On the Automation of Encoding Processes in the Quantum IO Monad

James Barratt

Rule Financial Ltd

`james.barratt@rulefinancial.co.uk`

It is now clear that the use of resilient encoding schemes will be required for any quantum computing device to be realised. However, quantum programmers of the future will not wish to be tied up in the particulars of such encoding schemes. Quantum programming languages and libraries are already being developed, one of which is the Quantum IO Monad. QIO, as it is often abbreviated to, provides an interface to define and simulate quantum computations via a library of functions written in Haskell, a purely functional programming language. A solution is presented that takes an arbitrary QIO program and returns an equivalent program incorporating some specified quantum error correction techniques.

1   Introduction

Although theoretically possible, it is not yet clear whether building a sizeable quantum computer is feasible. The main obstacle is to find ways of dealing with decoherence and other quantum noise. When a quantum system interacts with its environment quantum information is leaked out and it begins to act in a probabilistic manner. In effect, parts of the system are being measured by the environment. Extraneous operations may also appear randomly during the computation, which corrupts the state and ultimately the result. Although research is focused on building quantum computers that are less likely to interact with their environment, it is impossible to completely isolate a system and therefore decoherence and quantum noise are inevitable. However, a number of techniques have been developed by the quantum software community that promise to reduce their effects on the system even further.

Quantum error correction encapsulates software related techniques for reducing the impact of decoherence and other quantum noise. Some of the techniques that have so far been developed are inspired by coding theory and techniques already used within classical error correction. The basic principle is to encode information in such a way that the existence of errors can be detected and the nature of those errors identified. It is then possible to apply recovery operators before the information is decoded into its original form. However, checking for errors in a quantum computer is more problematic and there are new, entirely non classical, errors to contend with.

2   The Quantum IO Monad

The Quantum IO Monad is a library of functions written in Haskell that provides an interface to define and simulate quantum computations[1]. It is used within the current work to demonstrate the use of encoding schemes. In functional programming computation is considered to be the application of functions as opposed to the manipulation of some global state. The following is a simple example of a QIO program.

example :: *QIO Bool*
example = **do**
    q1 ← *mkQbit False*
    applyU $ *unot q1*
    b ← *measQbit q1*
    return b

This function is written using **do** notation, which resembles a more imperative style, but is in fact functional underneath. Such seemingly impure computations may be defined thanks to the use of monads that are built upon some polymorphic type. The function *mkQbit* creates a new qubit initialised in the state given by the boolean argument, with False and True representing the base states $|0\rangle$ and $|1\rangle$ respectively. Unitary operations are then applied using *applyU*, the quantum NOT gate, or the Pauli X gate, in this case. The *unot* operation is defined as a *Rotation*, as are all single qubit operations. Other unitaries defined in QIO include *swap*, *cond,* for controlled operations, and *ulet*, which is used to declare and encapsulate the use of ancillary qubits. Finally, qubits are measured using *measQbit* and a boolean value is returned randomly depending on the superposition of the qubit at the time. The qubit itself is also collapsed into the base state corresponding to this boolean value.

3   Implementing Quantum Error Correction in QIO

Although it may be some time before real quantum computers are realised, work is already being carried out by computer scientists to develop good quantum programming languages. The development of the Quantum IO Monad is an example of such work. Although reliable error correction techniques will be crucial, in order to allow programmers to concentrate on the matter at hand some form of automated conversion into equivalent but error resilient programs would be extremely advantageous. The following describes an attempt to do just this within the Quantum IO Monad.

3.1   Introducing an Encoded Qubit Type

The first step towards being able to convert a QIO program into an equivalent that incorporates quantum error correction is to declare a class called *EnQubit*, instances of which will represent encoded qubits. By defining such a class the main program can be written in a generic fashion without the need to understand how encoded qubits are represented[2]. As long as the instances of this class provide definitions of the functions declared here there will be no need to alter the main program when switching schemes, since the appropriate functions will be called at run time.

Each *EnQbit* is thought to have a "parent" qubit, the one from which the encoded state is produced. The function *getQbit* returns this particular qubit and is required by *measEnQbit*, defined below.

class *EnQbit a* **where**
    mkEnQbit :: *Bool* → *QIO a*
    getQbit :: *a* → *Qbit*
    measEnQbit :: *a* → *QIO Bool*
    measEnQbit eq = **do**
        applyU $ *decode eq*
        b ← *measQbit* (*getQbit eq*)
        applyU $ *encode eq*
        return b

The class also provides flexibility in the error correcting codes that may be employed. By providing different *encode* and *correct* functions a different encoding scheme may be used with the same representation. The function *decode* is in fact defined here as the inverse of *encode*, since it is unitary.

$$encode :: a \rightarrow U$$
$$decode :: a \rightarrow U$$
$$decode\ eq = urev\ \$\ encode\ eq$$
$$correct :: a \rightarrow U$$

The following functions define *EnQbit* versions of the standard unitary operators of QIO. These are implemented to simply decode the given *EnQbit* and call the appropriate standard unitary, passing the "parent" qubit. Before returning, the qubits are encoded again. This approach only provides protection in between the application of operations, while qubits are perhaps being stored or transmitted, as suggested by Shor[3]. Being able to perform operations on the actual encoded qubits themselves would make decoding unnecessary until the end of the computation and thus provide greater protection. This is possible for certain operations in a bitwise fashion, those with the property of transversality, but it does depend on the code being used[5]. So although these functions are defined here as a standard, they may be overridden by instances of the class, if such encoded operations are available.

$$rotEnQbit :: a \rightarrow Rotation \rightarrow U$$
$$swapEnQbit :: a \rightarrow a \rightarrow U$$
$$condEnQbit :: a \rightarrow (Bool \rightarrow U) \rightarrow U$$
$$uletEnQbit :: Bool \rightarrow (Qbit \rightarrow a \rightarrow U) \rightarrow U$$

The *uletEnQbit* function is not defined here but simply declared as this depends entirely on the number of qubits being used in the encoding, to determine how many ancillas to create.

In terms of encoded qubit representations, the following type could be used for the three qubit bit flip code, with *EnQbit* functions defined as appropriate.

newtype *EQ3* = *EQ3* (*Qbit*, *Qbit*, *Qbit*)

instance *EnQbit EQ3* **where**
   mkEnQbit = *mkEQ3*
   getQbit = *fstEQ3*
   encode = *encode3*
   correct = *correct3*
   uletEnQbit = *uletEQ3*

However, this representation could also be used for the three qubit phase flip code by simply altering the implementations of *encode* and *correct*. An instance has also been implemented for Steane's code[4] using a 7-tuple of qubits and appropriate encoding and correction procedures.

3.2 Converting a QIO program

In order to manipulate an existing QIO program, the function *ecQIO'* pattern matches on it and inserts new function calls depending on the construct found, exploiting the monadic structure of the *QIO* type. It is defined recursively with the base case being *QReturn*, signifying the end of the computation.

$$ecQIO' :: EnQbit\ a \Rightarrow QIO\ t \rightarrow [a] \rightarrow QIO\ t$$
$$ecQIO'\ (QReturn\ a)\ \_ = QReturn\ a$$

When a qubit is being created, measured or manipulated the appropriate *EnQbit* function is substituted in to replace the original. As such we obtain a program that consists of extra operations and qubits, together encapsulating the logical state of the single qubit referred to. Whatsmore, a list is passed on in

the recursive call and threaded through the whole computation as a register of the encoded qubits. In the case of *MkQbit* the newly created *EnQbit* is appended to the list before passing it on. Here *MkQbit* is one of the constructors of the *QIO* type, which the convenience function *mkQbit* seen earlier relies upon.

> *ecQIO′* (*MkQbit b f*) *eqs* = **do**
>   *eq* ← *mkEnQbit b*
>   *ecQIO′* (*f* (*getQbit eq*)) (*eq* : *eqs*)

The interesting case in *ecQIO'* is the *ApplyU* constructor, where it is necessary to pattern match further on the *U* data type representing unitary operations. This takes place in the function *extendU*, where we plan to extend the operation's influence to include the rest of the qubits in the encoded qubit.

> *ecQIO′* (*ApplyU u f*) *eqs* = **do**
>   *applyU* $ *extendU u eqs*
>   *ecQIO′ f eqs*

By pattern matching on the *U* data type being applied, *extendU* calls the appropriate function implementing the *EnQbit* version of the unitary. The case of a rotation is given here.

> *extendU* :: *EnQbit a* ⇒ *U* → [*a*] → *U*
> *extendU* (*Rot q r u*) *eqs* =
>   **let** *eq* = (*fromJust* (*getEnQbit q eqs*)) **in**
>   *rotEnQbit eq r* ‘*mappend*‘
>   *correctAll eqs* ‘*mappend*‘
>   *extendU u eqs*

Since a standard QIO program performs operations on individual qubits, a mechanism for identifying their associated *EnQbit* is needed when faced with measurement and operator constructs. A function *getEnQbit* performs this task by making a comparison between the qubit given and the "parent" qubit of each *EnQbit* in our list.

The function *correctAll* is called upon, and the returned unitary appended to the end of the rotation, to perform the error correction procedure on all encoded qubits created so far, which should take place periodically in order to prevent a build up of errors that is unmanageable for the code. If this were to happen the decoding procedure would yield an incorrect value.

4 Conclusions

In terms of providing error correction to QIO programs, the components that are responsible for converting a program appear to fulfill their task of inserting new function calls at appropriate points. Flexibility is also provided on the implementation of codes used, as the same converting functions can work with any coding scheme.

Two encoding schemes have been implemented, the first being the three qubit bit flip code, with Steane's code as the second. Unfortunately, it appears that programs encoded with Steane's code are not evaluated efficiently, preventing any real observation of adding Steane's code to even the simplest of programs. As more qubits and/or unitary operations are added the performance degrades very quickly. This is not such an issue for the three qubit bit flip code, which is evaluated fairly quickly even for larger programs using error correction.

The applicability of this kind of encoding automation has been demonstrated however, and if these evaluations were to be taking place on a real quantum computer then these problems would not be present.

Acknowledgements

References

[1] Thorsten Altenkirch & Alexander Green (2010): *The Quantum IO Monad*. In Simon Gay & Ian McKie, editors: *Semantic Techniques in Quantum Computation*, Cambridge University Press, pp. 173–205.

[2] James Barratt (2010): *Simulating Decoherence and Implementing Quantum Error Correction in the Quantum IO Monad*. Master's thesis, University of Nottingham.

[3] Peter W. Shor (1995): *Scheme for reducing decoherence in quantum computer memory*. Phys. Rev. A 52, pp. R2493–R2496, doi:10.1103/PhysRevA.52.R2493.

[4] A. M. Steane (1996): *Error Correcting Codes in Quantum Theory*. Phys. Rev. Lett. 77, pp. 793–797, doi:10.1103/PhysRevLett.77.793.

[5] Wojciech Hubert Zurek & Raymond Laflamme (1996): *Quantum Logical Operations on Encoded Qubits*. Phys. Rev. Lett. 77, pp. 4683–4686, doi:10.1103/PhysRevLett.77.4683.