# Embedding Quantum Programming in Haskell

Report as coursework of “Advances in Programming Languages”,
University of Edinburgh, 11 November 2016

Lukas Convent

**Abstract**

While the field of quantum computation is growing, quantum programming languages are becoming more and more important. Using Haskell as a pure functional host language, we present how Haskell's type system, in particular through the use of monads, enables an elegant separation between the declaration and execution of quantum programs. We discuss two implementations (Quipper, Quantum IO monad) and demonstrate their applicability by the example of Grover's search algorithm.

## 1 Introduction

While computer scientists still only have the prospect of quantum devices as a tool for computation, the field of quantum information is blossoming, producing efficient algorithms for problems whose classical solutions are not expected to be as efficient. To write down algorithms, to test them on classical computers and in general to better understand the potential of quantum computation, there is a need of quantum programming languages.

After giving a brief overview over quantum computation, this report describes how Haskell can be used as a host language to embed a quantum programming language by presenting Green et al.'s Quipper [6] and Green's and Altenkirch's Quantum IO monad (QIO monad) [1]. The focus lies on how Haskell's type system, in particular via monadic structures enabled by parametric types and type classes, can be harnessed for an elegant embedding. We describe how monads allow a strict separation between program declaration and their execution, allowing for different quantum program interpretations such as nondeterministic execution or quantum circuit generation. Furthermore, we sketch how type classes are used to apply quantum programming to arbitrary data structures in a generic way.

We demonstrate the presented concepts by an example of Grover's search algorithm [7], showing how a single query to an election oracle can give away substantial information about the election's outcome (the full source code is attached as an appendix). We conclude by pointing out the relevant resources this report is based on and other work done in the field of (functional) quantum programming.

## 2 Quantum Computation

To have a self-contained paper, we briefly present the basic notions of quantum computation with their corresponding notation and refer for a thorough introduction to [8]. We call the quantum counterpart to the classical bit a **qubit**. Qubits can be implemented through different physical

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Figure 1: Entangled state $|\Phi^+\rangle$

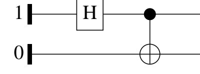


Figure 2: Circuit with output $|\Phi^+\rangle$

constructions, such as polarized photons. Mathematically, their state is represented as follows. Whereas a classical bit is given by one of two values 0 or 1, the qubit is given by unit vectors of a two-dimensional Hilbert space over the complex numbers. This is exactly the vector space $\mathbb{C} \times \mathbb{C}$ with inner product $\langle u|v\rangle = \overline{u_1} * v_1 + \overline{u_2} * v_2$, of which we only consider vectors with $\langle u|u\rangle = 1$. We use the notation $|\cdot\rangle$ for states and among the bases which span this vector space, we identify the **computational basis** $\{|0\rangle, |1\rangle\}$ with

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

We use $|0\rangle$ (resp. $|1\rangle$) as a representation for the bit value 0 (resp. 1). In general, every state can be written in terms of the computational basis:

$$|u\rangle = \alpha|0\rangle + \beta|1\rangle$$

What does this infinite amount of states represent and how is it useful? First, if not identical to $|0\rangle$ or $|1\rangle$, $|u\rangle$ is said to be in superposition with respect to $|0\rangle$ and $|1\rangle$ where the parameters $\alpha$ and $\beta$ form a probability distribution of how a state changes when **measured** in the computational basis[1]. For a state $|u\rangle$, the outcome of the measurement (i.e. the changed state) is $|0\rangle$ with probability $\alpha^2$ and $|1\rangle$ with probability $\beta^2$. Besides measurement, the state of a qubit can me changed by unitary operators, also called quantum gates. A **unitary operator** $U$ can be represented by a $2 \times 2$ complex-valued matrix $U$ s.t. $U^\dagger U = UU^\dagger = I$, with $U^\dagger$ being the adjoint of $U$. Examples of unitary operators are the **not-gate** $\boldsymbol{X}$ and the **Hadamard gate** $\boldsymbol{H}$. $X$ flips the parameters $\alpha$ and $\beta$ in the computational basis. $H$ can be used to flip the state $|0\rangle$ to $|+\rangle$ and $|1\rangle$ to $|-\rangle$ (and vice versa, since $H$ is unitary), which are equally in superposition between $|0\rangle$ and $|1\rangle$.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \qquad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

We now come to the description of a **compound qubit system**. Its space is described by the tensor product of its subspaces, e.g. the compound system consisting of two single qubit systems $\mathcal{H}_1$ and $\mathcal{H}_2$ is described by $\mathcal{H}_1 \otimes \mathcal{H}_2$. This space can now be spanned by the cartesian product of the single system's basis vectors, e.g. $\mathcal{H}_1 \otimes \mathcal{H}_2$ can be spanned by $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. Note that $|00\rangle$ is a shorthand notation for $|0\rangle \otimes |0\rangle$. Measurements and unitary operations can now be performed on both single qubits and on the whole compound system.

A special quantum phenomenon which can only be observed in this multi-qubit setting is called **entanglement**. Consider a two-qubit system which is prepared in the $|\Phi^+\rangle$ state (one of the Bell states), as shown in Fig. 1. If we measure either of the two qubits, the other qubit's state will be determined to have the same state[2]. Therefore, after the measurement the system is either in state $|00\rangle$ or $|11\rangle$.

[1] Measurements can be performed in any orthonormal basis, measuring in a different basis though can be simulated by an appropriate pre-transformation and a consequent measurement in the computational basis.

[2] Fascinatingly, the two qubits can be arbitrarily far apart from each other and this counter-intuitive effect is still observable.

Quantum algorithms consist of unitary operations and measurements. Typically, measurements are only made at the end, following a cascade of unitary operations (quantum gates). Besides their matrix representation, the common notation for cascades of unitary operations are **quantum circuits**. For instance, Fig. 2 shows a quantum circuit which produces the $|\Phi^+\rangle$ state. There is one common quantum gate in this circuit which we have not introduced yet, the **controlled-X-gate**. In the given circuit, the upper qubit controls whether the not-gate (the $\oplus$ corresponds to the $X$ gate) is applied to the lower qubit.

$$\textit{controlled-}X(|00\rangle) = |00\rangle \qquad \textit{controlled-}X(|01\rangle) = |01\rangle$$
$$\textit{controlled-}X(|10\rangle) = |11\rangle \qquad \textit{controlled-}X(|11\rangle) = |10\rangle$$

Because the upper qubit is in superposition (due to the Hadamard gate), whether the qubit flip happens or not depends completely on the upper qubit's state, thus resulting in an entangled state. The controlled-X-gate can be generalized to a controlled-U-gate (for an arbitrary gate U) and to controlled gates with more than one control qubit.

# 3 Quantum Programming

Because quantum computers are not yet in the state of being fully deployable, we do not know yet what hardware model will come out on top. However, investigating quantum programming before it is used in the real world seems a very good idea since it will give us a tool for thinking about quantum algorithms and enables us to anticipate pitfalls when eventually designing them for real quantum devices. Following Valiron's proposals [13], we look at some important properties which can be expected of a quantum programming language. In the next section then, we describe in which ways Quipper and the QIO monad achieve to satisfy them.

**Runtime abstraction.** The hardware interface sets constraints on how a programming language needs to be designed and what it will compile to. For example, most classical computers use the RAM model for memory management and most programming languages allow the possibility of allocating memory space and using it during the runtime of a program. For qubits, holding on to the state of a qubit turns out to be very expensive, if possible at all for longer durations of time. Instead of a standard computer architecture with fetch and write instructions, a circuit architecture with direct qubit flow might be more applicable. In either way, since we can only test our programs on classical computers, interpreting programs in different ways is desirable, requiring abstraction from execution on a high level.

**Quantum data structures.** First, there needs to be a clear interface between quantum data and classical data. A quantum programming language should enable the definition of larger data types. For every data type, classical instances should be convertible into quantum representation and every quantum instance should be measurable, resulting in a classical outcome.

**Operations on quantum programs.** Certain operations which can be performed on quantum programs in a general way. Here is an example: Because all gates are unitary, i.e. reversible, larger combinations of gates (program blocks) are reversible, too. The reversion of gate combinations should therefore be enabled in an automatic fashion. Other examples are iteration, circuit transformations and optimizations.

# 4 Embedding in Haskell's Type System

With respect to the desiderata depicted in the last section, many approaches to designing a quantum programming language have been made [4, 13]. We focus on how Haskell can be used as a host language to create an embedded quantum programming language fulfilling the requirements and emphasize at which points it excels. In particular, we focus on the language Quipper by Green et al. [6] but also compare it to Green's and Altenkirch's QIO monad [1].

## 4.1 Program declaration

At the core of both Quipper and the QIO monad lies the use of monadic structures for program declarations. We give a brief introduction to monads and then focus on how and why the concept is applied.

### 4.1.1 Monads

Monadic structures can be identified on type constructors of kind[3] `* -> *`, e.g. on the `[]` (list) or the `Maybe` type constructor. Intuitively, for a type constructor to have a monadic structure means that it offers support for pipelining computations. Because we think that the comparison to a computational pipeline is useful for understanding monads, we use it in the following as a figurative term for monad. In Haskell, a type constructor capable of pipelining computation can be identified by instantiating the `Monad` type class. We give an example of a monad instance in section 4.1.2.

```
class Monad where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m a) -> m b
```

First, every value can be put into a pipeline (`return`). Second, if there is a pipeline, another computation step can be added to it, feeding the output of that pipeline as input to the added computation step (`>>=`). Noticeable is that the type of computed intermediate values may vary throughout the pipeline, only the last element of the pipeline establishes the type of the resulting pipeline.

Another important aspect is that there are no further constraints on the type constructor `m`, thereby allowing for a powerful tool of encapsulation. In the case of `[]`, we can do pattern matching and apply the usual operations (like `head`, `tail`) on a pipeline, because it is exactly the list resulting from the pipelined computations. On the other hand, the type constructor `m` could also be abstract, taking away any chance of inspecting the "inside" of the pipeline. This is used to implement a real-world interface in Haskell: All interaction with the real world is described by the `IO` pipeline, but values will never get out of the pipeline, keeping Haskell a pure language.

### 4.1.2 Quantum programs as monads

We describe now how quantum programs are represented as monads. Their implementations for Quipper and the QIO monad make use of the same essential ideas, but because the QIO monad is less complex and seems more concise to us, we explain the concept on this example and then refer to the Quipper implementation afterwards.

Quantum programs in the QIO monad consist of only two datatypes, unitary gates `U` and programs `QIO`.

[3] The kind of a type constructor corresponds to the "type" of a type constructor, with `*` being the kind of all instantiable types

```
data U = UReturn                    data QIO a = QReturn a
       | Rot Qbit Rotation U                   | MkQbit Bool (Qbit -> QIO a)
       | Cond Qbit (Bool -> U) U               | ApplyU U (QIO a)
       | ...                                   | Meas Qbit (Bool -> QIO a)
```

Unitary gates (`U`) can be interpreted as lists of matrices which can be executed in line (matrix multiplication). Note that the `Cond` constructor introduces no branching but merely implements the controlled-U gates. Programs (`QIO a`) can make use of gates (`ApplyU`), allocate qubits (`MkQbit`) and measure them (`Meas`). They constitute a higher level than unitary gates, as their execution actually produces a value of type `a`. With the exception of `QReturn`, all of these computational basical blocks (`MkQbit`, `ApplyU` and `Meas`) have specified "continuations", i.e. programs to be executed afterwards. If we want to pipe two programs together, we can simply pipe the second program onto the continuation of the first one - this instantiates exactly the monad pattern.

```
instance Monad QIO where
  return = QReturn
  (QReturn v)  >>= f  = f v
  (MkQbit b g) >>= f  = MkQbit b (\x -> g x >>= f)
  (ApplyU u q) >>= f  = ApplyU u (q >>= f)
  (Meas x g)   >>= f  = Meas x (\b -> g b >>= f)
```

To give future program parts access to an allocated qubit or a measurement outcome, we have to make it available, i.e. pass it down the pipeline.

```
mkQbit :: Bool -> QIO Qbit          applyU :: U -> QIO ()
mkQbit b = MkQbit b return          applyU u = ApplyU u (return ())

measQbit :: Qbit -> QIO Bool
measQbit x = Meas x return
```

In the following is a first example of a quantum program which uses a Hadamard gate to produce the $|+\rangle$ state (this common example is also given in [9]). We give it in three equivalent versions: First, following the syntax as just introduced. Second, we introduce the use of Haskell's do-notation, which is provided as syntactic sugar for monads and allows for an imperative style. Third, we present the same program using the Quipper language.

```
qPlus :: QIO Qbit                    qPlus :: QIO Qbit      qPlus :: Circ Qubit
qPlus =                              qPlus = do             qPlus = do
  (mkQbit True) >>=                      x <- mkQbit True       x <- qinit True
  (\x -> (applyU (uhad x)) >>=           applyU (uhad x)        x <- hadamard x
         (return x))                     return x               return x
```

Note that besides different naming, the only visible difference between the QIO monad and Quipper is that applying unitary gates may not produce a unit value (`QIO ()`) but a reference to the affected qubit, as it is for the `hadamard` gate the case. This is just a matter of style though, not effecting the language's expressivity.

## 4.2 Program execution

By describing quantum programs through monads, we can now take a look at how this helps with us to fulfill the desired property of runtime abstraction. The key term here is that monads have given us a way to *describe* programs, allowing room for interpretation later on. We find three main interpretations of programs written with Quipper and the QIO monad:

1. **Nondeterministic classical execution**. Instead of letting nature determine the outcome of measurements, we can use a pseudo-random number generator. This classical simulation is generally inefficient, because the space required for each stored qubit doubles with every additional qubit (due to more potential superpositions) [3]. It is also noteworthy that Haskell is strictly pure, allowing nondeterminism only through encapsulation via the `IO` monad.

2. **Computing probability distribution for output states**. The overall resulting state of the whole system (possibly entangled) can be described as a superposition with corresponding probabilities (for a one-qubit system through $\alpha^2$ and $\beta^2$). This simulation is as inefficient as the nondeterministic simulation, for the same reasons.

3. **Generating quantum circuit**.

We now focus on how in general one can implement different interpretations on top of quantum programs that were defined through monads. To explain the concept, we follow the implementation of the QIO monad and simplify it a little. We emphasize that the key idea of monad transformation is still captured and is implemented very similarly in both the QIO monad and Quipper.

The key idea lies in transforming programs of the `QIO` monad into another monad which implements the desired interpretation. In the case of nondeterministic classical execution this will result in a procedure `run :: QIO a -> IO a`. Similarly, we can define a monad for computing a probability distribution or generating a quantum circuit.

To keep track of allocated qubits, their states and other data, program execution is stateful. We can thus define a program execution as a function taking a state and outputting the result[4]. We assume a type `State` that represents the program state. We can now describe what it means for a program (`:: QIO a`) to be executed (`:: State -> IO a`), by giving the following implementation sketch.

```
ioExec :: QIO a -> State -> IO a
ioExec (QReturn v)  s = return v
ioExec (MkQbit b g) s = ioExec (g new-qubit) update-s
ioExec (ApplyU u q) s = ioExec q update-s
ioExec (Meas x g)   s = do outcome <- use-boolean-random-gen
                           ioExec (g outcome) update-s-according-to-outcome
```

If the program simply returns a value, we let the execution return it, too. In case of a qubit allocation, the added qubit will be tracked in future states and the rest of the program is executed. Applying a quantum gate results in an update of state. Measuring involves a random boolean generator with odds determined by the state (for a one-qubit system, the odds are represented by $\alpha^2$ and $\beta^2$). Note that this is the reason for being compelled to the `IO` monad, since the random boolean generator introduces nondeterminism.

Finally, we can implement `run`, relying on a defined `initialState`.

```
run :: QIO a -> IO a
run e = ioExec e initialState
```

## 4.3 Data structures and operations on quantum programs

Generalising the connection between a bit and a qubit, each classical data structure has a quantum companion. Quipper adds another level to this by distinguishing classical data on the meta-level

[4]As a generalization, stateful program execution can be defined as a function taking a pre-state to a post-state, thereby outputting a result. This corresponds to the State monad.

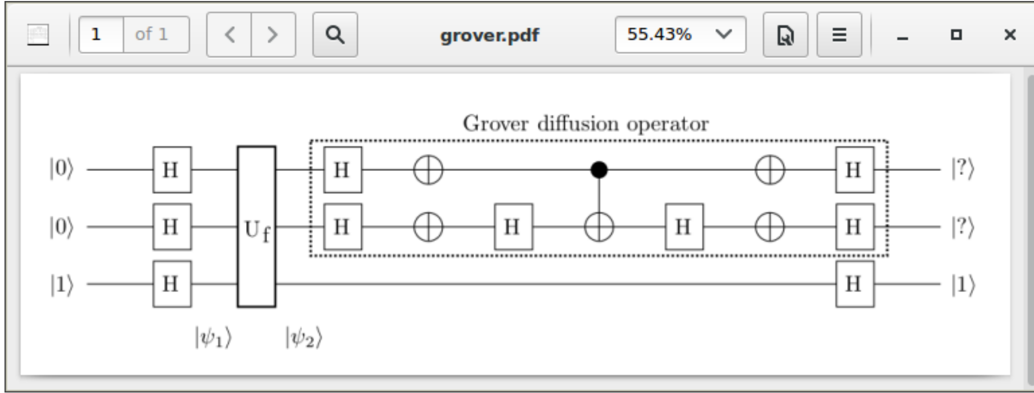


Figure 3: Grover's algorithm for $n = 2$ (Annotated Quipper-generated circuit)

(i.e. outside of the `Circ` monad) from classical measurement outcomes within the `Circ` monad. By using a type class `QShape`, the connection between the three versions can be established and even extended to complex structures in a general way.

```
instance QShape (Bool, Qubit, Bit)
instance (QShape b q c, QShape b' q' c') => (QShape (b, b') (q, q') (c, c'))
```

This type class can now be used for a polymorphic declaration of allocation and measurement procedures.

```
qinit :: QShape b q c => b -> Circ q
measure :: QShape b q c => q -> Circ c
```

Finally, we sketch how operations on circuits in a generic way can be implemented. The course of action is rather straightforward: Because we have quantum programs as monads at hand, we can perform operations to restructure them (e.g. concatenation) and to change their elements (e.g. gate substitution, reversion). As an example, the procedure `reverse_simple` reverses a quantum program confined to a special class of input data.

```
reverse_simple :: (QCData_Simple x, QCData y) =>
                    (x -> Circ y) -> (y -> Circ x)
```

# 5 Example: Grover's Algorithm

We now present an example of a famous algorithm, Grover's search algorithm [7]. The specification can be clearly explained on the following situation: Assume that for an oncoming election, there is an oracle which can predict which of the four possible parties will get the largest popular vote. Questioned about one of the four parties, the oracle can only answer with "yes" or "no". To find out about the winner, the worst-case scenario would be that we have to query the oracle three times before we know the answer. Grover's search algorithm lets us solve the task by one single query.

In a simple formal setting, we have as **input** a function $f : \{0,1\}^n \to \{0,1\}$ for which an unknown $u$ exists s.t. $f(u) = 1$ and $f(x) = 0$ for $x \neq u$. As **output** we get $u$, i.e. we invert $f$ at its image 1. For the election scenario with $n = 2$ (4 parties can be encoded by 2 bits) we have the circuit as shown in Fig. 3 and present a quantum program in Quipper.

```haskell
grover :: Circ ((Qubit, Qubit), Qubit)
grover = do
  ((x1, x2), y) <- qinit ((False, False), True)
  x1 <- hadamard x1                       -- Hadamard on each input
  x2 <- hadamard x2
  y <- hadamard y
  ((x1, x2), y) <- oracle ((x1, x2), y)   -- Query Oracle
  (x1, x2) <- diffusion_op (x1, x2)       -- Apply diffusion operator
  y <- hadamard y                         -- Hadamard on y
  return ((x1, x2), y)
```

We do not explain the whole algorithm in detail, but sketch briefly how the components take part in the computation. First of all, we apply the Hadamard gate to each input qubit, leading to state $|\psi_1\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right)$. Next comes the oracle gate $U_f$ which flips the lowest qubit for exactly one combination $|00\rangle$, $|01\rangle$, $|10\rangle$ or $|11\rangle$, namely for the unknown $|u\rangle$. This will mark that part of the superposition by a minus-sign, e.g. for $u = 11$ we have $|\psi_2\rangle = (|00\rangle + |01\rangle + |10\rangle - |11\rangle)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right)$. The Grover diffusion operator will then isolate the part which is tagged by the minus and gives it as output, i.e. for $u = 11$ the final states of the first two qubits are both $|1\rangle$.

# 6 Resources, Related Work and Conclusion

All concepts described in this report are implemented in Quipper [6] and the QIO monad [1]. Sources of both are available online [11, 9] and are well documented, especially the concise implementation of the QIO monad allows good insight.

Quipper is more complex than the QIO monad and much more than a proof of concept, aiming at providing as much support as possible for the progammer in various aspects of quantum programming: Automatic conversion from classical circuits to quantum circuits, circuit decomposition and analysis tools for quantum circuits are only a few examples. Its potential is demonstrated by the implementation of several quantum algorithms. Very helpful as an introduction to Quipper is [5] as a hands-on tutorial, demonstrating the syntax and concept on a quantum teleportation example.

The QIO monad aims at a clean and concise implementation of quantum programs as monads while at the same time being as powerful as Quipper. Its focus on simplicity allows the authors to present all core structures in the paper itself, allowing for a strong connection between conceptual idea and implementation.

Apart from these two approaches to an embedding in Haskell, the area of functional quantum programming has been explored in further directions. An overview over quantum programming languages in general has been given by Gay [4], depicting the different directions taken as of until 2006, also among functional quantum programming. Valiron has published a two-parted tutorial on quantum computation [12] (part 1) and more specifically on quantum programming [13] (part 2), together examining quantum computation "from a progammer's perspective".

Rather theoretical research has been undertaken by van Tonder [14] and Selinger and Valiron [10] on investigating how the lambda calculus can be transferred to the quantum domain.

**Conclusion.** By using principal notions of the pure functional programming language Haskell such as monadic structures, it has proved to enable a clean separation of quantum program declaration and execution, making it possible to define different interpretations of the execution of a quantum program. Furthermore, by relating classical and quantum versions of a data type through type

classes, statically checkable constraints can be established, achieving automatic type class inference at the same time for convenient programming. As expected, certain wrong program elements are not eliminated by the type system, e.g. the violation of the no-cloning theorem by re-measuring a state although it has already evolved. The authors of Quipper are working on an extension of the type system to become linear, thereby statically enforcing the single use (measurement) of a quantum state [6].

From a more distant perspective, the presented concept of quantum programming in Haskell can be seen as a specific example of language-embedding in Haskell, demonstrating how the full power of its type system can be taken advantage of. An example of a different application is the work by Claessen, who has investigated this concept with respect to hardware description and verification [2].

# References


[1] Thorsten Altenkirch and Alexander S. Green. "The Quantum IO Monad". In: *Semantic Techniques in Quantum Computation.*

[2] Koen Claessen. "Embedded languages for describing and verifying hardware". In: (2001).

[3] Richard P. Feynman. "Simulating physics with computers". In: *International journal of theoretical physics* 21.6 (1982), pp. 467–488.

[4] Simon J. Gay. "Quantum programming languages: Survey and bibliography". In: *Mathematical Structures in Computer Science* 16.4 (2006), pp. 581–600.

[5] Alexander S. Green et al. "An introduction to quantum programming in Quipper". In: *International Conference on Reversible Computation.* 2013, pp. 110–124.

[6] Alexander S. Green et al. "Quipper: a scalable quantum programming language". In: *ACM SIGPLAN Notices.* Vol. 48. 2013, pp. 333–342.

[7] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *arXiv:quant-ph/9605043* (May 29, 1996). arXiv: `quant-ph/9605043`.

[8] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* 2010.

[9] *QIO: The Quantum IO Monad is a library for defining quantum computations in Haskell.* URL: `https://hackage.haskell.org/package/QIO` (visited on 10/11/2016).

[10] Peter Selinger and Benoit Valiron. "A lambda calculus for quantum computation with classical control". In: *Mathematical Structures in Computer Science* 16.3 (2006), pp. 527–552.

[11] *The Quipper Language.* URL: `http://www.mathstat.dal.ca/~selinger/quipper/` (visited on 10/11/2016).

[12] Benoît Valiron. "Quantum computation: a tutorial". In: *New Generation Computing* 30.4 (2012), pp. 271–296.

[13] Benoît Valiron. "Quantum computation: from a programmer's perspective". In: *New Generation Computing* 31.1 (2013), pp. 1–26.

[14] André Van Tonder. "A lambda calculus for quantum computation". In: *SIAM Journal on Computing* 33.5 (2004), pp. 1109–1135.

# A Full Example: Grover's Algorithm

```
import Quipper
import QuipperLib.Simulation
import System.Random

-- This example oracle is essentially the Toffoli gate, flipping y iff x1 = x2 = |1>
-- This corresponds to the unknown u = (1, 1)
oracle :: ((Qubit, Qubit), Qubit) -> Circ ((Qubit, Qubit), Qubit)
oracle ((x1, x2), y) = do
  y <- qnot y `controlled` (x1, x2)
  return ((x1, x2), y)

diffusion_op :: (Qubit, Qubit) -> Circ (Qubit, Qubit)
diffusion_op (x1, x2) = do
  x1 <- hadamard x1
  x2 <- hadamard x2
  -- next stage
  x1 <- qnot x1
  x2 <- qnot x2
  -- next stage
  x2 <- hadamard x2
  x2 <- qnot x2 `controlled` x1
  x2 <- hadamard x2
  -- next stage
  x1 <- qnot x1
  x2 <- qnot x2
  -- next stage
  x1 <- hadamard x1
  x2 <- hadamard x2
  return ((x1, x2))

grover :: Circ ((Qubit, Qubit), Qubit)
grover = do
  ((x1, x2), y) <- qinit ((False, False), True)
  x1 <- hadamard x1                          -- Hadamard on each input
  x2 <- hadamard x2
  y <- hadamard y
  ((x1, x2), y) <- oracle ((x1, x2), y)  -- Query Oracle
  (x1, x2) <- diffusion_op (x1, x2)      -- Apply diffusion operator
  y <- hadamard y                            -- Hadamard on y
  return ((x1, x2), y)

-- Print PDF of generated circuit
main_gen_circ :: IO ()
main_gen_circ = print_simple PDF grover

-- Nondeterministic execution
main_run :: IO ()
main_run = do
  g <- newStdGen
  print $ run_generic g (0.0 :: Double) grover

main :: IO ()
main = main_gen_circ
```