Functional Quantum Programming

Shin-Cheng Mu Richard Bird

DRAFT

Abstract

It has been shown that non-determinism, both angelic and demonic, can be encoded in a functional language in different representation of sets. In this paper we see quantum programming as a special kind of non-deterministic programming where negative probabilities are allowed. The point is demonstrated by coding two simple quantum algorithms in Haskell. A monadic style of quantum programming is also proposed. Programs are written in an imperative style but the programmer is encouraged to think in terms of values rather than quantum registers.

1 Introduction

It was noted by [4] in 1982 a profound difference between the nature of physical evolution under the laws of quantum physics and classical physics. Simulation of quantum mechanics on a classical computer usually faces an exponential slowdown in running time. On the other side of the same coin, this also suggests the potential power of a new generation of machines based their laws on quantum physics.

Since then a new branch of technology bloomed to explore this new possibility. Efforts has been put into realisation of quantum computing devices. Developments on algorithms for those machines, however, is still slow. It was not until a decade later when the first quantum algorithm attracted people's attention was presented by Deutsch and Jozsa [2]. Some impressive breakthrough was made since then, the most famous one being Shor's factorisation algorithm. Further improvements again slowed down. Today we have a small number of quantum algorithms, most of them variations of each other and can be classified into two main categories [11].

It seems like a good chance for we computing scientists, with knowledge about algorithms and programming languages, to get involved. However, quantum physics itself is a subject one can spend a life's effort on. It is thus desirable to build an abstraction layer upon the details of physics, pretty much like how computing scientists today invent algorithms without knowing how the electrons flow in the chips. Particularly favourable would be a programming language designed for quantum computing, yet allowing reasoning the way we used to do. Previous attempts has yielded promising results. In particular, a extension of

Dijkstra's guarded command language was proposed in [9, 11], incorporating a calculus for program refinement.

This paper explores a complementary approach: to simulate quantum programming in a functional language. The idea of embedding non-determinism in a functional language through different representation of sets has been proposed in [7]. Quantum programming can be seen as one of its special case involving probability. We will illustrate this by coding some quantum algorithms in Haskell. Finally, a monadic style for writing quantum programs is proposed.

2 Non-determinism and the list monad

Before going into quantum computing, we will first review some known facts about lists, list monads, and their use for modelling non-determinism. As an example, consider the problem of computing an arbitrary (consecutive) segment of a given list. An elegant way to formulate the problem is to use two relations, or two non-deterministic functions *prefix* and *suffix*. The relation $\mathit{prefix} :: [a] \to [a]$ non-deterministically returns an arbitrary prefix of the given list, while $\mathit{suffix} :: [a] \to [a]$ returns an arbitrary suffix. The problem is thus formulates as $\mathit{segment} = \mathit{suffix} \cdot \mathit{prefix}$.

Working in a functional language, however, we do not have non-deterministic functions, as they violate the basic requirement for being a function − to yield the same value for the same input. Nevertheless, non-deterministic functions can be simulated by functions returning the set of all possible solutions[7]. A function *prefixes* returning the set of all prefixes of the input list can be defined by:

$$\begin{array}{lcl} \mathit{prefixes} & :: & [a] \to \mathit{Set}\,[a] \\ \mathit{prefixes}\,[\,] & = & \mathit{singleton}\,[\,] \\ \mathit{prefixes}\,(a : x) & = & \mathit{singleton}\,[\,]\,\text{`}\mathit{union}\text{`}\,\mathit{map}\,(a\,:)\,(\mathit{prefixes}\,x) \end{array}$$

where *singleton* returns a singleton set, *union* performs set union, and *map* is overloaded for sets. One possible representation of sets in Haskell is via lists, i.e,

$$\textbf{type}\,\mathit{Set}\ a = [a]$$

In this case, the two set constructing functions above can simply be defined by $\mathit{singleton}\,x = [x]$ and $\mathit{union} = (+\!\!+)$. Similarly, *suffices* can be defined by:

$$\begin{array}{lcl} \mathit{suffixes} & :: & [a] \to \mathit{Set}\,[a] \\ \mathit{suffixes}\,[\,] & = & \mathit{singleton}\,[\,] \\ \mathit{suffixes}\,(a : x) & = & (a : x)\,\text{`}\mathit{union}\text{`}\,\mathit{suffixes}\,x \end{array}$$

The function *segments*, which returns the set of all consecutive segments of a given list, can thus be composed as below with the help of primitive list operators *map* and *concat*.

$$\begin{array}{lcl} \mathit{segments} & :: & [a] \to \mathit{Set}\,[a] \\ \mathit{segments} & = & \mathit{concat} \cdot \mathit{map}\,\mathit{suffixes} \cdot \mathit{prefixes} \end{array}$$

The use of a *concat* after a *map* to compose two list-returning functions is a general pattern captured by the list monad. This is how the $(\gg\!=)$ operator for the list monad is defined.

$$x \gg\!= f = concat\,(map\,f\,x)$$

The instance of $(\gg\!=)$ above has type $[a] \to (a \to [b]) \to [b]$. We can think of it as an *apply* function for lists, applying a list-returning function to a list of values. Furthermore, Haskell programmers are equipped with a convenient **do**-notation for monads. The functions *prefixes* and *segments* can be re-written in **do**-notation as below.

$$\begin{array}{lcl} prefixes\,[\,] & = & return\,[\,] \\ prefixes\,(a:x) & = & return\,[\,]\,\text{`}union\text{`} \\ & & (\mathbf{do}\quad y \leftarrow prefixes\,x \\ & & \qquad\quad return\,(a:y)) \end{array}$$

$$\begin{array}{lcll} segments\,x & = & \mathbf{do} & y \leftarrow prefixes\,x \\ & & & z \leftarrow suffixes\,y \\ & & & return\,z \end{array}$$

The **do**-notation gives programmers a feeling that they are dealing with a single value rather than a set of them. In the definition for *segments*, for instance, identifiers y and z have type $[a]$. It looks like we take *one* arbitrary prefix of x, calling it y, take *one* arbitrary suffix of y, calling it z, and return it. The fact that there is a whole set of values to be processed is taken care of by the underlying $(\gg\!=)$ operator.

Simulating non-determinism with sets represented by lists is similar to *angelic* non-determinism in logic programming. All the answers are enumerated in a list and are ready to be taken one by one. In fact, the list monad has close relationship with backtracking and has been used to model the semantics of logic programming [10].

It may also be the case that the programmer only needs one arbitrary segment of the given list, no matter which one it is. It can be done with the help of a function *choose* defined below.

$$\begin{array}{lcll} choose & :: & [a] \to IOa & \\ choose\,x & = & \mathbf{do} & i \leftarrow randomRIO(0, length\,x - 1) \\ & & & return\,(x!!i) \end{array}$$

It picks an arbitrary element of the set by taking a random number. The function *segment* returning an arbitrary segment can thus be written as:

$$\begin{array}{lcl} segment & :: & [a] \to IO\,[a] \\ segment & = & choose \cdot segments \end{array}$$

An obvious explanation why *choose* has to wrap the returned value in an *IO* monad is that the primitive taking a random number in the Haskell Prelude involves the *IO* monad. The more fundamental reason, however, is that it preserves the validity of many important program transformation laws. Say, beta reduction is still valid because *segment*, or *choose*, *are* functions. They always return the same *action* of choosing an element from a set, although the action may yield different results each time it is carried out.

Interestingly, if it can be made sure that *choose* is the only way the contents of the set will be accessed (say, by making the set an abstract datatype with methods *singleton*, *union* and *choose*), an optimisation can be performed. Instead of collecting all the answers in a list and make the choice in the last step, we can decide which element to choose on-the-fly at the time of set union by some unsafe operation. The "set" representation will therefore only contain one element.

$$
\begin{aligned}
&\mathbf{data}\, \mathit{Set}\ a \quad = \quad \mathit{Only}\ a \\[1em]
&\mathit{singleton} \qquad\quad = \ \mathit{Only} \\
&x \,`\mathit{union}`\, y \qquad = \ \mathbf{if}\ r\ \mathbf{then}\ x\ \mathbf{else}\ y \\
&\qquad \mathbf{where} \quad r \ = \ \mathit{unsafePerformIO}\,(\mathit{randomIO} :: \mathit{IOBool}) \\
&\mathit{choose}\,(\mathit{Only}\ a) \ = \ \mathit{return}\ a
\end{aligned}
$$

Now that the decision is made earlier, *choose* simply returns the only element that is left. The routine *unsafePerformIO*, or its equivalent, is not part of the standard Haskell library but is supported by most implementations. As the name suggests, it forces the *IO* action given. At each set union, a random boolean r is taken, by which *union* picks one of its two arguments. Functions *prefixes*, *suffixes* and *segments*, if redefined using these new set operations, *do* pass around only one element of the set and are potentially faster. Conceptually, however, we can pretend that we are still passing the entire set around, except for there is no way to read its contents until *choose* finally picks one element in it, or, *collapses* the non-determinism. The use of *IO* monad preserves the program transformation laws just like before.

This way of dealing with non-determinism has been seen before in [8], as an attempt to add exception handling to Haskell. What could not be determined was what exception to raise. Conceptually, an entire set of exceptions are raised, while there is no way to know which one it is until we catch the exceptions using a primitive returning a value wrapped in *IO*. In practice, of course, only one exception is actually raised. Which one it is depends on the order of evaluation.

3 Simulating quantum computing

Now we come to quantum computing. In section 3.1 we will talk about how a quantum bit and a quantum register can be simulated in Haskell and in section 3.2 some operators on them. We will then present, as examples, two quantum algorithms in section 3.3 and 3.4.

3.1 Qubits and quregs

A *qubit* is the quantum analogy of a bit and a *qureg* that of a multi-bits register. While a bit is either true or false, a qubit may be found to be in either states with certain probabilities. Following the *ket* notation of Dirac [3], a bit yielding value 0 is denoted by $|0\rangle$ and a bit yielding 1 by $|1\rangle$. The state of a qubit ψ is linear superposition of classical states, written as a wave function

$$|\psi\rangle \quad = \quad a|0\rangle + b|1\rangle$$

where a and b, called *amplitudes*, are complex numbers satisfying $|a|^2 + |b|^2 = 1$. Intuitively, we say that $|a|^2$ and $|b|^2$ denote the probability that ψ is observed to have value 0 and 1, respectively. In this paper, it suffices to use possibly negative real numbers in place of complex numbers. A qubit can thus naturally be represented by two real numbers.

One might expect that a two-bits qureg can be represented by two qubits (or four real numbers), a three-bits qureg by three qubits (or six real numbers). However, an important quantum physical phenomena called *entanglement* shows that such a representation does not give us the full power of quantum computing. The four (eight) possible values of a two-bits (three-bits) qureg have to be considered separately. Displayed below is a two-bits qureg.

$$|\psi\rangle \quad = \quad a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$$

In general, an n-bit is represented by a vector of 2^n real numbers, sum of their squares being 1. Simulating a qureg this way incurs an exponential overhead, a common phenomena when simulating quantum mechanics on a classical computer. It is not yet known whether this exponential overhead is inevitable. Look at it from the other side, it shows that a qureg has an exponentially larger capacity to represent information than a classical register, a fundamental reason why quantum computers are potentially powerful.

We can represent a qureg by a vector of real numbers, corresponding to the amplitudes.

$$\mathbf{type}\ \mathit{QuReg} \quad = \quad [\mathit{Float}]$$

Say, $[0 \cdot 5, 0 \cdot 5, 0 \cdot 5, 0 \cdot 5]$ represents a two-bit qureg with equal probability, $0 \cdot 25$, for all the four values. Simple vector operations like addition and scalar are also defined.

$$\begin{array}{lcl}
\mathit{plus} & :: & \mathit{QuReg} \rightarrow \mathit{QuReg} \rightarrow \mathit{QuReg} \\
\mathit{plus} & = & \mathit{zipWith}\,(+) \\
\\
\mathit{scalar} & :: & \mathit{Float} \rightarrow \mathit{QuReg} \rightarrow \mathit{QuReg} \\
\mathit{scalar}\ a & = & \mathit{map}\,(a\times)
\end{array}$$

In the derivations later, where no confusion occurs, we will simply use the symbols $(+)$ and $(\times)$ for clarity.

The readers may have noticed the connection with the previous section. While a set can be represented by a list enumerating all its elements, a probabilistic set can be represented by a list of elements, each paired with the probability that it is in the set. If the domain of elements is finite and totally ordered, the elements can be omitted, leaving in the list only the probabilities. The representation for quregs here is a further extension, allowing the probabilities to be negative (or complex) numbers.

The function $\delta\, n$ embeds a classical state $|i\rangle$ into an n-bit qureg.

$$
\begin{array}{lcl}
\delta & :: & \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{QuReg} \\
\delta\, n\, i & = & \mathit{repeatN}\; i\, 0 +\!\!+ [1] +\!\!+ \mathit{repeatN}\; (2^n - i - 1)\, 0 \\
\mathit{repeatN}\; n & = & \mathit{take}\; n \cdot \mathit{repeat}
\end{array}
$$

Say, $\delta\, 2\, 2 = [0, 0, 1, 0]$. In this paper we will write a quantum state in the ket notation, say $|10\rangle$, and its vector representation $[0, 0, 1, 0]$ interchangeably.

In classical programming, the joint state of two variables is their cartesian product. Its quantum analogy is tensor product, defined by:

$$
\begin{array}{lcl}
(\otimes) & :: & \mathit{QuReg} \rightarrow \mathit{QuReg} \rightarrow \mathit{QuReg} \\
q \otimes r & = & [a \times b \mid a \leftarrow q, b \leftarrow r]
\end{array}
$$

Say, the probability that a two-bits qureg has value 01 is the probability that the first bit has value 0 times the probability the second bit has value 1.

$$
[a, b] \otimes [c, d] \quad = \quad [a \times c, a \times d, b \times c, b \times d]
$$

The state $|01\rangle$ is equivalent to $|0\rangle \otimes |1\rangle$. Indeed, $[1, 0] \otimes [0, 1] = [0, 1, 0, 0]$.

3.2 Quantum operators

Since an n-bit qureg can spontaneously be in 2^n states, one would imagine that if we apply a function f to it, we compute the value of f on 2^n inputs in just one step. An interesting dilemma, however, is that although we have so much information in a qureg, there is no way to read all of it! In a simulation, we do have at hand the amplitudes in the vector all the time and are free to make as many observations as we want. In real quantum computing, however, an observation collapses the quantum state. It is also called *finalisation*. Its most simple form, *diagonal* finalisation, is to toss a dice, biased according to the probabilities represented by the amplitudes of the qureg, and return the corresponding value. More complicated finalisations can be transformed in terms of a diagonal finalisation after some quantum operators. We represent a diagonal finalisation by a function *finalise*, which yields an IO action:

$$
\mathit{finalise} :: \mathit{QuReg} \rightarrow \mathit{IO}\; \mathit{Int}
$$

It will be used in pretty much the same way as the function *choose* in section 2.

Although we cannot observe the values of f on all the inputs, we can still, via some alternative ways to apply f to the qureg, extract some properties of f of interest to us. One possibility is as below:

$$
\begin{array}{lll}
\mathit{trans} & :: & (\mathit{Int} \to \mathit{Int}) \to \mathit{QuReg} \to \mathit{QuReg} \\
\mathit{trans}\, f\; q & = & \mathit{zipWith}\,(\times)\; \mathit{signs}\; q \\
\quad \textbf{where} & & \mathit{signs} \;=\; \mathit{map}\,(\lambda x \to (-1)^{(f\; x)})\,[0 \cdot\cdot 2^n - 1] \\
 & & n \;=\; \log_2(\mathit{length}\; q)
\end{array}
$$

Let f be a function from $\{0 \cdot\cdot 2^n - 1\}$ to an integer, $\mathit{trans}\, f$ is a quantum operator which flips the signs of the amplitudes if f yields an odd value. For example,

$$\mathit{trans}\; \mathit{id}\,[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}] \quad = \quad [\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}]$$

Some quantum operators are defined by tensor product lifted to functions. Given a quantum operator h and k on m and n-bits quregs respectively, $h \otimes k$ is a quantum operator on $(m+n)$-bits quregs, defined by

$$(h \otimes k)(q \otimes r) \quad = \quad (h\; q) \otimes (k\; r)$$

However, as we mentioned above, an $(m+n)$-bits qureg is represented by a vector of 2^{m+n} real numbers, and, due to entanglement, it is not always possible to decompose it back to two quregs of sizes m and n. How do we compute $h \otimes k$, then? The answer lies in linearity of quantum operators, that is, they distribute into summation. We take $m = n = 1$ in the example below. The tensor product $h \otimes k$ can be distributed into the sums of a given two-bits qureg.

$$
\begin{array}{ll}
 & (h \otimes k)\,(a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle) \\
= & \quad \{\text{linearity}\} \\
 & a \times (h \otimes k)\,|00\rangle + b \times (h \otimes k)\,|01\rangle \\
 & + c \times (h \otimes k)\,|10\rangle + d \times (h \otimes k)\,|11\rangle \\
= & \quad \{\text{tensor product, } |xy\rangle = |x\rangle \otimes |y\rangle\} \\
 & a \times (h\,|0\rangle \otimes k\,|0\rangle) + b \times (h\,|0\rangle \otimes k\,|1\rangle) \\
 & + c \times (h\,|1\rangle \otimes k\,|0\rangle) + d \times (h\,|1\rangle \otimes k\,|1\rangle)
\end{array}
$$

Now each of the $h\,|x\rangle \otimes k\,|y\rangle$ are dealing with concrete states and can be computed accordingly. They each results in a 2-bit qureg. These quregs are then summed up with weights given by the amplitudes. More generally, tensor products of functions can be defined by:

$$
\begin{array}{lll}
(h \otimes k)\; q & = & \mathit{foldr1}\; \mathit{plus}\,(\mathit{zipWith}\; \mathit{scalar}\; q\; \mathit{ts}) \\
\quad \textbf{where} & & \mathit{ts} \;=\; [p \otimes r \mid p \leftarrow \mathit{hq}, r \leftarrow \mathit{kq}] \\
 & & \mathit{hq} \;=\; \mathit{map}\,(h \cdot \delta\,(n-1))\,[0 \cdot\cdot 2^{n-1} - 1] \\
 & & \mathit{kq} \;=\; \mathit{map}\,(k \cdot \delta\, 1)\,[0, 1] \\
 & & n \;=\; \log_2(\mathit{length}\; q)
\end{array}
$$

For simplicity we consider only the case when k is an operator on qubits, which is sufficient for the examples this paper.

As an example of quantum operator defined in terms of tensor products, the Hadamard function defined on qubits is defined by:

$$H_1(a|0\rangle + b|1\rangle) \quad = \quad \frac{1}{\sqrt{2}}(a+b)|0\rangle + \frac{1}{\sqrt{2}}(a-b)|1\rangle$$

The n-bits version, on the other hand, is defined by repeating tensor product $(H_1 \otimes H_1 \ldots \otimes H_1)$ n times. Its Haskell translation is immediate.

$$\begin{array}{lll} \mathit{hadamard}\; q & = & \mathit{had}\;(\mathit{length}\; q)\; q \\ \quad \textbf{where} & \begin{array}{lll} h\,[a,b] & = & [\frac{1}{\sqrt{2}} \times (a+b), \frac{1}{\sqrt{2}} \times (a-b)] \\ \mathit{had}\; 1 & = & h \\ \mathit{had}\;(n+1) & = & \mathit{had}\; n \otimes h \end{array} & \end{array}$$

We will talk a bit more about the Hadamard function. To get a feel of it, we derive its closed form for two-bits quregs. Note that $h\,|0\rangle = h\,[1,0] = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ and $h\,|0\rangle = h\,[0,1] = [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]$.

$$\begin{array}{ll} & \mathit{hadamard}\;(a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle) \\ = & \quad \{\text{linearity}\} \\ & a \times \mathit{hadamard}\;|00\rangle + b \times \mathit{hadamard}\;|01\rangle \\ & + c \times \mathit{hadamard}\;|10\rangle + d \times \mathit{hadamard}\;|11\rangle \\ = & \quad \{\text{tensor product}\} \\ & a \times (h\,|0\rangle \otimes h\,|0\rangle) + b \times (h\,|0\rangle \otimes h\,|1\rangle) \\ & + c \times (h\,|1\rangle \otimes h\,|0\rangle) + d \times (h\,|1\rangle \otimes h\,|1\rangle) \\ = & \quad \{\text{definition of } h\} \\ & a \times ([\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}] \otimes [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]) + b \times ([\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}] \otimes [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]) \\ & + c \times ([\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}] \otimes [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]) + d \times ([\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}] \otimes [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]) \\ = & \quad \{\text{tensor product}\} \\ & a \times [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}] + b \times [\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}] \\ & + c \times [\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}] + d \times [\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}] \\ = & \quad \{\text{arithmetics}\} \\ & [\frac{1}{2}(a+b+c+d), \frac{1}{2}(a-b+c-d), \\ & \;\frac{1}{2}(a+b-c-d), \frac{1}{2}(a-b-c+d)] \end{array}$$

Some observations. First, applying the Hadamard function to $|00\rangle$, we get $\mathit{hadamard}\,|00\rangle = \mathit{hadamard}\,[1,0,0,0] = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$. It helps to prepare a qureg with equal probabilities for each values. Indeed, one of the many uses of Hadamard function is to initialise quregs. The following function *ini* prepares an n-bit qureg initialised to equal probabilities for all values.

$$\begin{array}{lll} \mathit{ini} & :: & \mathit{Int} \to \mathit{QuReg} \\ \mathit{ini}\; n & = & \mathit{hadamard}\;(\delta\; n\; 0) \end{array}$$

Second, the Hadamard function sums up the amplitudes to state $|00\rangle$. For all other states, it gives an equal number of positive and negative signs to each of the amplitudes before the summation. This behaviour will turn out to be useful later.

3.3 The Deutsch-Jozsa classification

For quite a while, although people knew about the potential of quantum computers, there are no algorithms to exploit their computing power, until Deutsch and Jozsa demonstrated the first widely recognised quantum algorithm. It was first presented by Deutsch[1], dealing with qubits, but did not attract much attention until later extended to n-bits case by Deutsch and Jozsa[2].

Being one of the earliest quantum algorithms, it deals with a rather artificial problem. Given is a function $f :: \{0 \cdot \cdot 2^n - 1\} \to \{0, 1\}$ for some $n \geq 1$. We know nothing about the function except for that it is either constant (i.e. $f\ a$ always yields the same value for all a) or balanced (i.e. the sets $\{\, a \mid f(a) = 0\}$ and $\{\, a \mid f(a) = 1\}$ has the same size). How do we find out which case it is? On classical computers, it may take as many as $2^{n-1} + 1$ applications of f before we can safely make a conclusion. Can we do that with only one (quantum) application?

Their algorithm simply reads:

$$
\begin{array}{lcl}
\mathit{dj} & :: & \mathit{Int} \to (\mathit{Int} \to \mathit{Int}) \to \mathit{IO}\ \mathit{Int} \\
\mathit{dj}\ n\, f & = & (\mathit{finalise} \cdot \mathit{hadamard} \cdot \mathit{trans}\, f \cdot \mathit{ini})\ n
\end{array}
$$

As mentioned in the previous section, the function *ini* initialises the qureg to equal probability for all the 2^n values. For $n = 2$, for instance, $\mathit{ini}\, 2$ yields $[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$. The lifted function $\mathit{trans}\, f$ is then applied to this initialised qureg. In the case that f is a constant function always yielding zero, none of the signs get changed. In other words, $\mathit{trans}\, (\mathit{const}\, 0)\, (\mathit{ini}\, 2) = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$. The last hadamard function then sums all the probabilities to state $|00\rangle$, while the probabilities for all other states get cancelled out, yielding the qureg $[1, 0, 0, 0]$.

Similarly, when f is a constant function always returning 1, all the signs will be changed. The probability of state $|00\rangle$ will be -1 and those of other states still got cancelled out. The resulting qureg would be $[-1, 0, 0, 0]$. In both cases, the *finalise* step will give us 0 with 100% probability.

When f is balanced, however, we will have an equal number of $\frac{1}{2}$s and $-\frac{1}{2}$s after the $\mathit{trans}\, f$ step. So it becomes $|00\rangle$ to be cancelled out. Finalisation may yield any value but 0.

In summary, if the Deutsch-Jozsa algorithm yields 0, f is a constant function. If we get any other value, we may conclude that f is balanced.

3.4 Grover's point search algorithm

The Deutsch-Jozsa algorithm always delivers the desired result. The more typical quantum algorithm, like the one in this section described by Grover [5], computes the result within a margin of error. The problem is: given an array of 2^n bits containing a single 1, represented by a function $f :: \{0 \cdot \cdot 2^n - 1\} \to \{0, 1\}$, find the index where the 1 is.

Grover's algorithm uses a loop after the initialisation. The initialisation prepares a qureg with the same positive amplitudes for all possible values. In each iteration of the loop, first the function *trans f* is applied to the prepared qureg. Only the amplitude corresponding to the only entry in f yielding 1 will have its sign changed. Call it q. The function *diffusion* is applied to q. Its local identifier *avg* stands for the average of the amplitudes in q. It creates a new qureg whose amplitudes result from subtracting from *avg* each amplitudes in q. The effect is that all the amplitudes are inverted about the average. The entry for which f yields 1 gets bigger because its sign was just changed, making it furthest from the average. The difference between this entry and others is thus amplified. The more iterations run, the more possible that this entry will be chosen in the end. The algorithm reads:

$$
\begin{array}{lcl}
\mathit{grover} & :: & \mathit{Int} \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{Int} \rightarrow \mathit{IO\ Int} \\
\mathit{grover}\ n\ f\ i & = & (\mathit{finalise} \cdot \mathit{loop}\ i\ (\mathit{diffusion} \cdot \mathit{trans}\ f) \cdot \mathit{ini})\ n \\
\quad \mathbf{where} & & \mathit{loop}\ i\ f = \mathit{head} \cdot \mathit{drop}\ i \cdot \mathit{iterate}\ f \\
 & & \mathit{diffusion}\ q = \mathit{map}\ (\lambda x \rightarrow 2 \times \mathit{avg} - x)\ q \\
 & & \quad \mathbf{where}\ \ \mathit{avg} = \frac{1}{2^n} \times \mathit{sum}\ q
\end{array}
$$

The important choice of i, which determines the number of iterations performed, will not be discussed here. A number proportional to $\sqrt{2^n}$ will be sufficient, while traditional searching algorithm needs at best $O(2^n)$ time.

4 Quregs as monads

Let us look again at the distributivity law in section 3.2. Given a quantum operator f should satisfy that given any quantum state q:

$$
\begin{array}{ll}
 & f\ q \\
= & \quad \{\text{assume } q = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle\} \\
 & f\ (a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle) \\
= & \quad \{\text{linearity}\} \\
 & a \times f\ |00\rangle + b \times f\ |01\rangle \\
 & + c \times f\ |10\rangle + d \times f\ |11\rangle
\end{array}
$$

In fact, all n-bit quantum operators can be written as $n \times n$ unitary matrices and their application is interpreted as matrix multiplication. Applying f to $|00\rangle$, $|01\rangle$...etc is actually recovering the columns of the matrix.

If we abstract the above expression over f and q, that is, define an *apply* function for quantum operators, what would its type be? Clearly, q has type *QuReg*. The operator f should have type $\mathit{QuReg} \rightarrow \mathit{QuReg}$. However, in the expression above, f is only applied to classical states, i.e. ordinary values. So we may simplify its type to $\mathit{Int} \rightarrow \mathit{QuReg}$. The result of the application should, of course, be a *QuReg*. The *apply* functions should thus have type

$$\mathit{apply} :: (\mathit{Int} \rightarrow \mathit{QuReg}) \rightarrow \mathit{QuReg} \rightarrow \mathit{QuReg}$$

or, if we reverse the position of f and q:

$$\mathit{apply} :: \mathit{QuReg} \to (\mathit{Int} \to \mathit{QuReg}) \to \mathit{QuReg}$$

But, that resembles the type of the monadic $(\gg\!=)$ operator! That inspires the idea that we can write quantum programs in a monadic style.

True, a *qureg* cannot be a monad, because it does not take a type parameter like the list type constructor[1]. Never the less, you can still define a $(\gg\!=)$ and *return* operator for quregs. It is just that you cannot use **do**-notation, which is just a convenient syntax sugar anyway. The $(\gg\!=)$ operator can be defined by:

$$\begin{array}{lcl}
(\gg\!=) & :: & \mathit{QuReg} \to (\mathit{Int} \to \mathit{QuReg}) \to \mathit{QuReg} \\
q \gg\!= f & = & \mathit{foldr1}\ \mathit{plus}\ (\mathit{zipWith}\ \mathit{scalar}\ q\ (\mathit{map}\ f\ [0 \cdot \cdot 2^n - 1])) \\
\quad \textbf{where}\ \ n & = & \log_2(\mathit{length}\ q)
\end{array}$$

while the $\mathit{return} :: \mathit{Int} \to \mathit{QuReg}$ operator is simply defined by $\mathit{return} = \delta$.

Assume that the **do**-notation is extended to quregs. The monadic version of the Deutsch-Jozsa algorithm can be simply written as

$$\begin{array}{lcll}
\mathit{dj}\ n\ f & = & \mathit{finalise}\ (\textbf{do} & x \leftarrow \mathit{ini}\ n \\
 & & & y \leftarrow \mathit{trans}\ n\ f\ x \\
 & & & \mathit{hadamard}\ n\ y)
\end{array}$$

Like with the list monad, the **do**-notation encourages the programmer to think in terms of individual values. In the definition above, the identifiers x and y has type *Int*. The programmer pretends that a concrete value x is extracted from the initialised qureg, fed to function $\mathit{trans}\, f$, resulting in another concrete integer y, and fed to the Hadamard function. We all know very well, however, that this feeling of value-based programming is only a metaphor. What actually happens in the monad is that all possible values are fed to $\mathit{trans}\, f$ and *hadamard* and then summed up with different weights.

We still need to make corresponding changes to the basic quantum operators. Now that the job of feeding different values to quantum operators is dealt with by $(\gg\!=)$, tensor product on functions has a simpler definition.

$$(h \otimes k)\ x \quad = \quad h\ (x \text{ `}\mathit{div}\text{` } 2) \otimes k\ (x \text{ `}\mathit{mod}\text{` } 2)$$

Hadamard function now takes concrete values, and an extra parameter specifying the size of the qureg. The biggest change is in the sub-function h.

$$\begin{array}{lcl}
\mathit{hadamard}\ 1 & = & h \\
\mathit{hadamard}\ (n+1) & = & \mathit{had}\ n \otimes h \\
\quad \textbf{where}\ \ h\ 0 & = & [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}] \\
\qquad\qquad h\ 1 & = & [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]
\end{array}$$

[1] Some other representations for quregs can be generalised to contain a type parameter, but some context restrictions on the parameter will need to be satisfied. So it cannot be a monad in the Haskell sense anyway.

The definition of h is basically just writing down the matrix corresponding to the quantum operator. The definition of $trans\, f$ is similar to δ, except for it can be either a 1 or a -1 in the middle.

$$trans\; n\; f\; x \quad = \quad repeatN\; i\; 0 \mathbin{+\!\!+} [(-1)^{(f\; x)}] \mathbin{+\!\!+} repeatN\; (2^n - i - 1)\; 0$$

Similarly, the Grover's searching algorithm can be written in monadic style as below.

$$\begin{array}{llll}
grover\; n\; f\; i & = & \multicolumn{2}{l}{finalise\; (\mathbf{do}\;\; x \leftarrow ini\; n} \\
 & & \multicolumn{2}{l}{\qquad\qquad\quad loop\; i\; x)} \\
\quad \mathbf{where} & loop\; 0\; x & = & return\; x \\
 & loop\; (i+1)\; x & = & \mathbf{do}\;\; y \leftarrow trans\; n\; f\; x \\
 & & & \qquad z \leftarrow diffusion\; n\; y \\
 & & & \qquad loop\; i\; z
\end{array}$$

It is made clear that the algorithm is a loop after the initialisation, while the loop consists of two steps. First the current state is fed to $trans\, f$, then to $amplify\; n$. Amplify is again defined in a way similar to δ.

$$diffusion\; n\; i \quad = \quad repeatN\; i\; \tfrac{1}{2^n} \mathbin{+\!\!+} [2 \times \tfrac{1}{2^n} - 1] \mathbin{+\!\!+} repeatN\; (2^n - i - 1)\; \tfrac{1}{2^n}$$

As a comparision, in the quantum computing texts, the matrix for the *diffusion* step is usually defined by $D_{ij} = \frac{2}{2^n}$ if $i \neq j$ and $D_{ii} = \frac{2}{2^n} - 1$. We are really writing the same matrix here.

This monadic style gives the programmer a feeling of imperative programming. The reader is encouraged to compare it with [11], where an extended version of Dijkstra's guarded command language was proposed as the programming language of choice for quantum programming with quregs the basic unit to operate on, or with [6], where a C-like language manipulating the values was used but the amplitudes are entirely implicit and must be discovered though all traces of the program. Whether the monadic style is appropriate, easier for the programmers, and whether it gives new insight into quantum programming, remains to be seen.

Another interesting phenomena is that it is not possible to apply the same optimisation as in section 2. Every intermediate value matters, and once the non-determinism is collapsed pre-maturely, the computation cannot carry on in the same way. That is another reason why simulating quantum computation on a classical device is bound to be expensive.

5 Conclusion

We have demonstrated the relationship between quantum programming and the usual way encoding non-determinism in representations of sets. Starting from the list representation, we showed how non-determinism can be encoded in sets in a functional language. Demonic non-determinism is encapsulated in an alternative representation of sets. Accessing the contents of the set yields a value wrapped in an *IO* monad, thus non-deterministic choice is allowed to be unsafely performed at an earlier stage. This is a continuation of the theme in [7, 8].

A qureg is represented by a special kind of probabilistic set in which negative probabilities are allowed. Quantum programming is then seen as a special kind of non-deterministic programming where the non-determinism is not allowed to collapse pre-maturely as in the previous case.

A monadic style of quantum programming is also proposed. Programs are written in an imperative style but the programmer is encouraged to think in terms of values rather than quregs. Whether it gives new insight into quantum programming, however, remains to be seen.

Acknowledgement

This work was inspired by Tony Hoare during Mu's summer internship job in Microsoft Research Ltd., Cambridge. Much of the facts about quantum computing presented in this paper is based on Paolo Zuliani's D.Phil thesis. Thanks is due to Paolo Zuliani for valuable discussions.

References

[1] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society, Series A*, volume 400, pages 97–117, 1985.

[2] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computer. In *Proceedings of the Royal Society, Series A*, volume 439, pages 553–558, 1992.

[3] P. A. M. Dirac. *The Principles of Quantum Mechanics, 4th edition*. Oxford University Press, 1958.

[4] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[5] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 212–219, 1996.

[6] L. K. Grover. Searching with quantum computers. *Dr. Dobb's Journal*, April 2001.

[7] J. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic programs. *Functional Programming Glasgow 1989*, pages 308–328, 1990.

[8] S. P. Jones, A. Reid, C. A. R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'99)*, Atlanta, 1999.

[9] J. W. Sanders and P. Zuliani. Quantum programming . In R. C. Backhouse and J. N. F. d. Oliveira, editors, *Mathematics of Program Construction 2000*, number 1837 in Lecture Notes in Computer Science, pages 80–99. Springer-Verlag, 2000.

[10] S. Seres, M. Spivey, and C. A. R. Hoare. Algebra of logic programming. In *Proceedings of ICLP'99*, Las Cruces, USA, 1999.

[11] P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University, 2001.