because the program does not comply with linearity :

```
linearity_example : IO (Vect 2 Bool)
linearity_example = run
  (do
    [q1,q2] <- newQubits
    measure [q2,q2]
  )

>> Error
While processing right hand side of
linearity_example. There are 2 uses
of linear name q2
```

4.2 Example : The Quantum Teleportation Protocol

In this section, we describe a first simple algorithm : the quantum teleportation protocol [12, 23]. It is used to transport quantum information, typically a qubit, from one place to another (see Figure 10). All the parameters are known statically. The interesting part is the interaction between classical and quantum data : the last part of the circuit consists in unitary corrections that depend on the results of the measurements of the first two qubits.

The code is given Figure 11. We first create the unitary operators, `circuitTeleportation` and `unitaryCorrection`. They are mathematical objects, and the second one depends on two bits of classical information, the results of a previous measurement. It is only in the next function, `teleportation`, that we make quantum operations by applying these circuits on some qubits and measuring the qubits. The quantum operation starts from a state with one qubit to a state with one qubit and returns the pointer to this qubit.

Finally, we run the teleportation protocol with the function `runTeleportation` starting with a qubit in a superposition state.

```
circuitTeleportation : Unitary 3
circuitTeleportation =
  H 0 (CNOT 0 1 (apply toBellBasis IdGate [1,2]))

unitaryCorrection : Bool -> Bool -> Unitary 1
unitaryCorrection b1 b2 =
  (if b2 then XGate else IdGate)
  `compose`
  (if b1 then ZGate else IdGate)

teleportation : (1 _ : Qubit) ->
                QuantumOp 1 1 Qubit
teleportation q0 = do
  [q1,q2] <- newQubits 2
  [q0,q1,q2] <-
    applyUnitary [q0,q1,q2] circuitTeleportation
  [b1,b2] <- measure [q0,q1]
  [q] <-
    applyUnitary [q2] (unitaryCorrection b1 b2)
  pure q

runTeleportation : IO Bool
runTeleportation = run
  (do
    q <- newQubit
    [q] <- applyUnitary [q] HGate
    q <- teleportation q
    measureQubit q
  )
```

Figure 11: The Quantum Teleportation Protocol

In this function, linearity is enforced, and this is why we have no choice but specify the linearity of the qubit pointer in the `teleportation` function.

5 VARIATIONAL QUANTUM PROGRAMMING

In this section, we explain why Qimaera is suitable for variational quantum programming and we implement the variational quantum eigensolver to show how classical and quantum computations can be mixed in an elegant way.

Variational quantum programming is a back-and-forth process between quantum and classical operations: the classical part of the algorithm uses the result of the qubit measurements to compute the parameters for the quantum part. As we implement Qimaera in an already existing programming language, Idris 2, we already have a good support for classical computations.

Moreover, the two kinds of operations, classical and quantum, can interact well as we implement two distinct types of operations in Qimaera. We separate unitary circuits generation, which is a mathematical operation that can be performed on a classical device, from effectful operations such as qubits creation, circuits application and measurements that are computed on a quantum device. This separation is particularly noticeable in the code of a variational algorithm as all the effectful operations on quantum states

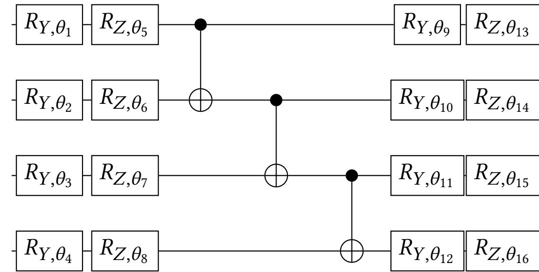


Figure 12: VQE ansatz for 4 qubits with a depth of 1

occur inside the `run` function. Outside this function, only classical operations are allowed so there are no restrictions on the usage of the information. On the contrary, within the state monad context, linearity is enforced to make correct usage of quantum resources so that laws of quantum physics are respected. It is then possible to correctly mix quantum and classical information even if they behave in very different ways. As creating circuits is still a classical operation performed using the `Unitary` data type, unitary gates can be parametrized by the results of classical computation (for instance in the teleportation protocol, the unitary corrections rely on a classical `if` statement depending on the result of the previous measurement). Furthermore, as the results on measurements are bits of classical information sent through the IO interface, they can be used directly for classical computations. This results in an elegant and natural way of combining classical and quantum operation to make them work in tandem to solve variational problems.

5.1 The Variational Quantum Eigensolver

The variational quantum eigensolver (VQE) is a hybrid classical-quantum algorithm that finds an upper bound of the lowest eigenvalue of a Hamiltonian matrix. In quantum physics, the Hamiltonian is an operator which describes the energy of a quantum system. Finding its lowest eigenvalue is equivalent to finding the ground-state energy of the system.

The quantum part of the algorithm consists in applying a unitary circuit, the ansatz (see Figure 12 for an ansatz for four qubits), parametrized by classical information. The classical part is an optimizer that generates the set of control parameters for the ansatz. Here, as we are interested in showing the interaction between quantum and classical computation, we do not implement the classical part. Instead of optimizing the parameters, our classical functions generate random numbers, but their input is still the result of the quantum measurements.

The code for the VQE is given Figure 13. The `ansatz` function builds the circuit represented Figure 12. It is only the mathematical unitary gate, it will be applied later, in the `VQE'` function. Here, we choose an ansatz with a linear entanglement. The arguments of the `ansatz` function are the number of qubits, n, the depth of the ansatz (which is the number of repetitions of the pattern rotations gates - linear entanglement) and two vectors of size $n \times (depth + 1)$ of parameters for the rotation gates R_Y and R_Z respectively. These parameters are optimized by the classical part. The three first functions, `linearEntanglement`, `tensorRz`, and `tensorRy` build the

```
linearEntanglement : (n : Nat) -> Unitary n

tensorRz : (n : Nat) -> Vect n Double -> Unitary n

tensorRy : (n : Nat) -> Vect n Double -> Unitary n

ansatz : (n : Nat) -> (depth : Nat) -> Vect (S depth) (Vect n Double) ->
         Vect (S depth) (Vect n Double) -> Unitary n
ansatz 0 _ _ _ = IdGate
ansatz (S n) 0 [v] [w] = tensorRz (S n) w @@ tensorRy (S n) v
ansatz (S n) (S r) (v :: vs) (w :: ws) =
  let circ1 = ansatz (S n) r vs ws
  in circ1 @@ linearEntanglement (S n) @@ tensorRz (S n) w @@ tensorRy (S n) v

pretendComputeEnergy : Vect n Bool -> IO Double

pretendClassicalWork : (n : Nat) -> (depth : Nat) -> (resultAnsatz : Vect n Bool) ->
                       (hamiltonian : Vect (power 2 n) (Vect (power 2 n) (Complex Double))) ->
                       IO (Vect (S depth) (Vect n Double), Vect (S depth) (Vect n Double))

VQE' : (n : Nat) -> (hamiltonian : Vect (power 2 n) (Vect (power 2 n) (Complex Double))) ->
       (nbIter : Nat) -> (depth : Nat) -> IO (Vect n Bool)
VQE' n _ 0 depth = pure (replicate n False)
VQE' n m (S k) depth = do
  v <- VQE' n m k depth
  (xs,ys) <- pretendClassicalWork n depth v m
  run (do
    let c = ansatz n depth xs ys
    q <- newQubits n
    q <- applyCircuit q c
    measure2 q)

VQE : (n : Nat) -> (hamiltonian : Vect (power 2 n) (Vect (power 2 n) (Complex Double))) ->
      (nbIter : Nat) -> (depth : Nat) -> IO Double
VQE n m k d = do
  res <- VQE' n m k d
  pretendComputeEnergy res
```

Figure 13: Code for the variational Quantum Eigensolver (VQE)

small components of the circuit, and the `ansatz` function composes them. No theorem proving is needed for this unitary circuit.

The classical part takes as input the result of the measurement of the quantum state. The function `pretendComputeEnergy` pretends to compute the energy of the state using the results of the measurements. The function `pretendClassicalWork` pretends to optimize the parameters of the ansatz given the results of the measurements. Its parameters are the number of qubits, n, the depth of the ansatz, and the Hamiltonian matrix of the problem. It outputs two vectors of $n \times (depth + 1)$ parameters for the R_y and R_z rotations.

The `VQE'` function computes the interaction between classical and quantum operations. Its parameters are the number of qubits, the Hamiltonian matrix of the problem, the number of iterations of the algorithm and the depth of the ansatz. It returns the results of the quantum operations at each iteration. The classical part is done by the `pretendClassicalWork` function, and the quantum part is completely done inside the run function. The last function, `VQE`, has the same parameters as `VQE'` but returns the eigenvalue of the Hamiltonian matrix after all the computations.

6 RELATED WORK

We will compare Qimaera to two broad classes of programming languages for quantum computing: (1) previously existing classical programming languages for which libraries for quantum programming were later developed; and (2) standalone quantum programming languages. With respect to these two broad classes, Qimaera clearly falls in the first one.

Examples in the first class include Google's Cirq [1] (Python libraries), IBM's Qiskit [5] (Python libraries), Rigetti's pyQuil [3] (Python libraries) and Quipper [7] (Haskell libraries). These libraries

offer a wide-range of interesting programming features, however, none of them are type safe and so it is possible to write erroneous quantum programs which are not detected by the type system during type checking (i.e. at compile time).

Languages in the second class include Microsoft's Q♯ [4], Silq [8] and Proto-Quipper-D [16]. Q♯ is not type safe, but the other two languages are. Proto-Quipper-D is a linear and dependently-typed programming language designed for circuit-based programming. However, it does not currently support dynamic lifting (i.e. it does not support effectful quantum programming), so it is currently unsuitable for programming of variational quantum algorithms. Silq is a nice language whose main feature is automatic uncomputation of temporary values, but it has only basic support for classical programming features and it does not support general recursion, so it also is unsuitable for variational quantum programming.

7 CONCLUSION AND FUTURE WORK

In this paper we showed how to implement libraries for (variational) quantum programming in Idris 2. Our set of libraries, called Qimaera, make heavy use of linearity and dependent types in order to ensure the type-safety of our approach to (variational) quantum programming. In §3, we showed how to represent quantum unitary transformations in Idris as an algebraic data type. This allows us to adopt a high-level algebraic and scalable approach to the reversible fragment of quantum computation and we showed how we can analyse, manipulate and transform such unitary circuits via high-level methods which are implemented in Idris. In §4, we showed how full (effectful) quantum computation may be represented in Idris using its advanced type system. Dependent types allow us to correctly represent the compositional nature of quantum operations. Linearity allows us to correctly model the quantum program dynamics and to ensure the proper consumption and manipulation of quantum resources. The induced computational effects are properly encapsulated via the IO monad which is used for the communication between the classical device, on which Idris is running, and the quantum device. We showed that our approach is suitable by demonstrating how variational quantum algorithms may be implemented in §5. To the best of our knowledge, this is the first programming language that has full support for both high-level classical and quantum programming features and that also is type-safe.

As part of future work, we intend to evaluate Qimaera by implementing more (variational) quantum algorithms and compiling an experience report which describes our findings. Another interesting issue is to consider whether *quantum control* can be implemented in Qimaera in some form. In our current implementation, we only support *classical control*, which means that the choice function that determines subsequent (quantum) dynamics is always determined based on available classical information (which may be extracted through quantum measurement). With quantum control, however, the choice function may be more general and this allows us, for example, to *derive* the behaviour of the CNOT gate, rather than assume it as a built-in constant (as we have done here). We believe that this is a complicated matter and we leave this question for future work.

REFERENCES

[1] [n.d.]. Cirq Website. https://quantumai.google/cirq. Accessed: 13.08.2021.
[2] [n.d.]. Indexed State Monad Blog Post. https://kseo.github.io/posts/2017-01-12-indexed-monads.html. Accessed: 13.08.2021.
[3] [n.d.]. Pyquil Website. https://pyquil-docs.rigetti.com/en/stable/. Accessed: 13.08.2021.
[4] [n.d.]. Q# Website. https://azure.microsoft.com/en-gb/resources/development-kit/quantum-computing. Accessed: 13.08.2021.
[5] [n.d.]. Qiskit Website. https://qiskit.org/. Accessed: 13.08.2021.
[6] [n.d.]. Quantum Flagship Website. https://qt.eu/. Accessed: 24.02.2021.
[7] [n.d.]. Quipper Website. https://www.mathstat.dal.ca/~selinger/quipper/. Accessed: 13.08.2021.
[8] [n.d.]. Silq Website. https://silq.ethz.ch/. Accessed: 13.08.2021.
[9] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[10] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X
[11] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189
[12] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters* 70, 1895 (1993).
[13] P.N. Benton. 1995. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic: 8th Workshop, CSL '94, Selected Papaers.* https://doi.org/10.1007/BFb0022251
[14] P. N. Benton and P. Wadler. 1996. Linear Logic, Monads and the Lambda Calculus. In *LICS 1996.*
[15] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs)*, Anders Møller and Manu Sridharan (Eds.), Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9
[16] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2020. A Tutorial Introduction to Quantum Circuit Programming in Dependently Typed Proto-Quipper. In *Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings (Lecture Notes in Computer Science)*, Ivan Lanese and Mariusz Rawski (Eds.), Vol. 12227. Springer, 153–168. https://doi.org/10.1007/978-3-030-52482-1_9
[17] J.-Y. Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1 – 101.
[18] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Phys. Rev. Lett.* 103 (Oct 2009), 150502. Issue 15. https://doi.org/10.1103/PhysRevLett.103.150502
[19] Xiaodong Jia, Andre Kornell, Bert Lindenhovius, Michael W. Mislove, and Vladimir Zamdzhiev. 2021. Semantics for Variational Quantum Programming. *CoRR* abs/2107.13347 (2021). arXiv:2107.13347 https://arxiv.org/abs/2107.13347
[20] Saunders Mac Lane. 1998. *Categories for the Working Mathematician (2nd ed.).* Springer.
[21] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12
[22] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. 2016. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics* 18, 2 (2016), 023023.
[23] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press. https://doi.org/10.1017/CBO9780511976667
[24] Romain Péchoux, Simon Perdrix, Mathys Rennela, and Vladimir Zamdzhiev. 2020. Quantum Programming with Inductive Datatypes: Causality and Affine Type Theory. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020 (Lecture Notes in Computer Science)*, Vol. 12077. Springer, 562–581. https://doi.org/10.1007/978-3-030-45231-5_29
[25] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 1–7.
[26] Peter W. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (1999), 303–332. https://doi.org/10.1137/S0036144598347011
[27] William K Wootters and Wojciech H Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803.