Type-safe Quantum Programming in Idris

Liliane-Joy Dandy, Emmanuel Jeandel, Vladimir Zamdzhiev

Type-safe Quantum Programming in Idris

Liliane-Joy Dandy[1,2,3], Emmanuel Jeandel[3], and Vladimir Zamdzhiev[3,4]

[1] EPFL, Lausanne, Switzerland
[2] École polytechnique, Palaiseau, France
[3] Université de Lorraine, CNRS, Inria, LORIA, F 54000 Nancy, France
[4] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
{liliane-joy.dandy,emmanuel.jeandel,vladimir.zamdzhiev}@{epfl.ch,loria.fr,inria.fr}

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 programming language that enable the programmer to implement hybrid classical-quantum algorithms where the full power of the elegant Idris language works in synchrony with quantum programming primitives. The two key ingredients of Idris that make this possible are (1) dependent types which allow us to implement unitary quantum operations; and (2) linearity which allows us to enforce fine-grained control over the execution of quantum operations so that we may detect and reject many physically inadmissible programs. We also show that Qimaera is suitable for variational quantum programming by providing implementations of two prominent variational quantum algorithms – QAOA and VQE.

1 Introduction

Variational Quantum Algorithms [30,25,13] present a computational paradigm where hybrid classical-quantum algorithms 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 passed onto the classical device 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 desired termination condition is satisfied.

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

Source code for Qimaera [1] and a full version of the paper [12] are available.

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 is that quantum information behaves very differently compared to classical information. As an example, quantum information cannot be copied in a uniform way [36], 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* [16,8,7,24,6] 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.

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 language [10] that allow the programmer to implement hybrid quantum-classical algorithms 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 [24,6] 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. 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 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 a representation of a (perhaps simple) computational environment for hybrid quantum-classical programming. We design a suitable (abstract) interface that allows us to model this situation accurately and which makes use of the IO monad. However, since the authors do not personally have any quantum hardware, we provide only one concrete implementation of our interface that simulates the relevant quantum operations on our classical computers by using the proper linear-algebraic formalism, but while still using the IO monad as prescribed by the abstract interface. From a high-level programming perspective, the abstract interface addresses the programming challenges induced by the classical-quantum device scenario, but it ignores lower-level considerations (e.g., error correction).

We emphasise that we can achieve type-safe hybrid quantum-classical programming in an *existing* 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 first-class support for classical programming features. We think our chosen language, Idris, is such a language. The advanced type system of Idris allows us to elegantly mix quantum and classical programming primitives and therefore allows us to achieve our objectives. We demonstrate that Qimaera is suitable for variational quantum programming by providing implementations of the two most prominent variational quantum algorithms – QAOA and VQE. Moreover, our implementation of these algorithms has been achieved in a *type-safe* programming framework. By this we mean that common quantum programming errors (copying of qubits, applying a CNOT operation with the same source and target, etc.) are *statically* detected and rejected by the Idris type checker. We also note that being able to combine quantum and classical programming is important in other scenarios too (for instance in quantum cryptography).

Quantum Circuits vs Recursive Quantum Programs. We want to stress that the focus of our paper is not about quantum circuits, but about (recursive) quantum *programs and algorithms*. While some quantum algorithms may be seen as quantum circuits, there are algorithms which are more general, for example, repeat-until-success (see §5.2) and variational quantum algorithms (see §6). Such algorithms are not quantum circuits in the traditional understanding of this notion, and for them general recursion, probabilistic effects and classical computation might be important.

More specifically, general recursion is important, because many existing quantum algorithms are *probabilistic* and find the correct answer with some probability. General recursion then allows the programmer to repeatedly run such an algorithm until the correct solution is found, thereby resulting in an almost-surely-terminating program, i.e., a program that terminates with probability 1. However, since there is no upper bound on the number of runs of the algorithm, general recursion is necessary to express this pattern. For instance, this can be used to repeatedly run Shor's algorithm until the algorithm succeeds in finding a divisor. This might also be useful for variational quantum algorithms, because it allows us to express more flexible termination conditions, which give us more than simple iterations.

Safety Properties. We consider type safety in quantum programming to be important, because it is easy to make mistakes where one can copy qubits or forget to use a qubit. The former is physically inadmissible due to the *no-cloning theorem* of quantum mechanics [36] and the latter usually leads to unexpected behaviour, because discarding quantum information causes a *side effect* that

may affect the rest of the quantum system. These observations suggest that we may design our systems and libraries carefully, by utilising *linear typing features*, so that these situations can be *statically* detected and rejected by the type system, therefore avoiding the problem. Otherwise, such situations could result in *runtime errors* (e.g., copying a qubit), which are clearly undesirable. In fact, in our experience, it is very easy to make such mistakes and this happened while we were implementing some of the quantum algorithms described in this paper. Our type-safe approach to quantum programming automatically detects and rejects these kinds of erroneous programs during type checking. While we do not have any proof of correctness, we believe that our approach is type-safe as long as the users do not modify our library files.

Why Idris instead of another language? The features that we require to achieve our objectives are: general recursion, dependent types and linearity. We chose Idris 2, because it is an excellent language that has all three of these features. Removing general recursion limits the expressivity of the language (as explained above). The other two features are used to reject erroneous quantum programs. We think that most programming languages that have the three features mentioned above are suitable for type-safe hybrid quantum-classical programming. In fact, one of the main points that we wish to demonstrate with this paper is that it is *not* necessary to build a standalone programming language in order to achieve the desired safety properties. Instead, the same can be achieved with already *existing* languages, such as Idris 2. This approach has some advantages (compared to designing a standalone language), such as: easier maintenance, larger library support, better integration with the newest developments in classical programming, etc.

Publication History. This HAL version of the paper was updated in 2025 to add an acknowledgement.

2 Background on Quantum Computation

Readers interested in a detailed introduction to quantum computing may consult [26]. In this section we summarise the basic notions that are relevant for our development.

The simplest non-trivial quantum system is the *quantum bit*, often abbreviated as *qubit*. Qubits 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 $|0\rangle \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, which may be seen as representing the classical bits 0 and 1. An arbitrary qubit is described by $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ where $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$.

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,

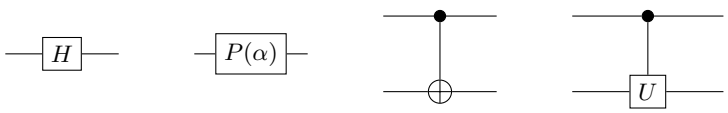


Fig. 1. The Hadamard, Phase Shift, CNOT and CU gates.

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.

The state space that describes a system of n qubits is the Hilbert space $\mathbb{C}^{2^n}$. 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.

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 their spatial structure.

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 $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ 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: $P(\alpha)|0\rangle = |0\rangle$ and $P(\alpha)|1\rangle = e^{i\alpha}|1\rangle$ 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 $\text{CNOT}|00\rangle = |00\rangle$; $\text{CNOT}|01\rangle = |01\rangle$; $\text{CNOT}|10\rangle = |11\rangle$ and $\text{CNOT}|11\rangle = |10\rangle$ and this unitary map may be used to generate quantum entanglement.

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 and parallel composition amounts to juxtaposition.

The CNOT gate is the simplest example of a *controlled unitary gate*. Given a unitary gate $U\colon \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, the controlled-$U$ unitary gate is the unitary gate $CU\colon \mathbb{C}^{2^{n+1}} \to \mathbb{C}^{2^{n+1}}$ whose action is determined by the assignments $CU(|0\rangle \otimes$

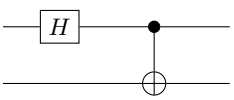


Fig. 2. A quantum circuit that may be used for the preparation of the Bell state.

$|\psi\rangle) = |0\rangle \otimes |\psi\rangle$ and $CU(|1\rangle \otimes |\psi\rangle) = |1\rangle \otimes (U\,|\psi\rangle)$. Controlled unitary operations are ubiquitous in quantum computing (see Figure 1 for their circuit depiction).

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 (i.e., the *adjoint*) of a given unitary map is ubiquitous.

A quantum state $|\psi\rangle \in \mathbb{C}^{2^n}$, with $n > 1$, is said to be *entangled* when there exists no non-trivial decomposition $|\psi\rangle = |\phi\rangle \otimes |\tau\rangle$. 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. A very important example of an entangled state is the *Bell state* given by $|\text{Bell}\rangle \stackrel{\text{def}}{=} \frac{|00\rangle + |11\rangle}{\sqrt{2}}$.

Preparing a new qubit in state $|0\rangle$ is an admissible physical operation. This, together with application of unitary gates as part of the computation, allows us to prepare arbitrary quantum states, e.g., the Bell state can be prepared by taking $|\text{Bell}\rangle = (\text{CNOT} \circ (H \otimes I))\,|00\rangle$ (see Figure 2).

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 $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, there are two possible outcomes: either the quantum system will collapse to state $|0\rangle$ and we obtain the classical bit 0 as evidence of this event, or, the quantum system will collapse to state $|1\rangle$ 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*.

Unlike classical information, quantum information cannot be uniformly copied. This is made precise by the *no-cloning* theorem [36]. There exists no unitary operation $U : \mathbb{C}^4 \to \mathbb{C}^4$, such that for every qubit $|\psi\rangle : U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$. This means that copying of quantum information is a *physically inadmissible* operation. Ideally, quantum programming languages should be designed so that these kinds of errors are detected during type checking.

3 Background on the Idris 2 Language

In this section, we give a short overview of the Idris 2 language and its main features that are relevant for the development of Qimaera. Idris 2 is a functional language with a syntax influenced by that of Haskell. The features of particular interest for us are dependent types and linearity, both of which are crucial for Qimaera. Its type system is based on Quantitative Type Theory [24,6], which specifies how dependent types and linearity are combined.

Dependent Types. In Idris, types are first-class primitives and they may be manipulated like other constructs of the language. This allows us to formulate more expressive types that can depend on values, and hence it enables us to make some properties and program invariants explicit.

Example 1. The type of vectors is a simple and useful example of a dependent type. A vector is a list with a fixed length that is part of the type. It can be defined as follows, where `S` is the successor function for natural numbers, and `a` is a polymorphic type:

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

The type `Vect` has two constructors (i.e., introduction rules). The first one constructs the empty vector, of length zero. The second one is used to introduce non-empty vectors: a vector with `k+1` elements of type `a` is constructed by combining an element of type `a` and a vector of size `k`.

Type dependency allows us to specify useful program properties and type checking ensures that they hold. For instance, we can define an `append` function that concatenates two vectors. Then, the size of the output vector is the sum of the sizes of the input vectors and this is specified by its type.

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

This information allows the language to detect a larger class of programming errors. Note that type dependency information is not available for the analogous function on lists. Type dependency may also be used to express constraints on the inputs of a function, e.g., we can define a *total* function, called `pop`, that cannot be applied to an empty vector.

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

Writing "`pop []`" is now an error which is detected statically, rather than dynamically, and we note that the same cannot be achieved if we were to replace vectors with lists.

Linearity. The type system of Idris 2 is based on Quantitative Type Theory, where every function argument is associated with a multiplicity that states the number of times the variable is used at runtime[5]. This multiplicity can be 0, 1 or ω. An argument with multiplicity 0 is only used at compile time (to determine type dependency information) and is erased at runtime. A *linear* argument has multiplicity 1 and it is used exactly once at runtime. Finally, ω represents the unrestricted multiplicity, which is default, where the function argument may be used any number of times.

Example 2. Consider the `pop` function which we just discussed. The (implicitly bound) variables `k` and `a` have multiplicity 0, because they are not explicitly specified as separate arguments, and they are *not* accessible at runtime in the function. The variables `x` and `xs`, which are explicitly bound, have the default (unrestricted) multiplicity.

Example 3. An important type which we define in Qimaera is the type of linear vectors, which we write as `LVect`. The only difference, compared to the standard vectors in Idris, is that the `(::)` constructor for `LVect` is a linear function in all of its arguments. Linearity in Idris 2 is specified by writing the multiplicity 1 in front of each argument.

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

We also use linear pairs that are already defined in Idris 2.

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

Linearity allows us to specify and enforce constraints on function arguments, e.g., it prevents us from duplicating data, so the function definition below leads to an error:

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

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

Linearity is prominently used in Qimaera. In particular, when manipulating quantum data, linearity is enforced in order to properly handle quantum resources and comply with the laws of quantum mechanics.

Remark 1. We learned only recently that there is a type of linear vectors in the Idris libraries. In the future we might replace our implementation with the one provided by the Idris developers.

[5] This can be understood similarly to how variables are used in linear λ-calculi.

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

Fig. 3. The Unitary data type (file: `Unitary.idr`).

4 Unitary Operations in Qimaera

We describe our representation of unitary transformations in Qimaera as an algebraic data type called `Unitary`. Every value of this type is, by design, an *algebraic decomposition* of a unitary operation in terms of the atomic unitary gates that we selected in §2.

The `Unitary` data type allows us to adopt a high-level *algebraic* and *scalable* approach towards the reversible fragment of quantum computation. This provides the programmer with some benefits as we show in this section. However, using the `Unitary` data type is actually entirely optional. Users who are interested in effectful quantum programming do *not* have to use it (see §5) and they may still do hybrid classical-quantum programming, but at the cost of losing the algebraic decomposition of unitary operations. However, there are many useful functions that are available for manipulating values of type `Unitary` that are not available for effectful quantum programs.

4.1 The Unitary Data Type

Quantum unitary operations admit an algebraic representation based on the atomic gates from the universal gate set we described. Our idea for the representation of unitary operations is based on this, or equivalently, on how unitary operations may be expressed in terms of unitary quantum circuit diagrams. Because of these reasons, linearity is not required for our formalisation of unitary operations. The code for the `Unitary` data type is listed in Figure 3 and we now describe our representation in greater detail.

Given a natural number `n : Nat`, the type of unitary operations on `n` qubits is given by `Unitary n`. Note that `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 operation on `n` qubits. Diagramatically, we can see this as constructing a circuit of `n` wires, without applying any other gates on any of the wires. It has a unique argument, `n`, which is implicit – it can be omitted when calling the `IdGate` constructor and it will often be inferred by Idris.

The second constructor, `H`, should be understood as applying the Hadamard gate H to the `j`-th qubit of some previously constructed unitary circuit which is specified as the last argument. The first implicit argument, `n`, is simply the arity of the resulting unitary operation. 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 circuit (last argument) and therefore the overall definition is algebraically and physically sound. We think that the implicit argument `prf` may be removed from our implementation if we change the type of `j` to `Fin n`, the type of natural numbers less than `n`. However, in our experience, we found it easier to work with the current implementation rather than with `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`.[6] The remaining arguments serve the same purpose as those for `H`.

The final 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: 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 and algebraically admissible.

In our representation of quantum unitary operations, we make use of type dependency to impose proof obligations on some of our constructors in order to guarantee that the representation makes sense in physical and algebraic terms. Indeed, this might sometimes be a burden for the users of the library. However, Idris can sometimes automatically infer the required proofs without any assistance from the user, e.g., when all arguments are statically known constants (see Example 4). This is discussed in detail in the next subsection.

4.2 Constructing Unitary Transformations

The four basic introduction rules of the `Unitary` type allow us to define *high-level functions* in Idris that can be used to construct complex unitary circuits out of simpler ones. We discuss this here and we show that the proof obligations

[6] This approximation is not a big 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 achieve approximation with *arbitrary* precision. So we can easily replace `P` with a `T` constructor.

from Figure 3 can sometimes be ameliorated and sometimes even completely sidestepped.

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

Example 4. The unitary circuit from Figure 2 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, e.g., 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 also is reported if we provide a wire number larger than 1. It also is useful to define *standalone* unitary gates 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
```

Composing Unitary Circuits. Our libraries provide functions for sequential composition (`compose`) and parallel composition (`tensor`) of unitary operations:

```
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 require proof obligations like the ones from Figure 3. This means that one of the main algebraic ways for composing unitary operations may be done without requiring such proofs. The use of these functions is ubiquitous in practice and we introduce the infix synonyms `(.)` and `(#)` for `compose` and `tensor`, respectively.

Example 5. The `toBellBasis` gate from Example 4 may be equivalently expressed in the following way:

```
toBellBasis : Unitary 2
toBellBasis = CNOTGate . (HGate # IdGate)
```

Qimaera provides another, more general, form of composition via 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} ->
        Unitary n
```

The `apply` function is used to apply a smaller unitary circuit 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 circuit. 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 that we support. 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 circuit 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 circuit 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 circuit will be applied to the p-th wire of the larger unitary circuit. This is best understood by example.

Example 6. Consider the following code sample:

```
U : Unitary 3
U = HGate # IdGate {n = 1} # (PGate 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 4 and represents the circuit given below left; `U` represents the circuit given below right:

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

Remark 2. Instead of using `apply`, there is another possible approach, in the spirit of *symmetric monoidal categories* [23, §XI], where we could add one extra introduction rule to the `Unitary` type for representing *permutations* of wires. However, in our view, this approach is less appealing, because one does not usually think of permutations (induced by the symmetric monoidal structure) as physical gates.

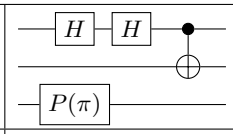
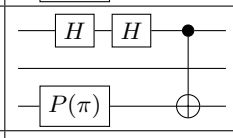
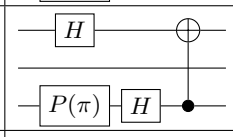
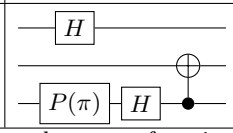
`apply toBellBasis U [0,1]`	
`apply toBellBasis U [0,2]`	
`apply toBellBasis U [2,0]`	
`apply toBellBasis U [2,1]`	

Table 1. Examples illustrating the `apply` function.

Adjoints of Unitary Circuits. Qimaera also provides a function

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

which computes the adjoint (i.e., inverse) of a given unitary circuit. One often has to apply the inverse of a given unitary circuit, so having a method such as this one is useful. Our implementation uses the standard approach for synthesising the adjoint. The adjoint may be used, for example, to uncompute the result of the application of unitary gates on auxiliary qubits.

Controlled Unitary Circuits. We also implement a function

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

which given a unitary circuit U constructs the corresponding controlled unitary circuit CU. Our implementation uses the standard and simple algorithm for doing this, but more efficient algorithms may also be implemented in principle.

Analysis of Unitary Circuits. Unitary circuits are represented in a scalable way in Qimaera and we can use Idris to optimise them. In particular, the function:

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

may be used to optimise a given unitary circuit by reducing the number of gates while keeping the action of the circuit unchanged. So far, this function provides only very basic optimisations, but more sophisticated and powerful ones may be added in principle. The point we wish to make is that unitary circuits in Qimaera may be analysed and manipulated like other algebraic data

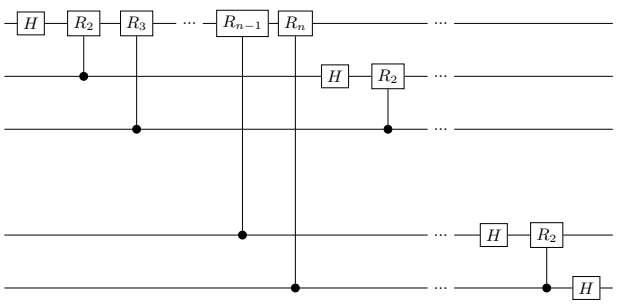


Fig. 4. The QFT unitary circuit on n qubits.

type structures using the capabilities of Idris. In fact, the file `Unitary.idr` also provides other functions that do this. For example, we provide functions for calculating the circuit depth, calculating the number of specific atomic gates used by a circuit, drawing circuits in the terminal and exporting circuits to Qiskit so that users may then use external analysis tools.

4.3 Example: The Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is an important unitary operator that is used in Shor's polynomial-time algorithm for integer factorisation [34]. The unitary circuit which realises QFT on n qubits is shown in Figure 4, where $R_n \stackrel{\text{def}}{=} P\left(\frac{2\pi}{2^n}\right)$. The Qimaera code which implements this unitary circuit is shown in Figure 5. Notice that we make use of the `controlled` function from §4.2 in the function `cRm`, so that we can implement the controlled R_n gates that are required. In this example, we have parameters that are universally quantified, so we need a few proofs in the code: one for using the `apply` function and one for correctly unifying the size of the circuit. These proof obligations appear when writing the `qftRec` function and Idris did not infer them automatically, so we had to provide the proofs. To get some intuition for the code: the `qftRec` function computes the recursive pattern that applies a Hadamard gate followed by the cascade of controlled R_n gates; the `qft` function then computes the other recursive pattern which consists in repeatedly using the pattern computed by `qftRec` and composing as appropriate.

5 Effectful Quantum Computation

In the previous section we showed how unitary circuits can be represented in Qimaera. This suffices to capture the pure, deterministic and reversible fragment of quantum computation. However, we need to also consider effectful and probabilistic quantum processes which may result from quantum measurements, because this is important for hybrid quantum-classical computation. In this section, we show how this can be done in a type-safe way by using monads, linearity and dependent types.

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

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

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

qft : (n : Nat) -> Unitary n
qft 0 = IdGate
qft (S k) =
  let g = qftRec (S k)
      h = (IdGate {n = 1}) # (qft k)
  in h . g
```

Fig. 5. Qimaera code for QFT (file: `QFT.idr`).

5.1 Representation of Quantum Effects in Qimaera

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 [32,29,22], which is in turn used to formally describe the operational semantics of quantum type systems.

Qubits in Qimaera. Because of the possibility of quantum entanglement, we cannot describe the state of an individual qubit which is part of a larger composite 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 (outside of the library file). Instead, our libraries provide functions (Figure 7) that ensure that a term of type `Qubit` is created with a fresh (i.e., unique) natural number that serves as its identifier within a monadic environment. This is handled by our functions through careful manipulation of the available data within the monadic environment. In fact, these functions are the expected way

for our users to access or manipulate qubits and, moreover, our users cannot access the unique identifiers (unless they modify our libraries). This allows us to formulate a representation where values of type `Qubit` unambiguously refer to the relevant parts of larger composite systems. Therefore, a value 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` do not carry any sort of linear-algebraic information.

Probabilistic Effects. Quantum measurements induce probabilistic computational effects which are inherited by the classical side of the computation in hybrid classical-quantum algorithms. Furthermore, in our intended computational scenario, the classical computer (on which Idris is running) sends instructions to, and receives data from, the quantum device. In order to correctly model all of this, it is clear that we have to use the IO monad in order to encapsulate these effects. However, when representing quantum program dynamics, we also need to *enforce linearity*, but all the functions provided by the IO monad (e.g., `pure` which introduces pure values to monadic types) 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 brevity, we define `R` to be our linear IO monad:

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

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

Quantum State Transformer. Quantum computation is *effectful*, and moreover, quantum information *cannot* be observed by the classical computer (on which Idris is running): it only receives classical information through communication with the quantum device. Because of this, we adopt a more abstract view on the hybrid classical-quantum computational process. In order to do this, we define an (abstract) *quantum state transformer* by combining several different concepts: *indexed state monads* [4][7], linearity and IO (and thus also probabilistic) effects. Our representation of these ideas in Qimaera is shown in Figure 6, where we omit the function definitions for brevity.

The type `QStateT` is parameterised by a choice of three (arbitrary) types, so it is fairly abstract. Soon, we will see that it is very useful for our purposes. The intended interpretation of this type is the following: any value of type

```
QStateT initialType finalType returnType
```

represents a stateful (quantum) computation starting from a (quantum) state of type `initialType` and ending in a (quantum) state of type `finalType` which

[7] See [33] for a Haskell implementation of this idea.

```
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
```

Fig. 6. Quantum state transformer (file: `QStateT.idr`).

produces a user-accessible result of type `returnType` during the computation. For example, a value of type

```
QStateT (LPair Qubit Qubit) Qubit Bool
```

should be understood as a quantum process that transforms a two-qubit state into a single-qubit state and returns a single (classical) value of type `Bool` to the user. The functions presented in Figure 6 allow us to adopt a *monadic programming discipline* when working with `QStateT` and we do so henceforth. We remark that `QStateT` makes use of the monad `R` which encapsulates the IO (and probabilistic) effects and that linearity is enforced when working with `QStateT`.

Effectful Quantum Programming. The `QStateT` monad can be used to define a suitable *abstract interface* for quantum programming. In Figure 7, we present an excerpt of the `QuantumOp` interface which allows us to write quantum programs and execute them in a type-safe way. All of the hybrid quantum-classical algorithms we present are implemented using this interface.

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 file `QuantumOp.idr` also provides functions `applyH`, `applyP` and `applyCNOT` which can be seen as special cases of `applyUnitary`. However, these three functions do not depend on the `Unitary` type.

The `measure` function is used to measure `i` qubits identified by the `LVect` argument and it returns a value of type `Vect i Bool` that represents the result of the measurement. After this, the `i` measured qubits are not reused, as one can see from the provided type information.

```
interface QuantumOp (0 t : Nat -> Type) where
  newQubits : (p : Nat) -> QStateT (t n) (t (n+p)) (LVect p Qubit)

  newQubit : QStateT (t n) (t (S n)) Qubit

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

  applyH : {n : Nat} -> (1 _ : Qubit) -> QStateT (t n) (t n) Qubit

  applyP : {n : Nat} -> Double -> (1 _ : Qubit) ->
                        QStateT (t n) (t n) Qubit

  applyCNOT : {n : Nat} -> (1 _ : Qubit) -> (1 _ : Qubit) ->
    QStateT (t n) (t n) (LPair Qubit Qubit)

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

  measureQubit : {n : Nat} -> (1 _ : Qubit) ->
                              QStateT (t (S n)) (t n) Bool

  measureAll : {n : Nat} -> (1 _ : LVect n Qubit) ->
    QStateT (t n) (t 0) (Vect n Bool)

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

Fig. 7. The `QuantumOp` interface (file: `QuantumOp.idr`).

Finally, the function `run` is used to *execute* quantum algorithms on the quantum device and obtain the classical information returned from it. Notice that `run` can be used to execute effectful quantum processes which start from the trivial quantum state (on zero qubits) and which terminate in the same trivial quantum state, but which also produce some number of classical bits as a user-accessible return result. This may be used to run 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 producing n bits of classical information. This quantum algorithm may then be represented as a value of type `QStateT (t 0) (t 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 modify the quantum state on the quantum device and may cause IO effects, because of the need to communicate with the quantum device. This is indeed reflected by our interface. Observe, that our interface is defined using the `QStateT` monad transformer which does incorporate IO effects (via the `R` monad we discussed previously).

Example 7. A fair coin toss may be implemented using quantum resources. The process is simple: (1) prepare the state $|0\rangle$; (2) apply the H gate to it; (3) measure the qubit and return this as output. We implement this as follows:

```
coin : QuantumOp t => IO Bool
coin = do
  [b] <- run (do
           q <- newQubit {t = t}
           q <- applyH q
           r <- measure [q]
           pure r
         )
  pure b
```

The top-level **do** block simply realises monadic sequencing for the standard IO monad. The **do** block within the `run` environment is more interesting and crucial for our development. It performs monadic sequencing for the `QStateT` monad and it represents the simple three-step algorithm we just described. The call to the `run` function executes this algorithm and users obtain the produced classical information by storing it in the variable `b` of type `Bool`. We emphasise that linearity is *enforced* within the `run` environment and this is what brings safety properties in our approach, e.g., all of the following scenarios are statically detected and rejected by Idris: passing the qubit `q` to a non-linear function, copying the qubit `q`, forgetting to measure the qubit `q`. For example, if in the above code we replace the last two statements in the `run` environment with "`pure True`", then Idris statically detects this error.

The function `coin` from Example 7 is implemented using our *abstract* interface. This means we can use this function in any *concrete* implementation of the `QuantumOp` interface. Since the authors do not have any quantum hardware, we provide one concrete implementation of this interface, called `SimulatedOp`, which performs linear-algebraic simulation of all the required operations. For example, if we wish to use the `coin` function, then the code:

```
testCoin : IO Bool
testCoin = coin {t = SimulatedOp}
```

defines a new function, called `testCoin`, which does the same as `coin`, but it specifically instructs Idris to use linear-algebraic simulation. We emphasise that all of our quantum algorithms are written using our abstract interface, so there is no need to reimplement them for any additional concrete implementations of the interface.

5.2 Example: Repeat-Until-Success Algorithm

Repeat-until-success (RUS) [27] is an algorithm for implementing quantum unitary operators by using *quantum measurements* and *general unbounded recursion*. The main advantage in using RUS over traditional deterministic techniques

```
RUS : QuantumOp t => (1 _ : Qubit) ->
      (u' : Unitary 2) -> (e : Unitary 1) ->
      QStateT (t 1) (t 1) Qubit
RUS q u' e = do
  q' <- newQubit
  [q',q] <- applyUnitary [q',q] u'
  b <- measureQubit q'
  if b then do
         [q] <- applyUnitary [q] (adjoint e)
         RUS q u' e
       else pure q

example_u' : Unitary 2
example_u' = H 0 $ T 0 $ CNOT 0 1 $ H 0 $ CNOT 0 1 $ T 0 $
             H 0 IdGate

runRUS : QuantumOp t => IO Bool
runRUS = do
  [b] <- run (do
              q <- newQubit {t = t}
              q <- RUS q example_u' IdGate
              measure [q]
         )
  pure b

testRUS : IO Bool
testRUS = runRUS {t = SimulatedOp}
```

Fig. 8. Repeat-until-success algorithm (file: `RUS.idr`).

that synthesise unitary operators, is that with RUS the expected number of T gates (which are expensive in terms of error correction[8]) can be reduced.

In the simplest case, we wish to realise a fixed single-qubit unitary operator $U : \mathbb{C}^2 \to \mathbb{C}^2$. The RUS algorithm is as follows. Given an input qubit $|\psi\rangle$, then: (1) prepare a new qubit in state $|0\rangle$; (2) apply a two-qubit unitary operator U' (chosen in advance depending on U); (3) measure the first qubit; (4) if the measurement outcome is 0 (which occurs with probability $p > 0$), then the output state is $U|\psi\rangle$, as required, and the algorithm terminates; otherwise the current state is $E|\psi\rangle$, where E is some other unitary operator (chosen in advance depending on U), so we apply $E^\dagger$ to this state and we go back to step (1). The unitary operators U' and E are chosen in advance, depending on U, before the algorithm starts so that the above conditions are satisfied. Note that synthesising U' and E is not part of the algorithm and we do not discuss this here.

Assuming that appropriate U' and E are chosen, this process always terminates in state $U|\psi\rangle$ (provided $p > 0$) so RUS indeed implements the unitary operator U. Note that this is an *algorithmic* realisation of U, not an algebraic one, and so we cannot write a program of type `Unitary` that achieves this. Instead, we represent this as a quantum program in Figure 8. There, `RUS q u'`

[8] We do not automatically implement error correction, so it has to be handled either by the developer or provided by the quantum device on the remote end.

`e` is the quantum state transformer which implements the RUS algorithm as above. The function `runRUS` simply executes the RUS algorithm on a qubit in state $|0\rangle$, with the unitary operator chosen from [27, Figure 8], then measures the qubit and returns the outcome. Both of these functions are written using our abstract interface. The function `testRUS` is the same as `runRUS`, but it also instructs Idris to use linear-algebraic simulation for the execution. Note that, in our implementation, we have taken a specific instance of RUS by choosing U' to be the unitary operator described by `example_u'` as discussed in [27, Figure 8].

Remark 3. The `run(-)` environment enforces linearity, so if we wish to use the `RUS` function within it, then the qubit argument must be linear in `RUS`.

6 Variational Quantum Programming

In the previous section we saw that Qimaera is suitable for writing recursive and effectful quantum programs that make use of quantum measurements. Moreover, Idris 2 is an excellent programming language with an advanced type system and first-class support for classical programming features. In order to demonstrate that Qimaera is suitable for hybrid classical-quantum programming, we also have to show that both classical and quantum programming features may be elegantly combined. This is the purpose of this section and we achieve this by implementing the two most prominent variational quantum algorithms: the Quantum Approximate Optimization Algorithm (QAOA) [13] and the Variational Quantum Eigensolver (VQE) [30]. In this paper we only describe QAOA. See the full paper [12] for more information on the implementation of VQE.

The objective of QAOA is to try to find the minimum (or maximum) eigenvalue of a Hamiltonian. A Hamiltonian is a Hermitian (i.e., self-adjoint) matrix $\mathcal{H}$ (we use a calligraphic font to differentiate it from H, the Hadamard matrix). Its minimum eigenvalue is the minimum (real) value λ such that $\mathcal{H}|\psi\rangle = \lambda|\psi\rangle$ for some nonzero vector $|\psi\rangle$. As $\mathcal{H}$ is unitarily diagonalizable, this is equivalent to the minimum of $\langle\psi|\mathcal{H}|\psi\rangle$ for all vectors $|\psi\rangle$ of norm 1, where $\langle\psi| \stackrel{\text{def}}{=} |\psi\rangle^\dagger$.

QAOA starts with some assumption on what the vector $|\psi\rangle$ looks like and usually $|\psi\rangle$ is prepared by a quantum circuit that depends on some real parameters $\alpha_1, \ldots, \alpha_p$. By measuring this state $|\psi\rangle$, one obtains some information on the value of $\langle\psi|\mathcal{H}|\psi\rangle$. This information can then be fed to a *classical* optimizer to change the value of the parameters $\alpha_1, \ldots, \alpha_p$ for subsequent execution.

This classical-quantum back and forth is repeated until some satisfactory termination condition has been satisfied. For example, we may simply repeat this process k times, where $k \in \mathbb{N}$ is some constant, but more sophisticated termination conditions are also possible. However, there is no guarantee that we will find the minimum eigenvalue.

Implementation of QAOA. QAOA is a variational algorithm [13] that approximately solves optimization problems. Let $f : \{0,1\}^n \to \mathbb{R}$ be a function for which we want to find its minimum. We see f as a diagonal Hamiltonian over n

qubits defined by $\mathcal{H}|x\rangle = f(x)|x\rangle$ for all $x \in \{0,1\}^n$. We are therefore searching for the minimum eigenvalue of this Hamiltonian.

In this case, the state $|\psi\rangle$ that minimises the Hamiltonian $\mathcal{H}$ is often assumed to be of the form: $|\psi\rangle = (HP(\beta_p)H)^{\otimes n} e^{\gamma_p \mathcal{H}} \cdots (HP(\beta_1)H)^{\otimes n} e^{\gamma_1 \mathcal{H}} H^{\otimes n} |0\rangle$. The depth parameter $p \in \mathbb{N}$ is usually fixed to be small, and we have a guarantee that the results of our algorithm become better when p becomes larger. To be able to produce a circuit which computes $|\psi\rangle$, the Hamiltonian $\mathcal{H}$ may be assumed to have a special form so that we can make a circuit for $e^{\gamma \mathcal{H}}$. A well-known and important example is to compute the maximum cut of an undirected graph, i.e., to solve the MAXCUT problem.

Our implementation for QAOA on the MAXCUT problem is presented in the file `QAOA.idr` and an excerpt is shown in Figure 9. The problem depends on the graph G for which we want the maximum cut, a depth parameter p, and some real parameters β_i, γ_i.

In our implementation, we have a function `QAOA_Unitary`, that takes these parameters as input and produces a unitary circuit that may be used to prepare the state $|\psi\rangle$ when applied to the initial state $|0\rangle^{\otimes n}$. We then measure this state $|\psi\rangle$ and present the result (a cut of the graph in the obvious binary encoding) to an optimiser. Our optimiser is implemented by the function `classicalOptimisation` that uses all observable information from all previous runs (which amounts to the values of the parameters β_i, γ_i and the value of the cuts that have been previously obtained through quantum measurements) to compute the subsequent rotation parameters β_i, γ_i that we will use for the next iteration. The type of this function indicates that it uses the IO monad: this is because we wish to allow the function to use probabilistic optimisation algorithms or even external tools. One of the simplest implementations of this function chooses the rotation parameters at random.

The interplay between the classical and the quantum part is presented in Figure 9. The function `QAOA` takes as input a natural number k representing how many times the whole routine will be done, the depth p of the circuit, and the graph G on which to compute the cut. Notice that the call to the quantum device is isolated inside the `run` function.

7 Related Work

In this section we compare Qimaera with other existing quantum programming languages that are implemented in software. We omit comparisons with quantum type systems that do not have a software implementation. We provide a feature comparison with some quantum programming languages in Table 2 and we now clarify the meaning of some of the selected features.

By *Type Safety* we mean that the language can statically detect (and reject) erroneous programs which duplicate quantum data. *General Recursion* is the ability to express recursive (possibly non-terminating) programs and almost-surely-terminating programs, such as RUS (see §5.2). *Measurements* is the ability to use the outcomes of quantum measurements in the control flow of programs.

```
QAOA_Unitary : {n : Nat} -> (betas : Vect p Double)
                         -> (gammas : Vect p Double)
                         -> (graph: Graph n) -> Unitary n

classicalOptimisation : {p : Nat}
                     -> (graph : Graph n)
                     -> (previous_info : Vect k (Vect p Double,
                        Vect p Double, Cut n))
                     -> IO (Vect p Double, Vect p Double)

QAOA' : QuantumOp t =>
        {n : Nat} ->
        (k : Nat) -> (p : Nat) -> (graph : Graph n) ->
        IO (Vect k (Vect p Double, Vect p Double, Cut n))
QAOA' 0 p graph = pure []
QAOA' (S k) p graph = do
  previous_info <- QAOA' {t} k p graph
  (betas, gammas) <- classicalOptimisation graph previous_info
  let circuit = QAOA_Unitary betas gammas graph
  cut <- run (do
              qs <- newQubits {t} n
              qs <- applyUnitary qs circuit
              measureAll qs
              )
  pure $ (betas, gammas, cut) :: previous_info

QAOA : QuantumOp t => {n : Nat} -> (k : Nat) -> (p : Nat) ->
                                    Graph n -> IO (Cut n)
QAOA k p graph = do
  res <- QAOA' {t} k p graph
  let cuts = map (\(_, _, cut) => cut) res
  let (cut,size) = bestCut graph cuts
  pure cut
```

Fig. 9. Qimaera implementation (excerpt) for the QAOA algorithm solving the MAX-CUT problem.

Promotion of Measurements is the ability to integrate the outcomes of quantum measurements as a *native* classical type (e.g., `Bool`): this essentially allows us to switch from a quantum mode of operation into a classical one and allows us to use both quantum and classical programming paradigms; it may be roughly understood as corresponding to the *promotion* rule of linear logic [16]. For *Higher-order Functions* we distinguish between purely classical ones and mixed classical-quantum (in the second column); some languages support both, but treat the quantum ones non-linearly which may cause loss of type safety. Finally, by *Effects* we mean the ability to incorporate probabilistic computational effects (which are an essential part of the dynamics of hybrid classical-quantum programs) and also IO (input/output) effects into our programming workflow.

The QWIRE language [28,31] and the SQIR language [20,19] are quantum circuit languages that are embedded in the Coq proof assistant [11]. Both of these languages have access to dependent types, courtesy of Coq. The focus of these languages is mostly on verification, whereas in Qimaera we focus on *programming* and Idris 2 has better support for classical, quantum and effectful programming features compared to Coq. Both QWIRE and SQIR represent quantum primitives through the use of low-level specification languages that are embedded in Coq: both of these specification languages lack the ability to express quantum algorithms that require general recursion and both of them lack the ability to express quantum higher-order functions. Because of the former reason, the RUS algorithm from §5.2 cannot be expressed in QWIRE or SQIR.

Silq [9] is a standalone quantum programming language which also is type-safe and whose main notable feature is automatic uncomputation of temporary values. We currently partially support this feature, because we have clearly identified and separated the reversible fragment of quantum computation (see the `Unitary` type) and we can synthesise the required adjoints by calling the `adjoint` function. Compared to Silq, the main advantage of Qimaera is that Idris has better support for classical programming features and so we believe that Qimaera is a better choice for hybrid classical-quantum programming. In addition, Silq does not support general recursion, so it cannot express quantum algorithms that rely on this (e.g., RUS §5.2).

Language	Type Safety	General Recursion	Dependent Types	Measurements	Promotion of Measurements	Higher-order Functions		Effects
						Classical	Quantum	
Quipper	×	✓	×	✓	✓	✓	(non-linear)	✓
Proto-Quipper-D	✓	✓	✓	×	×	✓	✓	×
Proto-Quipper-Dyn	✓	✓	×	✓	✓	✓	✓	✓
QWIRE	✓	×	✓	✓	×	✓	×	×
SQIR	✓	×	✓	✓	×	✓	×	×
Silq	✓	×	(limitted)	✓	✓	✓	✓	×
Qiskit	×	✓	×	✓	✓	✓	(non-linear)	✓
Q#	×	✓	×	✓	✓	✓	(non-linear)	✓
Cirq	×	✓	×	✓	✓	✓	(non-linear)	✓
Qimaera	✓	✓	✓	✓	✓	✓	✓	✓

Table 2. Feature comparison between Qimaera and other languages.

Quipper [18] and the Quantum IO monad (QIO) [3] are two domain specific languages (DSLs) embedded in Haskell. Neither of them are type safe because they do not utilise linearity and they cannot statically detect quantum programs that are physically inadmissible. However, thanks to the language similarities between Haskell and Idris, the programming style in these languages is somewhat similar to ours (e.g., all three use monads). In our view, both of these papers have been influential for the design of functional quantum programming languages.

Another recent language includes Proto-Quipper-D [14] which is a type-safe circuit description language. This language is based on a novel type system which shows how linearity and dependent types can be combined. A fundamental difference between Proto-Quipper-D and Qimaera is that linearity is the default mode of operation in Proto-Quipper-D, whereas in Qimaera the default mode is non-linear. The focus in Proto-Quipper-D is on *circuit* description and generation and the language currently lacks effectful quantum measurements and probabilistic effects, so it cannot be used for variational quantum programming at present. Another related language is Proto-Quipper-Dyn [15]. It is similar to Proto-Quipper-D, but it lacks dependent types (which Qimaera has). On the other hand, it can handle quantum measurements and has *dynamic lifting*, i.e., the ability to parametrise quantum circuits based on information observed from quantum measurements. Note that Qimaera also has dynamic lifting.

Other languages, include Google's Cirq [17] (a set of python libraries), IBM's Qiskit [2] (a set of python libraries) and Microsoft's Q♯ [35] (standalone). These languages offer a wide-range of quantum functions and features, however, none of them are type-safe. Qimaera does not have this problem and this is indeed its main advantage over them, together with dependent types.

8 Future Work

For future work, it would be interesting to consider methods that would allow us to reduce some of the proof obligations that are imposed by the `Unitary` data type. Going beyond Idris and our library, another natural direction is to consider whether programming languages that support substructural approaches other than linearity (e.g., uniqueness types, ownership) can be used to achieve type-safe quantum programming. It would also be interesting to consider the relevance of arrows [21,5] in quantum programming. Furthermore, implementing and testing our abstract interface on an actual hybrid quantum-classical hardware environment would most likely bring additional challenges.

Acknowledgements. We thank Robert Rand for discussions about this paper. We also thank the anonymous referees for their feedback which lead to multiple improvements of this paper. EJ is supported by the PEPR integrated project EPiQ and the European Project NEASQC (Grant Agreement 951821). VZ acknowledges support from the HQI-R&D ANR-22-PNCQ-0002 project. Most of the work was done in LORIA/Inria Nancy during an Inria internship of the first author who was a student at École polytechnique. The first and last authors have changed affiliations since then.

References

1. Qimaera github repository. https://github.com/zamdzhiev/Qimaera, accessed: 30.01.2023
2. Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., Cabrera-Hernández, F.J., Carballo-Franquis, J., Chen, A., Chen, C.F., Chow, J.M., Córcoles-Gonzales, A.D., Cross, A.J., Cross, A., Cruz-Benito, J., Culver, C., González, S.D.L.P., Torre, E.D.L., Ding, D., Dumitrescu, E., Duran, I., Eendebak, P., Everitt, M., Sertage, I.F., Frisch, A., Fuhrer, A., Gambetta, J., Gago, B.G., Gomez-Mosquera, J., Greenberg, D., Hamamura, I., Havlicek, V., Hellmers, J., Łukasz Herok, Horii, H., Hu, S., Imamichi, T., Itoko, T., Javadi-Abhari, A., Kanazawa, N., Karazeev, A., Krsulich, K., Liu, P., Luh, Y., Maeng, Y., Marques, M., Martín-Fernández, F.J., McClure, D.T., McKay, D., Meesala, S., Mezzacapo, A., Moll, N., Rodríguez, D.M., Nannicini, G., Nation, P., Ollitrault, P., O'Riordan, L.J., Paik, H., Pérez, J., Phan, A., Pistoia, M., Prutyanov, V., Reuter, M., Rice, J., Davila, A.R., Rudy, R.H.P., Ryu, M., Sathaye, N., Schnabel, C., Schoute, E., Setia, K., Shi, Y., Silva, A., Siraichi, Y., Sivarajah, S., Smolin, J.A., Soeken, M., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Trabing, K., Treinish, M., Turner, W., Vogt-Lee, D., Vuillot, C., Wildstrom, J.A., Wilson, J., Winston, E., Wood, C., Wood, S., Wörner, S., Akhalwaya, I.Y., Zoufal, C.: Qiskit: An open-source framework for quantum computing (Jan 2019). https://doi.org/10.5281/zenodo.2562111, https://doi.org/10.5281/zenodo.2562111
3. Altenkirch, T., Green, A.S.: The quantum IO monad. Semantic Techniques in Quantum Computation pp. 173–205 (2010)
4. Atkey, R.: Parameterised notions of computation. J. Funct. Program. **19**(3-4), 335–376 (2009). https://doi.org/10.1017/S095679680900728X, https://doi.org/10.1017/S095679680900728X
5. Atkey, R.: What is a categorical model of arrows? Electron. Notes Theor. Comput. Sci. **229**(5), 19–37 (2011). https://doi.org/10.1016/j.entcs.2011.02.014, https://doi.org/10.1016/j.entcs.2011.02.014
6. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189, https://doi.org/10.1145/3209108.3209189
7. Benton, P.N., Wadler, P.: Linear logic, monads and the lambda calculus. In: LICS 1996 (1996)
8. Benton, P.: A mixed linear and non-linear logic: Proofs, terms and models. In: Computer Science Logic: 8th Workshop, CSL '94, Selected Papaers (1995). https://doi.org/10.1007/BFb0022251, http://dx.doi.org/10.1007/BFb0022251

9. Bichsel, B., Baader, M., Gehr, T., Vechev, M.T.: Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 286–300. ACM (2020). https://doi.org/10.1145/3385412.3386007, https://doi.org/10.1145/3385412.3386007
10. Brady, E.C.: Idris 2: Quantitative type theory in practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, https://doi.org/10.4230/LIPIcs.ECOOP.2021.9
11. Coq Development Team: The Coq proof assistant reference manual. https://coq.inria.fr/distrib/current/refman/ (2021), accessed: 19.11.2021
12. Dandy, L.J., Jeandel, E., Zamdzhiev, V.: Type-safe quantum programming in Idris. https://doi.org/10.48550/ARXIV.2111.10867, https://arxiv.org/abs/2111.10867
13. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm (2014)
14. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In: Lanese, I., Rawski, M. (eds.) Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12227, pp. 153–168. Springer (2020). https://doi.org/10.1007/978-3-030-52482-1_9, https://doi.org/10.1007/978-3-030-52482-1_9
15. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-Quipper with dynamic lifting. CoRR **abs/2204.13041** (2022). https://doi.org/10.48550/arXiv.2204.13041, https://doi.org/10.48550/arXiv.2204.13041
16. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**, 1 – 101 (1987)
17. Google AI Quantum Team: Cirq. https://quantumai.google/cirq (2021), accessed: 13.08.2021
18. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: PLDI. pp. 333–342. ACM (2013)
19. Hietala, K., Rand, R., Hung, S., Li, L., Hicks, M.: Proving quantum programs correct. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.21, https://doi.org/10.4230/LIPIcs.ITP.2021.21
20. Hietala, K., Rand, R., Hung, S., Wu, X., Hicks, M.: A verified optimizer for quantum circuits. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434318, https://doi.org/10.1145/3434318
21. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4, https://doi.org/10.1016/S0167-6423(99)00023-4
22. Jia, X., Kornell, A., Lindenhovius, B., Mislove, M.W., Zamdzhiev, V.: Semantics for variational quantum programming. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498687, https://doi.org/10.1145/3498687
23. Mac Lane, S.: Categories for the Working Mathematician (2nd ed.). Springer (1998)

24. McBride, C.: I got plenty o' nuttin'. In: Lindley, S., McBride, C., Trinder, P.W., Sannella, D. (eds.) 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, vol. 9600, pp. 207–233. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_12, https://doi.org/10.1007/978-3-319-30936-1_12
25. McClean, J.R., Romero, J., Babbush, R., Aspuru-Guzik, A.: The theory of variational hybrid quantum-classical algorithms. New Journal of Physics **18**(2), 023023 (2016)
26. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010). https://doi.org/10.1017/CBO9780511976667
27. Paetznick, A., Svore, K.M.: Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. Quantum Info. Comput. **14**(15–16), 1277–1301 (Nov 2014)
28. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: a core language for quantum circuits. In: POPL. pp. 846–858. ACM (2017)
29. Péchoux, R., Perdrix, S., Rennela, M., Zamdzhiev, V.: Quantum programming with inductive datatypes: Causality and affine type theory. In: Foundations of Software Science and Computation Structures, FOSSACS 2020. Lecture Notes in Computer Science, vol. 12077, pp. 562–581. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_29
30. Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.H., Zhou, X.Q., Love, P.J., Aspuru-Guzik, A., O'brien, J.L.: A variational eigenvalue solver on a photonic quantum processor. Nature communications **5**(1), 1–7 (2014)
31. Rand, R., Paykin, J., Lee, D., Zdancewic, S.: ReQWIRE: Reasoning about reversible quantum circuits. In: Selinger, P., Chiribella, G. (eds.) Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018. EPTCS, vol. 287, pp. 299–312 (2018). https://doi.org/10.4204/EPTCS.287.17, https://doi.org/10.4204/EPTCS.287.17
32. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. Mathematical Structures in Computer Science **16**(3), 527–552 (2006)
33. Seo, K.Y.: Indexed state monad blog post. https://kseo.github.io/posts/2017-01-12-indexed-monads.html (2017), accessed: 13.08.2021
34. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Review **41**(2), 303–332 (1999). https://doi.org/10.1137/S0036144598347011
35. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level dsl. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3183895.3183901, https://doi.org/10.1145/3183895.3183901
36. Wootters, W.K., Zurek, W.H.: A single quantum cannot be cloned. Nature **299**(5886), 802–803 (1982)