Modeling Quantum Computing Constraints: No-Cloning Theorem and Monadic Expressiveness with Type-Level Programming

Flávio Borin Júnior
Universidade Federal de Santa Maria
Santa Maria, RS, Brazil
fbjunior@inf.ufsm.br

Juliana Kaizer Vizzotto
Universidade Federal de Santa Maria
Santa Maria, Brazil
juvizzotto@inf.ufsm.br

ABSTRACT

This paper presents a type-safe model of quantum computing, building upon the model introduced by Sabry in Haskell. By applying type-level programming techniques, we extend the model to constrain invalid operations and states at compile time. Our implementation enforces the no-cloning theorem at the type level via memory access patterns, while maintaining expressiveness via monadic composition of quantum operations. The implementation serves both as an introduction to quantum computing fundamentals and as a case study in enforcing physical constraints through a functional programming type system.

KEYWORDS

Type-Level Programming, Quantum Computing, Quantum Circuit Languages

1 Introduction

Quantum computing represents a paradigm shift from classical computing by applying the principles of quantum mechanics. It has been proven that it can solve several computational problems with asymptotic speedup. The fundamental unit of quantum information, the qubit, exhibits unique physical properties such as superposition, entanglement, and interference [8]. Quantum algorithms are developed to exploit these characteristics.

Since Shor's [11] work on a quantum algorithm for integer factorization in polynomial time, significant progress has been made in the field of quantum computing. Much of the existing literature focuses on the physical aspects of quantum computing or low-level quantum programming. Considering another point of view, this work is about high-level frameworks and abstractions for quantum programming. In this context, Sabry [10] proposes a model of quantum computing in Haskell, i.e., functional programming abstractions for programming quantum algorithms. Vizzotto [15] extends this abstraction even further using the concept of *arrows* [6]. Both works provide a layer of abstraction that enables quantum programming and circuit reasoning inside the functional programming language Haskell.

Sabry's didactic model of quantum computing, while a valuable contribution to functional quantum emulation, faces scalability issues when dealing with more than a couple of qubits. The model lacks a general way to handle all possible quantum memory sizes. More specifically, selecting the desired target qubits to apply a quantum operation on is verbose and requires a specific function definition in each case.

An important observation is the extensive use of type-level programming in functional programming languages. This machinery allows properties and constraints to be verified at compile time, enforcing correctness by design. Concepts such as dependent types and singletons are well established in the functional programming community, and significant research has been done in this area [5, 7]. In the context of quantum computing, these concepts prove to be particularly useful for modeling invariants [3]. The compile-time checks provided by such languages allow us to enforce quantum computing principles, such as the no-cloning theorem, which states that it is impossible to duplicate an arbitrary quantum state.

In many quantum programming languages, the no-cloning theorem is assured through linearity, strongly inspired by the Linear Lambda Calculus [13, 14]. Languages such as QML [1], Qimaera [3], and QWIRE [9] incorporate linear typing to ensure quantum data is not duplicated. Linearity assures that every function must use each parameter exactly once, thereby not allowing state copying.

This article aims to rework Sabry's model to generalize qubit access in quantum memory while enforcing the correctness of quantum operations and states through compile-time verification. Instead of relying on an explicit linear type system, our approach encodes quantum constraints purely on Haskell's type system. More specifically, we statically detect invalid memory access that violates the no-cloning theorem via qubit selection patterns. Despite these rigorous guarantees, expressiveness is achieved through monadic composition of quantum operations.

This article is structured as follows: in Section 2, we present Sabry's model for qubits and quantum operations, which we adapt to our purposes as a type-unsafe system. In Section 3, we aim to create an abstraction layer that uses type systems to provide safety and generalize selection of qubits. Section 4 is dedicated to the implementation of a monad that enhances

the expressiveness of quantum algorithm definitions and formalizes the typing rules for quantum operations. Finally, in Section 5, we present the results and conclusions of the work, suggesting potential future development to improve the model. The full implementation is available on GitHub[1].

2 Quantum Computing and Sabry's Model

A qubit state, the unit of information in quantum computing, can be seen as a linear combination of the basis states. Mathematically, a state is represented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $|0\rangle$ and $|1\rangle$ are the basis states and α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$.

Dirac notation, as it's called, provides a mathematical framework for reasoning about multiple qubit states. For example, the state $(|11\rangle + |00\rangle)/\sqrt{2}$ implies a 50% chance of a measurement of the system resulting in an outcome of either $|11\rangle$ or $|00\rangle$. Once measured, the quantum system collapses to the value of the outcome. Since the qubits in this state are entangled, it's not possible to measure one of them without affecting the other.

Using this idea, a quantum memory state (QV - Quantum Value) can be modeled as a mapping between basis states and complex amplitudes. In Haskell, this can be implemented using a Map from lists of binary values to complex scalars. We also present mkQV, for now, the standard function to build a quantum state.

```
type PA = Complex Double

data QV a = QV {
    qvSize :: Int,
    qvMap :: Map [a] PA
    }

mkQV :: Basis a ⇒ [([a], PA)] → QV a
mkQV pairs = QV (length $ fst $ head pairs)
                (fromList pairs)
```

For convenience, Sabry's model uses a typeclass to define which types can form a basis of the complex vector space [10]. This approach introduces polymorphism into the implementation, enabling the simulation of systems like qutrits, for example. We adapt this typeclass so that its function basis depends directly on the number of qubits. Additionally, we require that a basis type form a total order, as the implementation of Map relies on this property. This is equivalent to defining an ordering for the basis states, such as $|0\rangle \prec |1\rangle$, $|00\rangle \prec |01\rangle \prec |10\rangle \prec |11\rangle$, and so on.

```
-- 0 and 1
data Bit = O | I deriving (Ord, Read, Num)

class Ord a ⇒ Basis a where
  basis' :: [a]

instance Basis Bit where
  basis' = [O, 1]
```

[1] https://github.com/Fleivio/Well-Typed-Qubits

The basis function generates the complete basis set for an n-qubit system by performing a Cartesian product between the fundamental basis states.

```
basis :: Basis a ⇒ Int → [[a]]
basis 0 = [[]]
basis n = [b : bs | b ∈ basis', bs ∈ basis (n-1)]
```

A quantum gate is mathematically defined as a matrix with complex entries. We model the quantum operation (OP) as a mapping between pairs of basis states and complex amplitudes. A pair ([a], [a])) is an index to the matrix's rows and columns, and the complex quantity entries represent the probability amplitude of transitioning between the indexing states. The structure also stores the number of qubits associated with the operation. The function appOP implements the matrix-vector multiplication required for a quantum gate application on a QV.

```
data OP a = OP {
    opSize :: Int,
    opMap :: Map ([a], [a]) PA
  }

appOP :: Basis a ⇒ OP a → QV a → QV a
appOP qop qv
    = mkQV [(b, prob b) | b ∈ basis (opSize qop)]
  where
  prob b = sum [qop `getOpProb` (a, b) × qv `getProb` a
                | a ∈ basis (opSize qop)]
```

The controlled-not gate is a two-qubit operation that applies a bit flip to the target qubit if and only if the control qubit is $|1\rangle$. Its behavior can be summarized using the classical exclusive or (XOR) operation: C-NOT $|xy\rangle = |x\rangle \otimes |x \oplus y\rangle$. The C-Not gate (controlled-not) in our current implementation, can be defined as follows:

```
cnot :: OP Bit
cnot = mkOP [(([0,0],[0,0]),1),
             (([0,1],[0,1]),1),
             (([1,0],[1,1]),1),
             (([1,1],[1,0]),1)]
```

Quantum systems, as already mentioned, exhibit nonlocal effects. This characteristic allows operations that are seemingly local to affect the entire memory state. To abstract this global effect, we introduce a new type QR (Quantum Reference) that maintains a pointer to the memory and can be shared between different parts of the program. The function observeQR performs a projective measurement on a quantum reference at a given index, collapsing the entire state according to the outcome.

```
type QR a = (IORef (QV a))

observeQR :: Basis a ⇒ QR a → Int → IO a
observeQR ref index = ...
```

This datatype will be used to ensure that quantum measurements applied (seemingly) to a few qubits have an effect on the entire system.

3 Type-Level Safety and Generalization

Although the presented implementation is strictly correct, it has limitations.

First, without a type-level checking, there is nothing preventing one from defining invalid quantum states. On quantum computing, mixing states from different bases has no meaning. A quantum state is composed of different basis states in superposition. Since all these superposition states refer to the same physical system, they must agree on the qubit count.

Consider the following examples:

`mkQV [([1,1], 1/√2), ([0,0], 1/√2)]` $\equiv \frac{1}{\sqrt{2}}|11\rangle + \frac{1}{\sqrt{2}}|00\rangle$

This corresponds to a properly entangled state of two qubits.

`mkQV [([1], 1/√2), ([0,0], 1/√2)]` $\stackrel{?}{\equiv} \frac{1}{\sqrt{2}}|1\rangle + \frac{1}{\sqrt{2}}|00\rangle$

In contrast, this is a invalid mixing of a 1-qubit state and a 2-qubit state. Currently, we can create such a ill-formed state since the linked list length is not known at compile time.

Second, we need to define actual virtual values that make use of the `QR` type introduced earlier. A virtual value acts as a subset of the quantum memory and can translate quantum operations performed on a few qubits into their corresponding effects on the entire system. Sabry [10] achieved this by using adaptors — functions that decouple and join values using tuples. However, this approach becomes impractical as the memory size grows, since a new adaptor must be defined for each tuple size.

In this section, we present a mechanism that solves both problems by using lists instead of tuples, reworking the qubit access strategy, and implementing compile-time checks to enforce quantum computing invariants, as a result preventing the generation of invalid states. The typing rules regarding the quantum gates and measurements will be addressed in section 4.

3.1 Type-Level Programming

Type-level programming is a technique where logic and data are encoded on types to enforce compile-time guarantees. Haskell's supports this paradigm though extensions such as TypeFamilies, GADTs and DataKinds. In particular, DataKinds allows values to be lifted to the type level. This means that any ordinary term-level definition can be used on type signatures.

For example, consider a datatype `Natural`, representing natural numbers in Peano's form:

```
data Natural = Zero | Succ Natural
```

In this context, `Zero` and `Succ` can be used both as terms and types. Just as values are promoted to types, the type `Natural` itself is promoted to something higher than a type — a *kind*. Via the `DataKinds` extension, programmers can introduce custom kinds into Haskell's type system. By default, Haskell provides two kinds: `Type` (the kind of types) and `Constraint` (the kind of type classes constraints).

Another key concept to understand type-level programming in Haskell is singletons. A type is a singleton type when it's inhabited by only one value [7]. This establishes a direct correspondence between the value and type levels. This technique allows us to emulate dependent typing in Haskell, enabling the types of expressions to depend on the values they contain.

For example, to work with the previously defined `Natural` kind, we need a way to create values that hold the types `Zero` and `Succ n`. We define a singleton type `SNat` (short for Singleton Natural) that mirrors the structure of the `Natural` type. The key difference is that `SNat` encodes the natural number it represents on its type, using the promoted constructors `Zero` and `Succ n` presented earlier.

```
data SNat (n :: Natural) :: Type where
  SZero :: SNat Zero
  SSucc :: SNat n → SNat (Succ n)
```

The relationship between these types and other common Haskell kinds is illustrated in Figure 1.

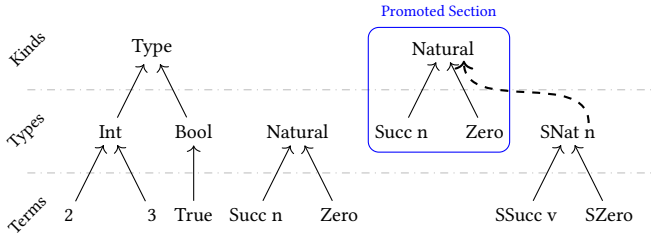

Figure 1. Hierarchy between terms, types and kinds. The dashed arrow highlights that the kind of `n` in `SNat`'s definition is `Natural`.

To support preexisting types at the type level, Haskell's base package provides the `GHC.TypeLits` module. This module includes essential definitions for natural numbers, singletons, type-level strings and lists, user-defined type errors, and a variety of useful type families. From this point forward, we will use this module in our code, meaning the previously defined `Natural` and `SNat` types will no longer be used.

By including this module, along with the `TypeApplications` extension, singleton values can be expressed concisely. Additionally, the module allows us to use type-level decimal literals, as shown below.

```
> :k 1
1 :: Natural

> :t SNat @2
SNat @2 :: KnownNat 2 ⇒ SNat 2
```

The first query asks for the kind of the type 1, which is Natural. The second query constructs a singleton natural by applying the type 2 on it.

3.2 Singleton Lists and Constraints

To ensure the rigor of quantum operations, we make use of an extension to the conventional linked list implementation. Applying concepts from type-level programming, we can store additional information about the list within its type. This information can then be used to restrict the range of functions that can be applied to the value. Doing so, we can verify whether a quantum state is well-defined and determine which quantum gates can be applied to it. This approach allows us to embed the mathematical constraints of quantum computing directly into the Haskell type system.

One of these structures, particularly useful for our purposes, is a list whose size is encoded in its type. In Haskell, this structure is usually called a vector. This datatype can be implemented by defining a list type that maintains its length concisely through its constructors. By induction, an empty list has a type that holds a size `Zero`, while the *cons* operation (`:>`) increments the size by one for each value appended.

```
type Vec :: Natural → Type → Type
data Vec n a where
  VNil :: Vec 0 a
  (:>) :: a → Vec n a → Vec (n + 1) a
infixr 5 :>
```

For example,

```
> :t 'a':>'b':>VNil
'a':>'b':>VNil :: Vec 2 Char
```

is a vector-2 of characters.

Another key datatype we need to introduce is the `SList`. We define the `SList` (short for Singleton-List or Selection-List, as we will use it later) to be a singleton type for a list of natural numbers. At type level, a `SList` represents a list of natural numbers, allowing static type checking about the contained values. At the value level, a `SList` keeps a list of `SNat` values, which can be easily converted to a list of `Int`s for runtime operations.

```
type SList :: [Natural] → Type
data SList as where
  SNil :: SList []
  (:-) :: SNat a → SList as → SList (a : as)
infixr 5 :-
```

The `SList` type serves as a selector for our quantum memory. Since it keeps type-level information about all its elements, we can perform compile-time checks to verify the correctness of a selection. For a selection list L to be a valid selector over an n-qubit memory, it must satisfy three conditions: (i) All values in L must be less than or equal to n; (ii) All values in L must be greater than 0; (iii) L must not contain any repeating values, assuring the no-cloning theorem.

The no-cloning theorem states that there does not exist a quantum operation capable of cloning an arbitrary qubit state. This constraint has profound implications for quantum algorithm design, and a strongly-typed quantum programming language must detect rule-breaking implicit cloning attempts. In our implementation, such implicit cloning would look like an index appearing multiple times in a selection list. That is exactly what the rule (iii) watches for.

Unlike ordinary type constraints, `TypeError` allows us to produce custom, human-readable error messages when a constraint is violated. It works by defining a constraint that always fails, and instead of a generic type mismatch message, the compiler emits the specified error content. This mechanism is especially useful for guiding the user when a quantum state or operation is invalid, as we can precisely indicate which rule has been broken and why. Each of the three individual constraints (`BoundCheck`, `NoZeroCheck`, and `NoCloningCheck`) uses `TypeError` to provide meaningful feedback when a selector fails to meet the required conditions.

```
type BoundCheck :: Natural -> [Natural] -> Constraint
type BoundCheck n xs
  = If (Maximum xs <=? n) ()
    (TypeError (
        Text "Index out of bounds on Qubit selection"
        :$$: Text "You got " :<>: ShowType n :<>: Text "
        qubits" :$$: Text "But tried to select qubits "
        :<>: ShowType xs))

type NoZeroCheck :: [Natural] -> Constraint
type NoZeroCheck xs
  = If (HasZero xs)
    (TypeError (
        Text "Zero qubit selection is not allowed" :$$:
        Text "The qubit selection list starts from 1"
        )) ()

type NoCloningCheck :: [Natural] -> Constraint
type NoCloningCheck xs
  = If (HasDupl xs)
    (TypeError (
        Text "No Cloning Theorem Violation" :$$:
        Text "You tried to select qubits with repetition
        " :<>: ShowType xs
        )) ()
```

To express conditional logic at the type level, we rely on the `If` type family provided by the Fcf (First-class Families) package. This type-level `If` behaves similarly to an ordinary if expression in term-level Haskell: it evaluates the first argument (a type-level boolean), and selects between the second and third branches accordingly. For instance, in the `BoundCheck` constraint, `If (Maximum xs <=? n) () (TypeError ...)` means that if the maximum index in `xs` is less than or equal to `n`, the constraint succeeds (represented by `()`); otherwise, the constraint fails with a custom error message provided by `TypeError`.

```
type ValidSelector :: [Natural] → Natural → Constraint
type ValidSelector xs n = (BoundCheck n xs, NoZeroCheck
xs, NoCloningCheck xs)
```

The `ValidSelector` constraint combines all three necessary verifications and throws a compile-time error if any of the conditions are violated.

3.3 Virtual Values

A virtual value is an abstraction over the quantum memory, allowing us to refer to some subset of it while keeping the quantum nonlocal effects. The type `Virt` contains: (i) A pointer `QR` to the whole quantum memory; (ii) A list of integer numbers indexing qubits in that memory; (iii) At type level, the abstraction length, equal to the size of the indexing list, enabling the static checks defined in the previous sections to be applied.

```
data Virt (a :: Type) (n :: Natural) :: Type where
  Virt :: QR a → [Int] → Virt a n
```

We define the necessary functions to work with the `Virt` type. It is important to note that we hide the type constructor `Virt` and expose the `mkQ` function in the module. This design ensures that users must provide a `Vec` type, inferring the type-level length without explicitly specifying it in the type constructor. The `Vec` type also provides us a simple way to check whether the user is providing a valid mapping between basis and complex values, preventing the mixture of different quantum vector spaces. By default, all qubits are indexed with the list $[1..s]$, which covers the entire quantum memory sequentially.

```
mkQ :: (KnownNat s, Basis a)
    ⇒ [(Vec s a, PA)]
    → IO (Virt a s)
mkQ pairs = do
  qr ← qrFromList pairs
  return $ Virt qr [1..fromIntegral $ natVal (Proxy @s)]
```

Analogous to the `QR` observe function, the `measureVirt` function provides a measurement implementation for the `Virt` datatype. The difference is that functions operation on virtual values acts only on the selected qubits.

```
measureVirt :: Basis a ⇒ Virt a s → Int → IO a
measureVirt (Virt qr acs) ix = ...
```

In a similar fashion, the `appV` function performs a quantum operation on the chosen qubits, as if they were a separated quantum state. The qubits not selected for the computation are left untouched. When a unitary operator U is applied to a specific subset of qubits (denoted by acs), the full operation can be expressed as $U_{acs} \otimes I_{\neg acs}$.

```
appV :: Basis a ⇒ OP a → Virt a s → IO ()
appV U (Virt ref acs) = do
  qv ← readIORef ref
  writeIORef ref $ adaptOp `appOP` qv
  where adaptOp =
      mkOP [ ((ua, ub), U `getOpProb` (a, b))
           | ua ∈ basis s, ub ∈ basis s
           , let (a, na) = decompose acs ua
                 (b, nb) = decompose acs ub
           , na == nb]
```

To select a portion of the quantum memory, we define the function `selectQ`, which takes a selection list (`SList`) and applies it to the current indexes of a virtual value. The function also adjusts the length of the virtual value to match the length of the selection list.

```
unsafeSelectQ :: SList nacs
              → Virt a n
              → Virt a (Length nacs)
unsafeSelectQ sl (Virt ref acs)
    = Virt ref (((acs !!) . pred) <$> sListToList sl)

selectQ :: ValidSelector nacs n
        ⇒ SList nacs
        → Virt a n
        → Virt a (Length nacs)
selectQ = unsafeSelectQ
```

To illustrate the selection functionality, consider the following virtual value on the state $|01\rangle$.

```
> mem ← mkQ [(0:>1:>VNil, 1)] ≡ |01⟩
```

One possible selection on this value is a rearrangement of the values.

```
> revMem = selectQ (SNat @2:-SNat @1:-SNil) mem ≡ |10⟩
```

Another possibility is selecting only the first qubit.

```
> firstQ = selectQ (SNat @1:-SNil) revMem ≡ |1⟩_|10⟩ subsystem
```

Notice that the selection of the first qubit does not truly discard quantum information. Instead, it abstracts the unselected qubits to comfortably work with slices of the quantum memory. In the last example, the full quantum state is still retained in the reference, but we have access to a projection on the designated indexes.

To ensure the selection is valid, we constrain the function with our `ValidSelector` checker. For instance, attempting to select the same qubit twice violates the rule (iii) of the constraint, triggering a type error:

```
> ⊥ = selectQ (SNat @1:-SNat @1:-SNil) mem
    ≡ error: No-cloning theorem violation
```

Similarly, trying to access out-of-bound indexes is not allowed.

```
> ⊥ = selectQ (SNat @10:-SNat @5:-SNil) mem
    ≡ error: Index out of bounds on qubit selection
```

`selectQ` can be used directly on virtual values with type safety. However, we still need to address the typing of quantum gates, and the current implementation remains quite verbose. For example, a quantum adder can be implemented as follows:

```
adderExample :: IO ()
adderExample = do
  mem ← mkQ [(1:>0:>0:>0:>VNil, 1)]

  q_124 = selectQ(SNat @1:-SNat @2:-SNat @4:-SNil) mem
  q_12 = selectQ(SNat @1:-SNat @2:-SNil) mem
  q_234 = selectQ(SNat @2:-SNat @3:-SNat @4:-SNil) mem
  q_23 = selectQ(SNat @2:-SNat @3:-SNil) mem

  appV toffoli q_124
  appV cnot q_12
  appV toffoli q_234
  appV cnot q_23
  appV cnot q_12

  printQ mem
```

In the snippet above, `toffoli` represents a 3-qubit controlled-not gate:

```
toffoli :: OP Bit
toffoli = mkOP [(([0,0,0],[0,0,0]),1),
                (([0,0,1],[0,0,1]),1),
                (([0,1,0],[0,1,0]),1),
                (([0,1,1],[0,1,1]),1),
                (([1,0,1],[1,0,1]),1),
                (([1,0,0],[1,0,0]),1),
                (([1,1,0],[1,1,1]),1),
                (([1,1,1],[1,1,0]),1)]
```

We can pinpoint the sources of syntax overhead. Each operation requires explicit selection of qubits from the memory, leading to an overwhelming repetition of the `selectQ` function prior to defining the circuit logic. This issue is magnified since each selection involves a large singleton list, the definition of which itself requires many individual singletons.

4 QAct Monad

In this section, we introduce a monadic type designed to reduce the syntactic overhead of quantum operations and provide a safe way to define and apply quantum gates ans measurements. We conclude the section presenting an implementation example of Grover's algorithm.

We introduce the monad `QAct b s a` constructed as a `ReaderT` monad transformer that holds a memory of s qudits, where b represents the dimension of each qudit (usually being 2-dimensional, for qubits). The type a represents the return value of the computation performed within the monad context. Since accessing a virtual value involves an `IO` operation, we use the monad transformer to compose the Reader and IO monads. The evaluation function `runQ` is simply `runReaderT`, which takes the initial quantum memory state as a virtual value.

```
type QAct :: Type → Natural → Type → Type
type QAct b s a = ReaderT (Virt b s) IO a

type QBitAct s a = QAct Bit s a

runQ :: QAct b s a → Virt b s → IO a
runQ = runReaderT
```

From that definition, we present four basic functions that build a `QAct`. Later on, we use these building blocks to extend the framework.

For applying a standard quantum gate, we use the function `qActMatrix`. Here, a Matrix represents a quantum logic gate (`OP`), which is constructed using `Vecs`. The use of `Vec` ensures that the size of the operation is known at compile time, preventing runtime errors related to dimension mismatches. This function uses the `ask` operation from the `Reader` monad to access the wrapped quantum state, and then applies the operation within the `IO` monad.

```
type Matrix s b = [((Vec s b, Vec s b), PA)]

qActMatrix :: Basis b ⇒ Matrix s b → QAct b s ()
qActMatrix mat = do
  let op = mkOP mat
  vv ← ask
  liftIO $ appV op vv
```

For example, the Hadamard gate – a fundamental single-qubit gate – creates an equal superposition of states when applied to a computational basis state:

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

In Haskell, the definition is similar to the matrix definition of the Hadamard gate:

```
h :: QBitAct 1 ()
h = qActMatrix [
      ((0:>VNil, 0:>VNil), 1/√2),
      ((0:>VNil, 1:>VNil), 1/√2),
      ((1:>VNil, 0:>VNil), 1/√2),
      ((1:>VNil, 1:>VNil), -1/√2)
    ]
```

Using `qActMatrix` to define a C-NOT in our implementation looks like the following:

```
cnot :: QBitAct 2 ()
cnot = qActMatrix [
      ((0:>0:>VNil, 0:>0:>VNil), 1),
      ((0:>1:>VNil, 0:>1:>VNil), 1),
      ((1:>0:>VNil, 1:>1:>VNil), 1),
      ((1:>1:>VNil, 1:>0:>VNil), 1)
    ]
```

Currently, we do not statically check whether a matrix is unitary. To accomplish such a check, we would need to model singleton complex numbers and perform type-level arithmetic, which is not possible in Haskell due to the lack of type-level floating-point numbers and its inherently precision problems.

Two of the other constructors for a `QAct` are `sample` and `measure`. The `sample` function serves as a debugging tool: it prints the current state of the quantum memory without collapsing it. In contrast, the `measure` function performs a measurement on the quantum system, collapsing the state and returning the obtained value.

```
sample :: Show b ⇒ QAct b s ()
sample = do
  virt ← ask
  liftIO $ printQ virt

measure :: (KnownNat ix, Basis b, ValidSelector '[ix] n)
⇒ SNat ix → QAct b n b
measure sn = do
  virt ← ask
  liftIO $ measureVirt virt sn
```

The app function is arguably the most important constructor for `QAct`. It plays a crucial role in the modularity of the implementation by allowing the application of a `QAct` computation to a subset of the quantum state. Specifically, app takes a selection list (`SList acs`) and a `QAct` computation to be performed on the resulting value of the selection. The

selection list defines how to extract a subset of the quantum state from the virtual value kept under the current Reader monad. This approach enables the combination of `QActs` of different sizes, as long as the selection list is a valid map between them.

```
app :: ValidSelector acs n
    ⇒ SList acs
    → QAct b (Length acs) a
    → QAct b n a
app sl act = do
  qv ← ask
  let adaptedValue = unsafeSelectQ sl qv
  liftIO $ runReaderT act adaptedValue
```

Using the Hadamard (h) and controlled-not (cnot) gates defined earlier, we can define an entangle `QAct` as such:

```
entangle :: QBitAct 2 ()
entangle = do
  app (SNat @1 :- SNil) h
  app (SNat @1 :- SNat @2 :- SNil) cnot
```

Given the composable nature of the QAct monad, the entangle operation can be used as a building block within more complex quantum circuits:

```
specialEntangle :: QBitAct 3 ()
specialEntangle = do
  app (SNat @1 :- SNat @2 :- SNil) entangle
  app (SNat @2 :- SNat @3 :- SNil) entangle
```

Also, direct composition of quantum gates is simply expressed as monadic binding. For example, consider the $\sqrt{NOT}$ gate:

```
s :: QBitAct 1 ()
s = qActMatrix [
      ((0:>VNil, 0:>VNil), 1),
      ((1:>VNil, 1:>VNil), 0 :+ 1)]

sqrtNot :: QBitAct 1 ()
sqrtNot = h >> s >> h
```

As both the Hadamard and S gates work on a single qubit, the expression is well-typed and requires no explicit selection connecting the operations.

4.1 Quoters and Labels

To improve the expressiveness of the implementation, we can use overloaded labels and quasi-quotations to build singleton lists and type-level naturals.

The `QuasiQuotes` extensions allow us to define a domain-specific language for building our program. For our purposes, it can be used to create a syntax sugar for `SLists`, which is then desugared by the compiler into the standard `SList` constructors.

We define the quasi-quoter `qb` to desugar as follows:

```
[qb|1 2 3|] ≡ SNat @1 :- SNat @2 :- SNat @3 :- SNil
```

Using the same idea, we build quoter for `Vecs` and `Virts` values:

```
[nl|1 1 0|] ≡ 1 :> 1 :> 0 :> VNil
[mkq|1 1 0|] ≡ mkQ [(1 :> 1 :> 0 :> VNil, 1)]
```

In a similar manner, the *OverloadedLabels* allows us to use identifiers whose interpretation depends on their text. Labels always have a hash prefix (#) and, in this context, are used to simplify the construction of `SNat` values:

```
#1 ≡ SNat @1
```

Deutsch's algorithm is a quantum procedure that decides whether a function $f : \text{Bit} \to \text{Bit}$ is constant [4]. The algorithm exploits quantum parallelism to evaluate the given function on all possible inputs simultaneously and then measures the control qubit. If the observed result is 0, then the function is constant. Using the improved syntax, the Deutsch's algorithm can be modeled as follows:

```
deutsch :: QBitAct 2 a → QBitAct 2 Bit
deutsch uf = do
  app [qb|1|]   h
  app [qb|2|]   h
  app [qb|1 2|] uf
  app [qb|1|]   h
  measure #1

testDeutsch :: IO ()
testDeutsch = do
  mem ← [mkq|0 1|]
  r ← runQ (deutsch cnot) mem
  case r of
    0 → print "cnot is constant"
    1 → print "cnot is balanced"
```

In this example, we execute the Deutsch's algorithm on the `cnot` operation. Since the algorithm expects a 2-qubit transform (one target and one control), the expression typechecks. This ensures that mismatches between the arity of operations are caught at compile time.

4.2 Oracles and Grover's Algorithm

Oracles are crucial operations for many quantum algorithms. They encode problem-specific information in a unitary gate, acting as a black box[8]. Oracles play a significant role on quantum algorithms speedup. As noted by Simon [12], for certain problem, within a bounded error probability, there exists a quantum oracle that can solve it in polynomial time, whereas any classical Turing machine would require exponential time to achieve the same result [2].

A oracle may be defined in terms of a classical function. A specific oracle type is the phase oracle, whose inner function is use to decide wether the input qubits will suffer a phase flip. More precisely, given a function $f : \text{Bit}^n \to \text{Bit}$, the phase oracle definition is $Z_f |x\rangle = (-1)^{f(x)} |x\rangle$. In our implementation, a phase oracle builder function is shown bellow.

```
phaseOracle :: (Basis b, KnownNat n)
  ⇒ (Vec n b → Bool)
  → QAct b n ()
phaseOracle f = do
  let op = mkOP [((b,b), if f b then -1 else 1)
                | b ∈ basis n)]
  vv ← ask
  liftIO $ appV op vv
```

For example, consider the phase oracle zAny that is built upon the function $f\,|x\rangle = |x\rangle \neq |000\rangle$.

```
zAny :: QBitAct 3 ()
zAny = phaseOracle ([nl|0 0 0|] ≠)
```

We can evaluate the zAny oracle on different possible entries, and, as expected, all entries except $|000\rangle$ suffer a phase shift.

```
> runQ zAny =<< |011⟩ ≡ − |011⟩
> runQ zAny =<< |010⟩ ≡ − |010⟩
> runQ zAny =<< |000⟩ ≡ |000⟩
```

Grover's algorithm presents a powerful use case for oracles. By evaluating an oracle on superposition states, we can perform searches on unstructured data. The algorithm starts by creating a uniform superposition state by applying a Hadamard gate on each qubit (i.e., `appAll h` $\equiv H^{\otimes n}$). The oracle is then evaluated on this state, effectively searching over all entries simultaneoulsy via quantum parallelism.

Next, the Grover's Diffusion Operator – implemented as `appAll h >> zAny >> appAll h`, corresponding to the transform $H^{\otimes n} Z_{any} H^{\otimes n}$) performs a reflection relative to the average system's state. At the end of this step, the amplitude of the phase-shifted state is increased. By repeating this process $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ (N being the number os possible inputs) times, Grover's algorithm maximizes the probability of measuring the desired state.

In the implementation bellow, we define the `grover` operation for three qubits. Notice that the diffusion step is repeated $\lfloor \frac{\pi}{4}\sqrt{2^3} \rfloor = 2$ times utilizing the `replicateM_` function.

```
grover :: QBitAct 3 () → QBitAct 3 (Vec 3 Bit)
grover zf = do
  let targets = [qb|1 2 3|]

  appAll h
  replicateM_ 2 (
    zf >> appAll h >> zAny >> appAll h
    )
  measureN targets

testGrover :: IO ()
testGrover = do
  outcome ← [mkq|0 0 0|]
    >>= runQ (grover $ phaseOracle ( == [nl|1 0 1|] ))
  print outcome
```

The `testGrover` function initializes the virtual value for the memory, and applies the algorithm. When executed, it prints the result of the measurement, which is expected to be $|101\rangle$ with high probability.

5 Conclusion

This work presented a type-safe model of quantum computing in Haskell by reworking Sabry's original framework. Our main contribution is the introduction of a general mechanism for accessing and manipulating quantum memory slices while not relying on a built-in linear type system. The implementation allowed managing the state of quantum values with type safety and expressiveness. We have written several algorithms other than Grover's and Deutsch's presented earlier.

Our implementation reinforces the use of strong type systems for formal verification of invariants. It prevents invalid operations such as out-of-bounds selections, violations of the no-cloning theorem, and operation size mismatches at compile time. Moreover, the monadic structure of our abstraction enables clean composability of quantum operations, making the development of quantum algorithms ergonomic and extensible.

While previous approaches such as the use of Arrows [15] and linear type systems [1, 3, 9] provide powerful abstractions for quantum computation, our solution offers an alternative without requiring linearity or category-theoretic overhead. These features make our approach not only suitable for practical quantum algorithm prototyping on non-linear type systems but also a valuable introduction to advanced functional programming and type-level programming.

Nonetheless, the current implementation assumes a fixed memory size. This limits the ability to write algorithms polymorphic over the number of qubits. As a result, it limits the operability of many important algorithms – such as Shor's, Deutsch-Jozsa's, and Grover's – which are designed to work over inputs of varying sizes.

As future work, we consider the following extensions:

- **Polymorphism over memory size**: Currently, every `QAct` must have a fixed size. As mentioned, this would have significant impact on *Deutsch-Jozsa's* and *Grover's* algorithm design.
- **Operation History**: By using monad transformers, we can combine additional monads alongside `Reader` and `IO`. We suggest incorporating the `Writer` monad to store the quantum operations performed. This approach could enable the exploration of quantum algorithm reversibility [13] or circuit visualization.

The repository containing the full implementation, along with examples and documentation, is available on GitHub (see link in the introduction). We hope that the repository will be both a handy practical resource for developing quantum algorithms, as well as a source for interested researchers in the interface between quantum computation and type-level programming.

REFERENCES

[1] T. Altenkirch and J. Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 249–258. doi:10.1109/LICS.2005.1

[2] André Berthiaume and Gilles Brassard. 1994. Oracle quantum computing. *Journal of modern optics* 41, 12 (1994), 2521–2535.

[3] Liliane-Joy Dandy, Emmanuel Jeandel, and Vladimir Zamdzhiev. 2023. Type-safe Quantum Programming in Idris. In *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings* (Paris,

France). Springer-Verlag, Berlin, Heidelberg, 507–534. doi:10.1007/978-3-031-30044-8_19

[4] D. Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. In *Proc. R. Soc. Lond.* 97–117. doi:10.1098/rspa.1985.0070

[5] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. *SIGPLAN Not.* 47, 12, 117–130. doi:10.1145/2430532.2364522

[6] John Hughes. 2005. Programming with Arrows. In *Advanced Functional Programming*, Varmo Vene and Tarmo Uustalu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–129.

[7] Sandy Maguire. 2019. *Thinking with Types: Type-Level Programming in Haskell* (1th ed.).

[8] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press.

[9] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *SIGPLAN Not.* 52, 1 (Jan. 2017), 846–858. doi:10.1145/3093333.3009894

[10] Amr Sabry. 2003. Modeling quantum computing in Haskell *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 39–49. doi:10.1145/871895.871900

[11] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science.* 124–134. doi:10.1109/SFCS.1994.365700

[12] D.R. Simon. 1994. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science.* 116–123. doi:10.1109/SFCS.1994.365701

[13] André van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135. doi:10.1137/S0097539703432165 arXiv:https://doi.org/10.1137/S0097539703432165

[14] Andre van Tonder and Miquel Dorca. 2003. Quantum computation, categorical semantics and linear logic. (10 2003). arXiv:quant-ph/0312174

[15] Juliana Vizzotto, Thorsten Altenkirch, and Amr Sabry. 2006. Structuring quantum effects: superoperators as arrows. *Mathematical. Structures in Comp. Sci.* 16, 3, 453–468. doi:10.1017/S0960129506005287