MODELANDO COMPUTAÇÃO QUÂNTICA EM HASKELL

AMR SABRY

Resumo. O artigo desenvolve um modelo para a computação quântica desde a perspectiva da programação funcional. O model explica as idéias fundamentais da computação quântica em um nível de abstração que é familiar a programadores funcionais. O modelo também ilustra algumas das dificuldades inerentes à interpretação da mecânica quântica e salienta diferenças entre modelos de computação computação quântica e tradicional (funcionais ou não).

1. Introdução

A Computação Quântica evoca fortes conexões com a programação funcional pura: ela inclui uma noção pré-definida de paralelismo (mesmo que essa noção seja qualitativamente diferente da noção encontrada na programação funcional) e está baseada em fundamentos matemáticos (espaços vetoriais, álgebra de matrizes, etc.) que podem ser modelados elegantemente em uma linguagem funcional [15].

É natural, portanto, modelar computação quântica em uma linguagem funcional. Como uma primeira aproximação, a computação quântica pode ser vista como uma extensão da computação probabilística clássica: partindo de uma mônada para computações probabilísticas, pode-se desenvolver um elegante mas rudimentar modelo de computação quântica funcional [21]. Este artigo tenta oferecer um modelo mais completo de computação quântica em Haskell com dois objetivos maiores em mente:

(1) Explicar a computação quântica em um nível de abstração familiar à comunidade de linguagem de programação, aos invés dos modelos usados pelos físicos.
(2) Elicitar as conexões entre computação quântica e programação funcional e avaliar a adequação das abstrações funcionais ao domínio da computação quântica.

O primeiro objetivo é alcançado em um grau razoável. O principal desafio aqui é que qualquer modelo operacional da computação quântica deve, de algum modo, comprometer-se com uma *interpretação da mecânica quântica*, a qual tem sido, e ainda é, assunto de debate entre os físicos. Em particular, nosso modelo deve *implementar* algum mecanismo para o colapso da função de onda inerente à medição.

– *Publicado em*:
Haskell Workshop – Proceedings of the ACM SIGPLAN Workshop on Haskell.
Uppsala, Sweden. August 28 - 28, 2003.
– *Disponível em*:
http://www.cs.indiana.edu/ sabry/research.html
– *Traduzido por*:
Antônio Carlos da Rocha Costa
ESIN/UCPel, PPGC-PGIE/UFRGS
rocha@atlas.ucpel.tche.br
Janeiro/2004.

Nós usamos efeitos colaterais globais, independentemente de isso ter alguma coisa a ver com a "realidade física", ou não.

Em relação ao segundo objetivo, nós argumentamos que, contrariamente a investigações preliminares, as abstrações da programação funcional (tais como realizadas em Haskell) não são tão adequadas assim à computação quântica. Os desacordos são, entretanto, bastante instrutivos. Primeiro, eles explicam alguma coisa da essência da computação quântica, naquilo em que ela difere da programação funcional. De modo muito significativo, ao contrário do que acontece com programas funcionais, o raciocínio sobre sistemas quânticos é não-composicional, o que – nós argumentamos – exige novas abstrações. Segundo, os desacordos também expõe algumas das limitações da linguagem Haskel, quando aplicada a um domínio radicalmente novo.

O resto do artigo é organizado como segue. A seção 2 introduz os ingredientes básicos da computação quântica: estruturas de dados quânticas. A seção 3 aumenta o modelo com operações e funções sobre dados quânticos. A seção 4 trata da questão candente da observação ou medição. A seção 5 propõe um padrão de projeto reminiscente do padrão Fachada [12] e relacionado de perto com as *referências componíveis* de Kagawa [14] para convenientemente manipular componentes de estruturas de dados emaranhadas. A seção 6 ilustra o modelo resultante dando diversos exemplos. Finalmente, a seção 7 discute trabalhos relacionados e conclui o artigo.

Nós tentamos tornar o artigo tão auto-contido como possível, mas naturalmente nós não podemos incluir uma introdução completa à mecânica quântica e aos seus fundamentos matemáticos, nem podemos examinar completamente o campo da computação quântica. Nós convidamos o leitor a consultar artigos introdutórios clássicos [29, 23] para informação básica adicional sobre os conceitos e operações introduzidas neste artigo.

2. Dados Quânticos

Os blocos básicos da computação quântica são os *qubits*, ou bits quânticos. Depois de explicar os qubits como um tipo de dado similar ao tipo clásico *Bool*, nós generalizamos a construção para outros tipos de dados.

2.1. Tipos de Dados Enumerados.

Em Haskell, o tipo de dado booleano e seus construtores são definidos como segue:

```
data Bool = False | True
```

Um valor do tipo *Bool* pode ser ou *False* ou *True*, mas nunca igual a ambos ao mesmo tempo. Em contraste, qubits ou booleanos quânticos, cujo tipo nós denotamos por $QV\ Bool$, são valores da forma geral seguinte:

```
p |False> + q |True>
```

onde p e q são números complexos representando *amplitudes de probabilidade*, cada construtor c é interpretado como um *vetor unitário* $|c\rangle$, e $+$ é a adição de vetores. Tal valor booleano quântico é, de alguma maneira, *False* e *True* ao mesmo tempo e essa *superposição* pode ser explorada em computações quânticas. Por exemplo,

a superposição poderia ser usada para explorar dois caminhos alternativos de uma computação em paralelo. O uso de números complexos para representar as amplitudes de probabilidade significa que as amplitudes de probabilidade tem uma fase e, portanto, podem se reforçar uma à outra ou cancelar uma à outra durante computações intermediárias. Em última instância, entretanto, os únicos observáveis são ainda apenas o *False* e o *True*. A probabilidade de observar *False* ou *True* é proporcional ao quadrado da magnitude de p e q, respectivamente.

Generalizando a partir dos booleanos para tipos arbitrários a e suas versões quânticas $QV\ a$, nós observamos o seguinte:

- Todos os construtores para o tipo a vão ser interpretados como vetores unitários a partir dos quais nós podemos construir valores quânticos. Por conveniência, a lista dos vetores unitários é um operador sobrecarregado, para cada tipo de interesse:

```
class (Eq a, Ord a) => Basis a where
    basis :: [a]
```

  Nós devemos ser capazes de distinguir os vetores unitários uns dos outros, portanto nós exigimos que o tipo a seja um membro da classe Eq. Nós também exigimos que os vetores unitários venham associados com uma order arbitrária, mas fixa (i.é, que o tipo a seja um membro da classe Ord) a fim de usar a biblioteca *FiniteMap*, conforme mostrado abaixo. Aqui estão alguns exemplos simples:

```
data Move = Vertical | Horizontal
data Rotation = CtrClockwise | Clockwise
data Color = Red | Yellow | Blue

instance Basis Bool where
    basis = [False, True]
instance Basis Move where
    basis = [Vertical, Horizontal]
instance Basis Rotation where
    basis = [CtrClockwise, Clockwise]
instance Basis Color where
    basis = [Red, Yellow, Blue]
```

- Dados os vetores unitários do tipo a, valores do tipo $QV\ a$ são mapeamentos que associam cada vetor unitário com uma amplitude de probabilidade. Em muitos artigos sobre computação quântica, a identidade dos vetores unitários e sua ordem são mantidos implícitos e os valores quânticos são representados usando apenas uma seqüência de amplitudes de probabilidade. Apesar de isso parecer conveniente, não é flexível do ponto de variação do número de vetores unitários. Nossa representação dos valores quânticos vai, portanto, consistir de um mapeamento explícito de cada vetor unitário para sua amplitude de probabilidade. Falando abstratamente, esse mapeamento poderia ser realizado usando uma função, mas um protótipo inicial

dessa idéia apresentou uma falta de desempenho muito grande, a ponto de mesmo alguns dos exemplos mais simples não poderem ser simulados. Mesmo em um exemplo de computação quântica muito simples, as funções seriam tipicamente aninhadas em diversos níveis e a redução exponencial da velocidade seria inaceitável. Ao invés disso, nós realizamos o mapeamento usando a biblioteca *FiniteMap* (que assume que cada tipo a é uma instância da classe *Ord*, como nós exigimos):

```
type PA = Complex Double
type QV a = FiniteMap a PA
```

Por convenção, vetores unitários associados com uma amplitude de probabilidade zero vão ser freqüentemente omitidos do mapeamento finito. A função *pr* retorna a amplitude de probabilidade associada com um dado vetor unitário:

```
qv :: Basis a => [(a, PA)] -> QV a qv = listToFM

pr :: Basis a => FiniteMap a PA -> a -> PA pr fm k =
lookupWithDefaultFM fm 0 k
```

Aqui estão alguns exemplos simples:

```
qFalse, qTrue, qFT :: QV Bool
qFalse = unitFM False 1
qTrue = unitFm True 1
qFT = qv [(False, 1/(sqrt 2)), (True, 1/(sqrt 2))]

qUp :: QV Move
qUP :: unitFM Vertical 1
```

O valor *qFalse* é um valor booleano quântico que é sempre *False*; similarmente, o valor *qTrue* é um valor quântico que é sempre *True*. O valor *qFT* é "meio-*False*" e "meio-*True*". O valor *qUp* é um valor quântico que é sempre *Vertical*. O fator de normalização de $1/(sqrt\ 2)$ em *qFT* não é computacionalmente significativo e vai ser sempre omitido.

2.2. **Tipos Infinitos.** Em princípio, é possível estender as idéias da seção anterior para tipos de dados infinitos, tais como os números naturais:

```
instance Basis Integer where
    basis = [0..]
```

Um "bom" valor quântico do tipo *QV Integer* associaria uma amplitude de probabilidade não-nula a apenas uns poucos vetores unitários, ou se ele associasse amplitudes de probabilidade não-zero a todos os vetores unitários, as amplitudes deveriam "evanescer de modo suficientemente rápido" como:

```
qi :: QV Integer
qi = qv [(i, 1/i) | i <- basis, i /= 0]
```

Mas depois de expressar valores quânticos tais como *qi*, nós podemos fazer muito pouco com eles, pelo menos se nós vamos manter a apresentação e o código razoavelmente simples. Como ficará claro nas Seções 3 e 4, nós vamos precisar realizar operações *estritas*, como adição, sobre as amplitudes de probabilidade associadas com todos os vetores unitários. Quando o número de vetores unitários é finito, isso pode ser feito com a primitiva Haskell +; quando o número de vetores unitários é infinito, como no caso de *Integer*, nós necessitamos manipular séries convergentes e integrais, ao invés de adições e somatórios. Nós não lidamos com tipos de dados infinitos no resto deste relatório.

2.3. **Pares.** Dados dois valores quânticos do tipo $QV\ a$ e $QV\ b$, nós podemos construir duas espécies de pares: uma do tipo *(QV a, QV b)* e outra do tipo *QV (a,b)*. A primeira espécie de pares não tem nada de especial: valores quânticos são como qualquer outro valor, no sentido de que eles podem ser armazenados em estruturas de dados. O segundo tipo de par é uma instância de uma noção mais geral de valores *emaranhados* que tem diversas novas propriedades sem correspondentes no caso clássico e que requer, portanto, um estudo cuidadoso.

Primeiro, dada a bases para pares:

```
instance (Basis a, Basis b) => Basis (a, b)
    where basis = [(a,b)|(a,b) <- basis]
```

nós podemos escrever alguns exemplos:

```
p1, p2, p3 :: QV (Bool, Bool)

p1 = qv [((False,False),1), ((False,True),1)]

p2 = qv [((False,False),1), ((True,True),1)]

p3 = qv [((False,False),1),
         ((False,True),1),
         ((True,False),1),
         ((True,True),1)]
```

O primeiro componente do par quântico $p1$ é sempre *False* e o segundo é ou *False* ou *True* com igual probabilidade. Isso pode ser formalizado dizendo que o par é equivalente ao *produto tensorial* dos valores *qFalse* e *qFT*. O produto tensorial é geralmente definido como segue:

```
(&*) :: (Basis a, Basis B) =>
          QV a -> QV b -> QV (a,b)
qa &* qb =
   qv [((a,b), pr qa a * pr qb b)|(a,b) <- basis]
```

O fato de que os dois componentes do par *p1* podem ser separados em dois valores não é uma propriedade geral dos pares quânticos. De fato, os componentes do par *p2* não podem ser separados. A razão intuitiva é simples: cada componente do par *p2* pode ser *False* ou *True* com igual probabilidade, o que sugere que o par poderia ser igual a *qFT &* qFT*. Mas, é claro, esse produto tensorial produz *p3* que é bastante diferente de *p2*. Os componentes de um par como *p2*, que não podem ser separados, estão *emaranhados*.

A situação para pares generaliza para outros tipos de dados estruturados que também podem estar emaranhados. Valores emaranhados são um aspecto fundamental da computação quântica e vão ser revisitados com mais detalhes nas próximas seções. Nós notamos agora, entretanto, que o emaranhamento implica que o raciocínio sobre sistemas quânticos é não-composicional:

> Um aspecto supreendente e não intuitivo do espaço de estados de um sistema quântico de n partículas é que o estado do sistema nem sempre pode ser descrito em termos dos estados das partes que o compõem. [23,p.308]

3. Funções/Operações

Na situação clássica, o único operador unário não-trivial sobre booleanos é a função *not* definida como segue:

```
not False = True
not True  = False
```

A correspondente função sobre valores booleanos quânticos mapeia o valor geral $(p|False\rangle + q|True\rangle)$ para $(q|False\rangle + p|True\rangle)$. Ela pode ser definida como segue:

```
qnot_f :: QV Bool -> QV Bool
qnot_f v = qv [(False, pr v True),
               (True, pr v False)]
```

É fácil calcular que *qnot_f qFalse* tem o valor *qTrue*, e vice-versa. Naturalmente, *qnot_f* também pode ser aplicado a valores mistos como *qFT*.

Por causa da estrutura mais rica dos booleanos quânticos podem-se definir muito mais funções, além do simples *qnot_f*. A função *hadamard_f*, definida abaixo, mapeia um valor geral da forma $(p|False\rangle + q|True\rangle)$ para $((p+q)|False\rangle + (p-q)|True\rangle)$. Essa operação pode ser definida como segue:

```
hadamard_f :: QV Bool -> QV Bool
hadamar_f v =
   let p = pr v False
       q = pr v True
   in qv [(False,p+q),(True,p-q)]
```

Um cálculo simples mostra que *hadamard_f qFalse* tem o valor qFT.

Deve estar claro, neste ponto, que existe um padrão geral para todas as operações sobre dados quânticos. O valor de saída tem uma amplitude de probabilidade associada com cada um dos seus vetores unitários; e cada uma dessas amplitudes pode depender das amplitudes de probabilidade de todos os vetores unitários do valor de entrada. Em outras palavras, uma operação sobre dados quânticos é completamente especificada por uma matriz que especifica como cada amplitude de probabilidade de entrada contribui para cada amplitude de probabilidade de saída. Nós representamos essa matriz por um outro mapeamento finito:

```
data Qop a b = Qop (FiniteMap (a,b) PA)

qop :: (Basis a, Basis B) => [((a,b),PA] -> Qop a b
qop = Qop.listToFM
```

Para aplicar uma operação a um valor quântico, nós multiplicamos a matriz pelo vetor representando o valor:

```
qApp :: (Basis a, Basis b) =>
           Qop a b -> QV a -> QV b

qApp (Qop m) v =
   let bF b = sum [pr m (a,b) * pr v a | a <- basis]
   in qv [(b, bF b) | b <- basis]
```

Por exemplo, as operações *qnot_f* e *hadamard_f*, mencionadas acima, podem ser definidas usando as seguintes matrizes:

```
qnot_op = qop [((False,True),1),
               ((True,False),1)]

hadamard_op = qop [((False,False),1),
                   ((False,True),1),
                   ((True,False),1),
                   ((True,True),-1)]
```

Como um outro exemplo, o seguinte operador translada dos estados de polarização vertical/horizontal de um fóton para os estados de polarização horário/anti-horário [19]:

```
m2r :: Qop Move Rotation
m2r = qop [((Vertical,CtrClockwise),1),
           ((Vertical,Clockwise),1),
           ((Horizontal,CtrClockwise),0 :+ -1),
           ((Horizontal,Clockwise),0 :+ 1)]
```

A notação $a :+ b$ é a sintaxe Haskell para o número complexo $a + ib$.

3.1. **Levantamento.** Para entender mais a natureza das operações quânticas, nós discutimos sumariamente um modo de levantar as funções clássicas para operações sobre valores quânticos. A idéia básica é simples: um vetor unitário de entrada contribui para um vetor unitário de saída se e somente se a função clássica relaciona os correspondentes construtores:

```
opLift :: (Basis a, Basis b) => (a -> b) -> Qop a b
opLift f = qop [((a, fa),1) | a <- basis]
```

Entretanto, isso nem sempre é razoável, uma vez que operações quânticas devem ser *unitárias* (i.é, a operação é inversível e, quando vista como uma matriz, a inversa da operação é apenas a transposta conjugada da matriz). No caso especial em que f é uma função *reversível*, a construção acima funciona e produz o que é chamado um *operador pseudo-clássico*. Por exemplo, a função *not* é reversível e a construção acima produz, na verdade, *qnot_op*.

Outras funções como *and* não são reversíveis: de uma saída *False*, não se pode calcular as duas entradas para a função *and*. Entretanto, uma função não-reversível como *and* pode ser trivialmente feita reversível pelo acréscimo de saídas adicionais, que transferem as entradas:

```
reversibleAnd a b = (a, b, a && b)
```

Em geral, qualquer computação clássica, não importando quão complexa, pode ser feita reversível pelo registro de um número suficiente de seus resultados intermediários. Bennet mostra como uma máquina de Turing universal pode ser simulada por uma máquina de Turing reversível [4]. A idéia também pode ser adaptada a máquinas abstratas como a máquina SECD [17] e otimizada para além do requisito ingênuo de armazenar cada valor intermediário [5]. Computação reversível é, também, um tópico interessante por si mesmo [11, 1].

3.2. **Produzindo Pares Emaranhados.** Operações controladas são o modo mais comum de introduzr emaranhamento em sistemas quânticos. A mais simples dessas operações é a operação não-controlado (*cnot*); *cnot* pega dois valores booleanos quânticos e:

- não faz nada se o primeiro valor (chamado de valor de controle) é *False*, e
- nega o segundo valor (chamado de valor alvo), caso contrário.

Isso parece bastante simples até que nós lembramos que o qubit de controle pode ser simultanemante *False* e *True*. Nesse caso, a operação realiza ambas as ações simultaneamente e o par de qubits resultante é emaranhado. Por exemplo, considere a situação em que o qubit de controle é qFT e o qubit alvo é *qFalse*. Dado que

o qubit de controle é *False* com uma probabilidade não-nula, uma saída possível para a operação é o estado (*False*,*False*). Também, como o qubit de controle é *True* com uma probabilidade não-nula, uma possível saída para a operação é o estado (*True*,*True*). Nenhuma outra saída é possível. Em outras palavras, aplicando a operação *cnot* a *qFT* e *qFalse* produz o par emaranhado *p2* da Seção 2.3.

Mais geralmente, a operação realizada sobre o segundo valor não precisa ser a negação, e o valor de controle não precisa ser um booleano. Nós abstraímos dessas duas situações para definir uma operação controlada genérica que toma dois argumentos: um operador quântico u, que deverá poder ser aplicado ao qubit alvo, e uma função clássica *enable*, que decide para cada valor de controle a se ele vai habilitar a aplicação da operação u ao alvo:

```
cop :: (Basis a, Basis b) =>
        (a -> Bool) -> Qop b b -> Qop (a,b) (a,b)
cop enable (Qop u) =
    qop ([(((a,b),(a,b)),1) |
              (a,b) <- basis, not(enable a)] ++
         [(((a,b1),(a,b2)), pr u (b1,b2)) |
              a <- basis, enable a,
              b1 <- basis, b2 <- basis])
```

Se o valor de controle de entrada não é habilitado então o par de saída é idêntico ao par de entrada (primeiro grupo); caso contrário, se o valor de controle de entrada é habilitado e idêntico ao valor de controle de saída, então a contribuição é a dada pelo operador u. Em todos os outros casos, a probabilidade de saída é zero e portanto omitida, seguindo nossa convenção.

A operação *cnot* é facilmente obtível a partir da operação genérica:

```
cnot :: Qop (Bool,Bool) (Bool,Bool)
cnot = cop id qnot_op
```

Outra operação controlada comum é o não-controlado-controlado, também chamado de operação *toffoli* [23]. A operação *toffoli* é essencialmente idêntica à operaçao *cnot*, mas é controlada por um par de valores booleanos. Sua definição é quase idêntida à de *cnot*:

```
toffoli :: Qop ((Bool,Bool),Bool) ((Bool,Bool),Bool)
toffoli = cop (uncurry (&&)) qnot_op
```

A operação *toffoli* é signficativa porque ela pode implementar todas as operações booleanas clássicas. Quando ambos valores de controle são *True* a operação nega o valor alvo. Se o valor alvo é *False*, a operação realiza um *and* lógico dos valores de controle. Como ela pode implementar *and* e *not*, a operação pode implementar qualquer função booleana.

4. Medições

Valores, sejam resultantes de computação clássica ou de computação quântica, devem ser observados, no final, para comunicar resultados para o mundo exterior. Em um modelo de programação clássica, o processo de observar um valor simplesmente retorna o valor. Em um modelo quântico, o processo de observação é mais complicado.

4.1. **Normalização.** Até agora, nós não impusemos nenhuma restrição sobre as amplitudes de probabilidade associadas com os vetores unitários de um valor quântico. Isso é válido, já que a magnitudo dos vetores não tem relevância computacional. É útil, entretanto, ter uma representação normalizada, antes de discutir os detalhes das medições. Normalização simplesmente consiste em dividir cada amplitude de probabilidade pela *norma* do valor. (No código a seguir, nós usamos $|\ |*2$ para referir à função (*square . magnitude*).)

```
normalize :: Basis a => QV a -> QV a
normalize v = (1/(norm v) :+ 0) *> v

norm :: Basis a => QV a -> Double
norm v = let probs = map | |**2 (eltsFM v)
         in sqrt(sum probs)

(*>) :: Basis a => PA -> QV a -> QV a
c *> v = mapFM (\ _ a -> c*a) v
```

Por exemplo, normalizando *p1, p2* e *p3* produzem-se:

```
np1 = qv [((False,False),1/(sqrt 2),
          ((False,True),1/(sqrt 2))]
np2 = qv [((False,False),1/(sqrt 2),
          ((True,True),1/(sqrt 2)]
np3 = qv [((False,False),1/2),
          ((False,True),1/2),
          ((True,False),1/2),
          ((True,True),1/2)]
```

4.2. **Observando Valores Simples.** Seja b um valor booleano quântico normalizado $(p|False\rangle + q|True\rangle)$ onde $|p|^2 + |q|^2 = 1$. Uma medição de b:

- retorna um resultado *res* que é ou *False* com probabilidade $|p|^2$ ou *True* com probabilidade $|q|^2$;
- como um *efeito colateral*, atualiza b, de modo que todas as observações futuras retornam *res*.

Assim, tão logo o valor b é observado, qualquer superposição de *False* e *True* que possa ter estado presente desaparece e o valor se torna ou um puro *False* ou um puro *True*.

4.3. **Observação e Emaranhamento.** Dado um par do tipo *QV (a,b)*, a mecânica quântica permite três medições: uma medição do estado do próprio par (ambos componentes são medidos ao mesmo tempo); ou uma medição na qual ou o componente esquerdo ou o componente direito (mas não ambos) são medidos.

Em um certo sentido, é bastante estranho que se possa operar sobre um dos componentes de um par emaranhado, de modo individual, *sem* que se possa separar esse componente do outro. De fato, o processo de observação fornece um outro modo de entender o emaranhamento. Dois valores são emaranhados se a observação de um deles afeta a medição do outro. Olhando de novo para nossos exemplos da Seção 2.3, nós decidimos que os componentes de *np3* não são emaranhados e que os componentes de *np2* são emaranhados. De fato:

- Cada componente de *np3* é *False* e *True* com igual probabilidade e, portanto, a observação do primeiro componente pode retornar *False* ou *True*. Se ela retorna *False*, o par é "atualizado" (a função de onda colapsa) para ser consistente com essa observação e se torna:

  ```
  qv [((False,False),1/(sqrt 2)),((False,True),1/(sqrt 2))]
  ```

  Uma observação futura do segundo componente ainda pode ser *False* ou *True* com igual probabilidade.
- Em contraste, mesmo que cada componente de *np2* seja *False* ou *True* com igual probabilidade, os valores são correlacionados. A observação do primeiro componente retornar *False* ou *True*. Se ela retorna *False*, o par é atualizado e se torna:

  ```
  qv [((False,False),1)]
  ```

  e qualquer observação futura do segundo componente *deve* agora retornar *False*.

4.4. **O Paradoxo EPR.** A mecânica quântica descreve os fenômenos do emaranhamento e da observação sem interpretá-los:

> É importante notar que não existe mecanismo postulado nessa teoria, sobre como uma função de onda é enviada para um auto-estado por um observável. Assim como a lógica matemática não precisa exigir causalidade por detrás de uma implicação entre proposições, a lógica da mecânica quântica não exige uma causa específica por detrás das observações... Contudo, o debate sobre a interpretação da teoria quântica tem freqüentemente levado seus participantes a afirmar que a causalidade foi demolida na física. [16,p.6]

Se nós vamos prover um modelo operacional da computação quântica, nós vamos necessitar de uma interpretação da mecânica quântica para explicar *como* o segundo componente de um par é afetado quando o primeiro componente é observado. Para entender as dificuldades, é útil rever o famoso paradoxo de Einstein, Podolsky e Rosen [9] e algumas tentativas de resolvê-lo.

Einstein, Podolsky e Rosen [9] propuseram um experimento de pensamento que usa valores emaranhados de um modo que parece violar princípios fundamentais da relatividade. A questão é a seguinte: quando um componente de um par de valores

emaranhados é observado, *como* a informação sobre o valor observado flui para o outro componente, se há realmente alguma informação fluindo entre eles! Há duas tentativas padrão para resolver esse paradoxo:

(1) A primeira explicação tentada é que cada componente do par tem um *estado local* que determina seu valor observado. Antes da observação, o estado local está oculto e somente pode ser descrito probabilisticamente. Tão logo o componente é observado, o estado oculto é exposto. No caso do par *np2*, acima, o estado local oculto de cada componente poderia ser *False*; os componentes poderiam então ser observados em qualquer ordem, e sem comunicação ou interação, ambas observações seriam iguais às esperadas. Se válida, essa idéia forneceria um modelo computacional simples e completamente local para a computação quântica. Infelizmente, Bell formula essa idéia matematicamente e mostra que ela é incompatível com as previsões estatísticas da mecânica quântica [2]. Bell conclui que qualquer teoria baseada em variáveis ocultas e que é consistente com as previsões estatísticas da mecânica quântica deve também incluir um mecanismo pelo qual o ajuste de um dispositivo de medida pode influenciar na leitura de outro instrumento, mesmo que remoto, e que o sinal entre eles deve se propagar instanteamente. Isso viola a relatividade especial.
(2) A outra tentativa de explicação está bastante relacionada com a primeira: o valor de cada componente é uma função de um valor medido do outro componente. Qualquer que seja o componente medido em primeiro lugar, ele comunica seu valor ao outro componente, que atualiza seu valor. Mas, como Einsteni, Podolsky e Rosen notaram, essa explicação também viola os princípios da relatividade especial. A noção de um componente sendo medido "em primeiro lugar" não é uma noção bem definida, já que depende da velocidade do agente que observa as medições. Em outras palavras, é possível que um observador veja que o componente da esquerda foi medido em primeiro lugar, enquanto um outro observador vê que foi o componente da direita que foi medido primeiro. Em suma, a idéia de comunicar um valor do primeiro componente a ser medido para o segundo componente não é compatível com ambos observadores, já que experimentos são invariantes sob a mudança de observadores.

Infelizmente, mesmo que essas duas explicações sejam sabidamente erradas, não existem realmente outras explicações amplamente aceitas. Existem, entretanto, diversas interpretações interessantes que deveriam ser investigadas com mais profundidade, na medida em que elas poderiam fornecer modelos operacionais interessantes para a computação quântica: em particular, duas interpretações relevantes são a interpretação dos múltiplos mundos, na qual todas as observações possíveis são realizadas em universos paralelos [10] e a interpretação transacional, na qual a computação é descrita como o ponto fixo de um processo que acontece no tempo tanto na direção de avanço temporal quanto de retrocesso temporal [6].

Para nossos propósitos, nós adotamos o mecanismo operacional mais simples para a observação de componentes de estruturas de dados emaranhadas, tal que o resultado da observação afeta todos os outros valores emaranhados: nós usamos um efeito colateral global sobre uma referência compartilhada. A comunicação entre os valores emaranhados acontece implicitamente e instantaneamente através da atribuição à referência compartilhada. Mesmo que isso não seja razoável de

uma perspectiva física, ela parece razoável em um ambiente de programação sem múltiplas threads. Na presença de múltiplas threads (caso que nós não consideramos), um problema reminiscente do problema notado por Einstein, Podolsky e Rosen pode ocorrer na forma de condições de corrida, se duas threads tentam medir diferentes componentes do par simultaneamente. Permanece para ser mostrado se o uso de efeitos colaterais globais em nosso modelo pode levar simulações computacionais quânticas a fornecerem resultados e efeitos que não correspondem a contra-partes físicas.

4.5. **Referências a Valores Quânticos.** Para modelar os efeitos colaterais implícitos no processo de observação, nós vamos usar referências explícitas: valores quânticos somente podem ser acesados via células de referência; a observação atualiza as células de referência com o valor observado:

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

mkQR :: QV a -> IO(QR a)
mkQR v = do r <- newIORef v
            return (QR r)
```

A função *mkQR* é uma ação-IO a qual, quando executada, aloca uma nova célula de referência e armazena nela o valor quântico fornecido.

Para observar um valor quântico acessível através de uma referência $QR\ a$, nós lemos o conteúdo da referência, observamos o valor e atualizamos a referência com o resultado da observação. Observar um valor requer os seguintes passos. Primeiro, nós normalizamos o valor. Então, nós calculamos a probabilidade associada com cada vetor unitário na base. Para cada vetor unitário, nós também computamos uma probabilidade cumulativa, a qual é a soma de sua probabilidade com todas as probabilidades dos vetores unitários antes dele na ordem (arbitrária, já que irrelevante) dada pela base. Já que probabilidades somam 1, nós escolhemos um número aleatório entre 0 e 1 e escolhemos o primeiro construtor com uma probabilidade cumulativa que excede esse número randômico:

```
observeR :: Basis a => QR a -> IO a
observeR (QR ptr) =
    do v <- readIORef ptr
       res - observeV v
       writeIORef ptr (unitFM res 1)
       return res

observeV :: Basis a => QV a -> IO a
observeV v =
    do let nv = normalize v
           probs = map ((| | ** 2) . pr nv) basis
       r <- getStdRandom (randomR(0.0,1.0))
       let cPsCs = zip (scanl1 (+) probs) basis
           Just(_,res) = find(\(p,_) -> r < p) cPsCs
       return res
```

Por exemplo, cada avaliação de *test*, abaixo, imprime ou três ocorrências de *False* ou três ocorrências de *True*: a primeira observação pode tanto ser *False* ou *True*, com igual probabilidade, mas uma vez realizada ela fixa o resultado das duas observações seguintes:

```
test = do x <- mkQR qFT
          o1 <- observeR x
          o2 <- observeR x
          o3 <- observeR x
          print (o1, o2, o3)
```

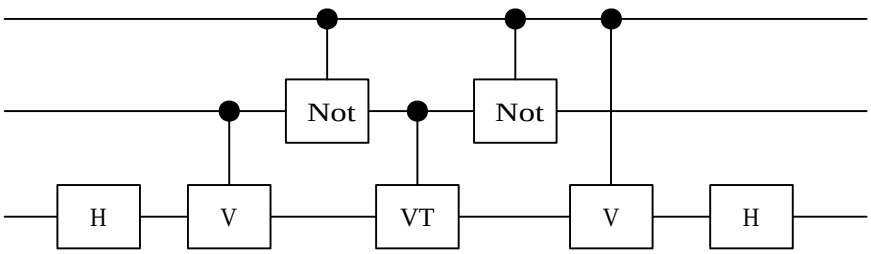
A observação de um dos componentes de um par é ligeiramente mais complicada. Nós somente mostramos o caso da observação do componente esquerdo do par; o outro caso é simétrico:

```
observeLeft :: (Basis a, Basis b) =>
                  QR (a,b) -> IO a
observeLeft (QR ptr) =
   do v <- readIORef ptr
      let leftF a = sqrt (sum [|pr v (a,b)| ** 2 | b <- basis]
          leftV = qv [(a, leftF a)|a <- basis]
      aobs <- observeV leftV
      let nv = qv [((aobs,b), pr v (aobs,b)) | b <- basis]
      writeIORef ptr (normalize nv)
      return aobs
```

Nós primeiro construímos um valor quântico *virtual* (*leftV*) que dá a probabilidade de cada vetor unitário ocorrer componente esquerdo. Essa probabilidade é calculada somando todas as ocorrências desse vetor unitário no par. O valor virtual é observado e isso seleciona um dos vetores unitários. O par é reconstituído com somente os componentes que são consistentes com a observação e o resultado é armazenado na célula de referência.

5. Dualidade Onda/Partícula

Em princípio, nós cobrimos a parte básica da computação quântica e podemos ir adiante, para alguns exemplos. Um exemplo elementar que nós consideramos é modelar essa implementação alternativa da operação *toffoli*:



O diagrama do circuito usa a notação padrão *de facto* para especificar computações quânticas. A convenção é que os valores fluem da esquerda para a direita em pasos correspondendo ao alinhamento das portas lógicas. Para o restante dessa discussão, nós nos referimos aos três qubits relevantes como *top*, *middle* e *bottom*:

(1) No primeiro passo, a operação *hadamard* é aplicada ao qubit *bottom*.
(2) No segundo paso, uma operação controlada *v_op* (que é definida abaixo) é aplicada ao par consistindo dos qubits *middle* e *bottom*:

```
v_op :: Qop Bool Bool
v_op = qop [((False,False),1),
         ((True,True),0 :+ 1)]
```

(3) No terceiro passo, a operação controlada *cnot* é aplicada ao par consistindo dos qubits *top* e *middle*.
(4) No quarto passo, uma operação controlada *vt_op* (a adjunta, ou transposta conjugada da *v_op*, definida acima) é aplicada ao par consistindo dos qubits *middle* e *bottom*:

```
vt_op :: Qop Bool Bool
vt_op = qop [((False,False),1),
          ((True,True),0 :+ 1)]
```

(5) O quinto passo é idêntico ao terceiro passo.
(6) No sexto passo, uma operação controlada *v_op* é aplicada ao par consistindo dos qubits *top* e *bottom*.
(7) Finalmente, no último passo, a operação *hadamard* é aplicada ao qubit *bottom*.

Implementar esse circuito razoavelmente elementar é complicado pelo fato de que os três qubits *top*, *middle* e *bottom* estão geralmente emaranhados. Não é possível manipular diretamente apenas o qubit *bottom* como requerido pelo primeiro passo do exemplo. Pior ainda, o circuito requer que as operações sejam aplicadas a três pares distintos de qubits: (*middle*,*bottom*), (*top*,*middle*) e (*top*,*bottom*) o que novamente, por definição de emaranhamento, não pode ser isolado para se adequar a cada operação. Essa situação é a contra-parte de programação da dualidade onda/partícula: por um lado, os três valores emaranhados formam uma onda "conectada"; por outro lado, cada uma delas é uma partícula independente, a qual pode ser operada individualmente, com o entendimento de que o resultado de tal operação afeta a onda inteira.

O modo ingênuo de modelar computações, tal como o modo utilizado acima, é definir funções especializadas que operam com componentes de estruturas de dados, como a nossa função *observeLeft* da Seção 4.5. Esse modo escapa rapidamente ao controle e diversos modelos de computação quântica tentam fornecer um mecanismo mais geral para lidar com esse problema. Por exemplo, Selinger inclui operações que realizam *permutações* arbitrárias das variáveis [24] e QCL [22] inclui a noção de um *registrador simbólico* que pode referir qualquer coleção de qubits mesmo se eles fazem parte de estruturas emaranhadas. Em nosso caso,nós propomos uma idéia relacionada de *valores virtuais*.

5.1. **Valores Virtuais e Adaptadores.** Um valor virtual é um valor que, apesar de possivelmente estar profundamente embutido dentro de uma estrutura e emaranhado com outros valores, pode ser operado individualmente. Um valor virtual é especificado dando toda a estrutura de dados ao qual ele pertence e um *adaptador*, que especifica o mapeamento dessa estrutura de dados toda para o valor em questão, e vice-versa. Mais especificamente, nós temos:

```
data Adaptor l g =
     Adaptor { dec :: g -> l, cmp :: l -> g}
data Virt a na u = Virt (QR u) (Adaptor (a,na) u)
```

O tipo (*Virt a na u*) define um valor virtual de tipo a que é emaranhado com valores de tipo *na*. O tipo u é o tipo da estrutura de dados total, que contém tanto a quanto *na*. O adaptador mapeia nas duas direções entre o tipo u e sua decomposição. Valores virtuais são relacionados a referências componíveis [14], que fornecem acessos a um campo ou a uma subestrutura relativa a uma tupla ou registro maior, usado como estado.

Por exemplo, na estrutura de dados do tipo

```
QV (((a,b,c),(d,e)),(f,g))
```

existem diveras maneiras de isolar um valor quântico do tipo *QV(d,g)*, dependendo de como se decide agrupar os outros valores com cada d e g. Dois modos possíveis são:

```
mkVirt1 :: QR (((a,b,c),(d,e)),(f,g)) ->
           Virt (d,g) (a,b,c,e,f) (((a,b,c),(d,e)),(f,g))
mkVirt1 r = Virt r a1
    where a1 =
          Adaptor { dec = \(((a,b,c),(d,e)),(f,g)) ->
                            ((d,g),(a,b,c,e,f)),
                    cmp = \((d,g),(a,b,c,e,f)) ->
                            (((a,b,c),(d,e)),(f,g)) }

mkVirt2 :: QR (((a,b,c),(d,e)),(f,g)) ->
           Virt (d,g) ((a,b,c),e,f) (((a,b,c),(d,e)),(f,g))
mkVir2 r = Virt r a2
    where a2 =
          Adaptor { dec = \(((a,b,c),(d,e)),(f,g)) ->
                            ((d,g),((a,b,c),e,f)),
                    cmp = \((d,g),((a,b,c),e,f)) ->
                            (((a,b,c),(d,e)),(f,g)) }
```

O mecanismo dos valores virtuais nos permite fingir que existe um par do tipo (d, g) na estrutura, mesmo que o tipo (d, g) não ocorra diretamente no tipo da estrutura e que os componentes do tipo d e g estejam profundamente aninhados.

Isso é reminiscente do padrão Fachada [12], no qual uma estrutura profundamente aninhada recebe uma interface plana, que dá acesso a seus referentes internos.

5.2. **Gerando Adaptadores.** A definição de adaptadores (pelo menos para estruturas de dados como tuplas) é tão regular que nós deveríamos ser capazes de automatizar sua geração apenas a partir do tipo da informação. Nós assumimos no restante deste artigo que os seguintes adaptadores foram gerados. Nós somente damos as definições para os dois primeiros:

```
ad_pair1 :: Adaptor (a1,a2) (a1,a2)
ad_pair1 = Adaptor { dec = \(a1,a2) -> (a1,a2),
                     cmp = \(a1,a2) -> (a1,a2) }
ad_pair2 :: Adaptor (a2,a1) (a1,a2)
ad_pair2 = Adpator { dec = \(a1,a2) -> (a2,a1),
                     cmp = \(a2,a1) -> (a1,a2) }
ad_triple23 ...
ad_triple12 ...
ad_triple13 ...
```

5.3. **Tudo é um Valor Virtual.** Para prover um modelo uniforme, nós reformulamos todas nossas operações em termos de valores virtuais. Primeiro, nós fornecemos um modo de converter referências individuais para valores quânticos em valores virtuais triviais e um modo de criar valores virtuais a partir de outros valores virtuais, pela composição de um novo adaptador:

```
virtFromR :: QR a -> Virt a () a
virtFromR r =
    Virt r (Adaptor { dec = \a -> (a,()),
                      cmp = \(a,()) -> a })

virtFromV :: Virt a na u ->
                 Adaptor (a1,a2) a ->
                     Virt a1 (a2,na) u
virtFromV (Virt r (Adaptor {dec = gdec, cmp = gcmp}))
          (Adaptor {dec = ldec, cmp = lcmp}) =
    Virt r (Adaptor { dec = \u -> let (a,na) = gdec u
                                      (a1,a2) = ldec a
                                  in (a1,(a2,na)),
                      cmp = \(a1,(a2,na)) ->
                                  gcmp(lcmp(a1,a2),na) })
```

Uma operação sobre valores quânticos recebeu previamente o tipo *Qop a b*, denotando o fato de que ela mapeia valores quânticos do tipo $QV\ a$ em valores quânticos do tipo $QV\ b$. Ao invés de valores quânticos simples, como antes, os valores de entrada e saída são agora virtuais, i.é, eles são to tipo *Virt a na ua* e *Virt b nb ub*. A operação do tipo *Qop a b* deve ainda fazer sentido, já que os valores de entrada e saída forem do tipo certo, exceto que eles são emaranhados, em estruturas grandes.

A aplicação não deve afetar, entretanto, esses outros valores emaranhados, que devem portanto ser do mesmo tipo. Assim, a aplicação geral é definida como segue:

```
app :: (Basis a, Basis b, Basis nab, Basis ua, Basis ub) =>
       Qop a b -> Virt a nab ua -> Virt b nab ub -> IO()
app (Qop f)
    (Virt (QR ra) (Adaptor { dec = deca, cmp = cmpa }))
    (Virt (QR rb) (Adaptor { dec = decb, cmp = cmpb })) =
    let gf = qop [((ua,ub),pr f (a,b))) |
                         ua <- basis, ub <- basis,
                         let (a,na) = deca ua,
                             (b,nb) = decb ub,
                         in na == nb ]
    in do fa <- readIORef ra
          let fb = normalize (qApp gf fa)
          in writeIOREf rb fb
```

O primeiro argumento é a operação a ser aplicada. Os seguintes dois argumentos são os valores de entrada e saída virtuais, que compartilham os mesmos vizinhos emaranhados. A operação é promovida de alguma coisa agindo sobre a e b para alguma coisa agindo sobre toda a estrutura emaranhada, no modo esperado.

Em geral, os valores virtuais de entrada e saída podem ser diferentes. Por exemplo, dado o valor virtual

```
ip :: Virt Move Bool (Move,Bool)
```

e um valor virtual

```
op :: Virt Rotation Bool (Bool, Rotation)
```

nós podemos usar *app m2r ip op* para traduzir de um estado de polarização de um fóton para outro estado, mesmo quando o fóton está emaranhado com algum qubit.

É mais comum, em exemplos simples, ter somente uma referência global para um valor quântico do tipo $QV\ u$, que é repetidamente atualizado no lugar, por operações sucessivas. Cada uma das operações sucessivas é do tipo $Qop\ a\ a$ para algum tipo a, que pode ser extraído de u via um adaptador. Para essas aplicações, nós podemos usar a seguinte versão mais simples de *app*:

```
app1 :: (Basis a, Basis na, Basis ua) =>
         Qop a a -> Virt a na ua -> IO()
app1 f v = app f v v
```

Um valor virtual pode ser observado usando uma idéia que generaliza *observeLeft* da Seção 4.5, usando o adaptador para decompor e compor o valor, ao invés do conhecimento pré-definido de que nós estamos manipulando o componente esquerdo de um par:

```
observeVV :: (Basis a, Basis na, Basis u) =>
             Virt a na u -> IO a
observeVV (Virt (QR r) (Adaptor { dec = dec, cmp = cmp })) =
     do v <- readIORef r
        let virtF a =
               sqrt (sum [(abs (pr v (cmp(a,na))))**2|
                          na <- basis]
        let virtV = qv [(a,virtF a) | a <- basis]
        aobs <- observeV virtV
        let nv = qv [(u,pr v (cmp(aobs,na)))|
                     u <- basis,
                     let (a,na) = dec u,
                     a == aobs]
        writeIORef r (normalize nv)
        return aobs
```

6. Exemplos

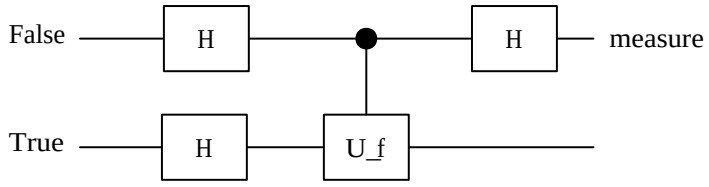
A maquinaria que nós desenvolvemos pode parecer bastante pesada, mas ela é bastante poderosa e torna a programação de diagrama de circuitos como o exemplo *toffoli* da Seção 5 bastante simples. O código completo (excluindo adaptadores) é:

```
toffoli' :: (Basis na, Basis u) =>
               Virt (Bool, Bool, Bool) na u -> IO()
toffoli' vtriple =
    let b = virtFromV vtriple ad_triple3
        mb = virtFromV vtriple ad_triple23
        tm = virtFromV vtriple ad_triple12
        tb = virtFromV vtriple ad_triple13
        cv = cop id vop
        cvt = cop id vtop
    in do app1 hadamard_op b
          app1 cv mb
          app1 cnot tm
          app1 cvt mb
          app1 cnot tm
          app1 cv tb
          ap1 hadamrd_op b
```

Dados três valores quânticos booleanos (emaranhados, ou não; parte de uma estrutura de dados maior, ou não), nós começamos por isolar as partes relevantes e, então, simplesmente aplicamos as operações na maneira óbvia: uma linha para cada passo no circuito. Nós usamos os mnemônicos *mb* para referir ao par de valores *middle* e *bottom*, *tm* para referir ao par de valores *top* e *middle*, etc.

6.1. **O Oráculo de Deutsch.** Um outro exemplo interessante é o oráculo de Deutsch [7], que dada uma função sobre booleanos decide com *uma só* chamada da função se ela é balanceada (*id* ou *not*) ou constante (*const True* ou *const False*).

Claro que a simulação Haskell aplica a função a ambos os valores *True* e *False*, mas uma implementação quântica real aplicaria a função uma vez só à superposição quântica. O exemplo não requer realmente a maquinaria dos valores virtuais, mas ele usa o poder da operação controlada genérica da Seção 3.2:




```
deutsch :: (Bool -> Bool) -> IO()
deutsch f =
    do inpr <- mkQR(qFalse &* qTrue)
       let both = virtFromR inpr
           top = virtFromV both ad_pair1
           bot = virtFromV both ad_pair2
           uf = cop f qnot
       app1 hadamard_op top
       app1 hadamard_op bot
       app1 uf both
       app1 hadamard_op top
       topV <- observeVV top
       putStr(if topV then "Balanced" else "Constant")
```

O oráculo funciona como segue. O valor *top* é transformado pela operação *hadamard* em $|False\rangle+|True\rangle$ e o valor *bottom* é transformado em $|False\rangle-|True\rangle$. Existem diversos casos, dependendo de f:

- Se f é *const False*: a linha de controle está sempre desabilitada e ambos valores *top* e *bottom* não mudam. A última operação *hadamard* transforma o valor *topo* $|False\rangle + |True\rangle$ em $(|False\rangle + |True\rangle) + (|False\rangle - |True\rangle)$, que simplifica para $|False\rangle$, se nós ignoramos o fator de normalização, como usual.
- f é *id*: a linha de controle é uma superposição $|False\rangle + |True\rangle$ e o valor *bottom* é fica ao mesmo tempo inalterado e negado, de uma maneira emaranhada com o valor *topo*. Mais precisamente, o par resultante dos valores *top* e *bottom* é:

  ```
  |(False,False)> - |(False,True)>
      + |(True,True)> - |(True,False)>
  ```

  que pode ser explicado como segue. Os primeiros dois componentes correspondem aos casos nos quais o valor *top* é *False*; como *f False* também é *False*, a linha de controle fica desabilitada e o valor *bottom* é $|False\rangle - |True\rangle$. Os dois últimos casos correspondem aos casos nos quais o valor *top* é *True*; como *f True* também é *True*, a linha de controle está habilitada e o valor *bottom* se torna $|True\rangle - |False\rangle$. Finalmente, o valor *top* é operado pelo *hadamard*, deixando o valor *bottom* intacto. Isso produz:

```
|(False,False)> + |(True,False)>
    - |(False,True)> - |(True,True)>
    + |(False,True)> - |(True,True)>
    - |(False,False)> + |(True,False)>
```

que simplifica para: $|(True, False) > -|(True, True) >$. Assim se observa que o valor *top* (esquerda) sempre retorna *True*.

- As situações em que f é *const True* ou *not* são como acima. No caso em que f é *const True*, a linha de controle está sempre habilitada e o valor *bottom* está sempre negado e, portanto, não está emaranhado com o valor *top*. No caso em que f é *not*, os valores estão emaranhados e uma análise similar mostra que o valor *top* (esquerdo) avalia para *True*. Portanto, em todos os casos, se o valor *top* é observado como *False*, a função é constante e, se o valor *top* é observado como *True*, a função é balanceada.

6.2. **Somador Quântico.** Um somador quântico de 1 bit pode ser definido usando Toffoli e portas não controladas [23]. As principais características do código são:

```
adder :: QV Bool -> QV Bool -> QV Bool -> IO()
adder inc x y =
    let sum = qFalse
        outc = qFalse
        adder_inputs = inc &* x &* y &* sum &* outc
    in do r <- mkQR vals
          let v = virtFromV (virtFromR r)...
          ...
          app1 toffoli vxy0
          app1 toffoli vix0
          app1 toffoli viy0
          ap1 cnot vis
          app1 cnot vxs
          app1 cnot vys
          (sum,out_carry) <- observeR vso
          print (sum,out_carry)
```

No código, nós omitimos os adaptadores. Os valores virtuais chamados v, com subscritos, usam as seguintes convenções: i se refere ao qubit de vem-um, x e y se referem aos dois qubits a serem somados, s se refere ao qubit de soma e o se refere ao qubit de vai-um. Assim, *vixo* é o valor virtual que se refere aos três qubits seguintes: vem-um, primeira entrada e vai-um. A somador pode ser chamado com *qFalse qTrue qTrue*, caso em que ele funciona como um somador clássico e produz $(False, True)$, mas também pode ser chamado com *qFT qFT qFT*.

7. Conclusões

Nós apresentamos um modelo de computação quântica embutida em Haskell. Nós esperamos que o model dê boas intuições sobre a computação quântica, para programadores. Nós temos usado o modelo para escrever diversos outros algoritmos, incluindo o algoritmo de fatoração de Shor [26]. Para exemplos mais complicados que os apresentado aqui, é útil ter um tipo de inteiros módulo n, para permitir

parametrização conveniente dos algoritmos sobre o tamanho da entrada. Isso pode ser codificado usando classes de tipos [13, 20], mas de um modo complicado e poderia talvez beneficiar-se de extensões de meta-programação de Haskell [25]. Também, mesmo que nós acreditemos que a idéia dos *valores virtuais* seja a idéia correta, talvez sua formulação corrente deixe espaço para melhorias.

Nosso modelo elicita uma diferença fundamental entre linguagens de programação clássica e linguagens de programação quântica. Na teoria das linguagens de programação clássicas, as expressões da linguagem podem ser agrupadas em *construtos de introdução* e *construtos de eleminação*, para os conetivos de tipo da linguagem. Uma linguagem de programação quântica só pode ter construtos de eliminação *virtuais*, porque, por definição, os elementos de uma estrutura de dados emaranhada não podem ser separados. Essa restrição leva a um estilo de programação não usual o qual, nós argumentamos, requer novas primitivas de programação.

Nosso modelo se distingue de outros sobre computação quântica e programação funcional, como segue. Tanto Skibinski [27] e Karczmarczuk [15] usaram Haskell extensivamente para modelar as estruturas matemáticas subjacentes à mecânica quântica, o que é um foco diferente e complementar ao nosso. Skibinski [28] também implementou um simulador Haskell para computação quântica, que opera no nível "físico" de abstração dos qubits e portas lógicas, o que nós, ao contrário, tentamos abstrair pelo uso de tipos de dados abstratos e funções.

Também existem diversas propostas de "linguagens de programação quântica" [22, 8, 24]. Tanto **pGCL**, uma linguagem imperativa que estende a linguagem de comandos guardados de Dijkstra [8], quanto **QPL**, uma linguagem funcional tipada com dados quânticos [24], são bem desenvolvidas e semanticamente bem fundamentadas e poderiam fornecer indicações interessantes para a programação funcional.

Agradecimentos

Nós gostaríamos de agradecer aos revisores anônimos pelos comentários extensivos e muito úteis.

Referências

[1] H. G. Baker. NREVERSAL of fortune – the thermodynamics of garbage collection. In: Y. Bekkers and J. Cohen, editors, *Memory Management; International Workshop IWMM 92; Proceddings*, pages 507-524, Berlin, Alemanha, 1992. Springer-Verlag.

[2] J. S. Bell. On the Einstein-Podolsky-Rosen paradox. In: [3], pages 14-21. Cambridge University Press, 1987.

[3] J. S. Bell. *Speakable and Unspeakable in Quantum Mechanics*. Cambridge University Press, 1987.

[4] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525-532, Nov. 1973.

[5] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. *Lecture Nots in Computer Science*, 2076:1017-1027, 2001.

[6] J. G. Cramer. The transactional interpretation of quantum mechanics. *Modern Physics*, 58:647-688, 1986.

[7] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. Roy. Soc. London, Ser. A*, 400:97-117, 1985.

[8] E. W. Dijkstra. *A Discipline of Programming*, capítulo 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[9] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, 47:777-780, 1935.

[10] H. Everett, III. "Relative state" formulation of quantum mechanics. *Reviews of Modern Physics*, 29:454, 1957.
[11] M. P. Frank. *Reversibility for Efficient Computing*. Tese PhD, MIT, 1999.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
[13] R. Hinze. Haskell does it with class. Slides of a talk given at the Generic Haskell meeting, Maio 2001.
[14] K. Kagawa. Mutable data structures and composable references in pure functional language. In: *State in Programming Languages (SIPL'95)*, p. 79-94, Jan. 1995.
[15] J. Karczmarczuk. Structure and interpretation of quantum mechanics - a functional framework. In: ACM SIGPLAN Haskell Workshop, 2003.
[16] L. H. Kauffman. *Quantum Topology and Quantum Computing*, capítulo IV de [18]. Americam Mathematical Society, 2002.
[17] W. Kluge. A reversible SE(M)CD machine. In: *11th International Workshop on the Implementation of Functional Languages, Lochem, The Netherlands, September 7-10, 1999*, número 1868 de Lecture Notes in Computer Science, p. 95-113. Springer-Verlag, Setembro 2000.
[18] S. J. Lomonaco, Jr., editor. *Quantum Computation: A Grand Mathematical Challenge for the Twenty-First Century and the Millenium*, vol. 58 de *Proceedigns of Symposia in Applied Mathematics*. American Mathematical Society, Mar. 2002.
[19] S. J. Lomonaco, Jr. *A Rosetta Stone for Quantum Mechanics with an Introduction to Quantum Computation*, capítulo I de [18]. Americam Mathematical Society, 2002.
[20] C. McBride. Faking it – simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375-392, Julho 2002.
[21] S.-C. Mu e R. Bird. Functional Quantum Programming. In: *Second Asian Workshop on Programming Languages and Systems, KAIST, Korea*, Dez. 2001.
[22] B. Ömer. A procedural formalism for quantum computing. Tese de mestrado, Department of Theoretcial Physics, Technical University of Vienna, 1998.
[23] E. Rieffel e W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, 32(3):300-335, Set. 2000.
[24] P. Selinger. Towards a quantum programming language. Não publicado, 2002.
[25] T. Sheard e S. Peyton-Jones. Template meta-programming for Haskell. In: *Proc. of the Workshop on Haskell*, p. 1-16. ACM, 2002.
[26] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484-1509, 1997.
[27] J. Skibiński. Collection of Haskell modules. Disponível em http://web.archive.org/web/20010415043244/www.numeri-quest.com/haskell/index.html, Criado: 1998-09-18, última modificação: 2001-04-02.
[28] J. Skibński. Haskell simulator of quantum computer. Disponível em http://web.archive.org/web/20010630025035/www.numeri-quest.com/haskell/QuantumComputer.html, Criado: 2001-05-02, última modificação: 2001-05-05.
[29] A. Steane. Quantum Computing. *Reports on Progress in Physics*, 61:117-173, 1998.

Department of Computer Science, Indiana University
E-mail address: sabry@cs.indiana.edu