Implementando o Cálculo Lambda Quântico em Haskell Utilizando a Modelagem de Sabry[*‡]

Flávio Borin Júnior[1], **Juliana Kaizer Vizzotto**[1]

[1]Curso de Ciência da Computação – Universidade Federal de Santa Maria (UFSM)
Av. Roraima, 1000 – 97.105-900 – Santa Maria – RS – Brasil

{fbjunior, juvizzotto}@inf.ufsm.br

Abstract. *This article aims to present a project that seeks an adaptation of Lambda Calculus to the realm of Quantum Computing. This endeavor is based on the quantum behavior implementation proposed by Sabry and the Quantum Lambda Calculus definition provided by Van Tonder. By combining these approaches, the goal is to explore and gain insights into the development of languages, and consequently algorithms, for Quantum Computing.*

Resumo. *Este artigo tem como objetivo apresentar a investigação de uma adaptação do Cálculo Lambda para a esfera da Computação Quântica. Para tanto apresenta-se a implementação do modelo de computação quântica proposto por Sabry e a definição do Cálculo Lambda Quântico fornecida por Van Tonder. Ao unir estas abordagens, almeja-se explorar e aprender sobre o desenvolvimento de linguagens, e consequentemente algoritmos, para a Computação Quântica.*

1. Introdução

O recente desenvolvimento da computação quântica tem transformado a forma como encaramos a resolução de problemas complexos. Ao explorar os princípios da mecânica quântica, esta abordagem promete a realização de computações significativamente mais rápidas do que a suportada por computadores clássicos. No entanto, a complexidade inerente aos sistemas quânticos tem-se revelado um desafio, exigindo uma compreensão não apenas da mecânica quântica, mas também das linguagens de programação que permitem controlar esses sistemas. Nesse contexto, as linguagens de programação de alto nível são poderosas ferramentas para desmistificar a computação quântica, ao abstrair esta complexidade e viabilizar uma compreensão mais acessível e uma programação eficaz desses dispositivos quânticos intricados.

A convergência entre a computação quântica e as linguagens de programação funcionais também desempenha um papel fundamental na compreensão e na exploração eficaz da computação quântica. Linguagens funcionais, que se baseiam em funções matemáticas e em abstrações de alto nível, compartilham uma afinidade natural com os princípios quânticos. Essas linguagens, como Haskell, fornecem checagem de tipos em tempo de compilação e provas de corretude. Para estas linguagens, existe uma extensa gama de trabalhos que podem ser base para o desenvolvimento de novas linguagens quânticas [Yanofsky and Mannucci 2008].

[*]Trabalho concluído

[†]O presente trabalho foi realizado com o apoio do CNPq(Brasil).

Um forte exemplo de trabalho, nesta linha de desenvolvimento, é o de Van Tonder [van Tonder 2004], que expande o conceito clássico de cálculo lambda para que suporte o desenvolvimento de operações que envolvam elementos de sistemas quânticos. O cálculo lambda, ao ser adaptado a nível quântico, fornece uma base sólida para o desenvolvimento de linguagens funcionais.

Uma base para a simulação da computação quântica em linguagens funcionais é fornecida por Amr Sabry [Sabry 2003]. Ao emular qubits em Haskell, Sabry possibilita um claro entendimento do comportamento quântico ao implementá-lo.

Desta forma, unindo ambas as abordagens, almeja-se o controle de qubits emulados em Haskell através de expressões do cálculo lambda. E, ao alcançar abordagens de mais alto nível para a computação quântica, expandir a compreensão e cognição sobre o funcionamento de linguagens de programação e sobre como sistemas quânticos possuem vantagens sobre sistemas clássicos.

O presente artigo está estruturado da seguinte maneira. Na Seção 2 apresenta-se a construção de um modelo de computação quântica em Haskell. Na Seção 3 discute-se a definição do Cálculo Lambda Quântico. Na seção 4 apresenta-se o desenvolvimento do Cálculo Lambda Quântico sobre os qubits apresentados nas Seção 2. Por fim, na Seção 5 discute-se as limitações da implementação apresentada e soluções para trabalhos posteriores.

2. Um modelo de computação quântica em Haskell

Nessa seção, apresenta-se brevemente o trabalho do autor Amr Sabry [Sabry 2003] no qual é discutido a construção de um modelo de computação quântica em Haskell.

Na computação quântica, o processamento de informação é realizado através de sistemas físicos quânticos. A unidade básica de informação é o qubit, um sistema de dois estados básicos: zero e um. Esses estados são representados em notação bra-ket como $|1\rangle$ e $|0\rangle$ respectivamente. Diferentemente de um bit clássico, os estados de um qubit estão sempre associados a uma amplitude de probabilidade. Desta forma, o estado de um qubit é matematicamente definido como $|\psi\rangle = \alpha|1\rangle + \beta|0\rangle$, onde α e β são números complexos.

Sendo assim, a computação quântica permite estados em sobreposição. Um exemplo de estado em sobreposição é $\frac{1}{\sqrt{2}}|1\rangle + \frac{1}{\sqrt{2}}|0\rangle$, chamado estado $|+\rangle$; ou ainda $\frac{1}{\sqrt{2}}|1\rangle - \frac{1}{\sqrt{2}}|0\rangle$, chamado estado $|-\rangle$. Esses estados indicam que a probabilidade de se ler quaisquer dos valores básicos é de 50%. Como a probabilidade total do estado precisa somar 100%, temos que $|\alpha|^2 + |\beta|^2 = 1$

No trabalho de Sabry [Sabry 2003] qubits são implementados como um dicionário que mapeia valores à probabilidades. Todo estado possui uma probabilidade de ser lido como resultado, mesmo que esta chance seja zero. Em Haskell, um valor quântico pode ser então definido como:

type QV a = Map a PA

Desta forma, um qubit é visto como um conjunto de tuplas *(Valor, Amplitude de Probabilidade)*. Deste ponto de vista, os estados dos qubits $|1\rangle$ e $|0\rangle$ são representados, respectivamente, por $\{(|1\rangle; 1)\}$ e $\{(|0\rangle; 1)\}$, e um qubit no caso de sobreposição $|+\rangle$, por $\{(|1\rangle; \frac{1}{\sqrt{2}}), (|0\rangle; \frac{1}{\sqrt{2}})\}$.

Estados quânticos podem ser combinados formando grupos de qubits. Matematicamente, a operação de combinação é definida como o produto tensorial, representado por $\otimes$. Em notação bra-ket, denota-se que $|1\rangle \otimes |0\rangle = |10\rangle$. Sabry implementa grupos de qubits como mapeamento de tuplas para probabilidades. Dessa forma, um grupo de qubits $|10\rangle$, é representado por $\{(|10\rangle; 1)\}$.

A definição de um algoritmo no sistema quântico é realizada através de portas lógicas quânticas. Estas portas definem transições entre dois estados quânticos. Uma característica relevante das operações quânticas é que elas são sempre reversíveis. Como consequência, quando se sabe a sequência na qual todo o processamento da informação foi realizado, pode-se reverter total ou parcialmente as operações, retornando para um estado posterior [van Tonder 2004]. Um exemplo famoso é a porta de Hadamard (*H*), que mapeia os estados da base para estados em sobreposição e vice-versa.

$$H|0\rangle = |+\rangle$$
$$H|1\rangle = |-\rangle$$
$$H|-\rangle = |1\rangle$$
$$H|+\rangle = |0\rangle$$

Para Sabry, uma porta quântica, é definida como um conjunto de transições entre duas n-uplas de qubits, associada a uma probabilidade. Representado em Haskell como

Q-op ($|a\rangle$, $|b\rangle$) *p*

Onde $|a\rangle$ é o valor quântico de entrada, $|b\rangle$ é o valor quântico de saída e *p* é a probabilidade do mapeamento ocorrer. Desta forma, a porta de Hadamard seria representada da seguinte forma:

$$H\ \{\ (|0\rangle, |0\rangle)\ \tfrac{1}{\sqrt{2}}\ ;\ (|0\rangle, |1\rangle)\ \tfrac{1}{\sqrt{2}}\ ;\ (|1\rangle, |0\rangle)\ \tfrac{1}{\sqrt{2}}\ ;\ (|1\rangle, |1\rangle)\ \tfrac{1}{\sqrt{2}}\ \}$$

Nesta abordagem, surge um problema para a aplicação de portas lógicas em grupos de qubits. Quando se deve aplicar a porta de Hadamard ao primeiro qubit de $|01\rangle$, é necessário criar uma estrutura de abstração que isola o primeiro qubit do estado global. Sabry [Sabry 2003] resolve o problema criando um estrutura de qubit virtual, que pode ser vista como uma tripla *(Valor destacado, Restante, Escopo global)*. Em Haskell, isso é declarado

data Virt a rest global = Virt (QR global) (Adaptor (a, rest) global)

onde *Adaptor* é um tipo que declara funções de transições entre *(a,rest)* e *global*.

Note que não é possível realmente isolar um qubit do escopo em que ele está inserido. Isso se dá uma vez que outros qubits podem estar emaranhados com o qubit destacado, sofrendo, assim, indiretamente as consequências das operações.

3. Cálculo Lambda Quântico de Van Tonder

Esta seção apresenta brevemente a definição do Cálculo Lambda Quântico de Van Tonder [van Tonder 2004].

O Cálculo Lambda, definido por Alonzo Church, é um modelo matemático que descreve computações através da aplicação de funções puras. Sua influência no estudo

de linguagens de programação é imensa, fornecendo os fundamentos para abordagens de programação baseadas em composição de funções e imutabilidade [van Tonder 2004].

No Cálculo Lambda clássico, um termo a ser avaliado é categorizado como uma variável, uma abstração, ou uma aplicação. Variáveis são convencionalmente denotadas por letras quaisquer. Abstrações são representadas como $\lambda x.\ t$ onde x é um parâmetro e t o seu corpo. Por sua vez, um termo de forma $(t1\ t2)$ corresponde a uma aplicação, indicando que o parâmetro da abstração em $t1$ deve ser substituído, em seu corpo, pelo valor de $t2$. Este tipo de substituição é chamado de redução beta (β).

A implementação do Cálculo Lambda recai sobre o conceito de Índices de Bruijn [Pierce 2002]. É definido que cada variável é referida por um índice numérico que indica a distância a qual ela se encontra em relação a sua abstração [Pierce 2002]. Dessa forma, a função identidade, $\lambda x.x$ é simplesmente escrita $\lambda 0$; um termo $\lambda x.\lambda y.x$ é referido como $\lambda\lambda 1$.

A partir dessa estratégia de implementação, podemos definir uma estrutura para representar termos do Cálculo Lambda. Em Haskell, uma possível implementação seria:

```haskell
data Term =
      Var Int
    | Abs Term
    | App Term Term
```

Para a manipulação de qubits e portas lógicas quânticas, é necessário estender a definição do Cálculo Lambda clássico para que contemple esses novos elementos [van Tonder 2004]. Além disso, como previamente mencionado, todas as operações realizadas por portas quânticas são reversíveis se a sequência das operações for armazenada. Sabendo disso, é importante manter o fluxo prévio das reduções, caso se queira aproveitar ao máximo os benefício da computação quântica.

Para Van Tonder [van Tonder 2004], um termo t ao ser reduzido, gera um histórico $\mathcal{H}$ que contém as mudanças que ocorreram no termo. Desta forma, por exemplo, o termo $(\lambda x.Hx)\ |0\rangle$ seria reduzido como se segue:

$$(\lambda x.Hx)\ |0\rangle$$
$$\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\ |\ \mathcal{H} = (H\_)$$

Porém, as regras de redução precisam evitar o emaranhamento entre qubits do estado atual e qubits salvos no histórico como operações anteriores. Para tal, Van Tonder [van Tonder 2004] apresenta uma definição formal dos termos e regras adaptadas, que, além de adicionar qubits e portas lógicas, evita esse tipo de emaranhamento. Esta solução envolve o uso do Cálculo Lambda Linear, que adiciona termos e abstrações lineares e não-lineares [van Tonder 2004].

```haskell
data LLT =
    Var Int
  | NonLinAbs LLT
  | NonLinTerm LLT
  | App LLT LLT
  | LinAbs LLT
  | undefined -- outras constantes
```

Neste contexto, uma abstração não-linear é equivalente à uma abstração clássica. Por outro lado, em uma abstração linear, o argumento aparece uma única vez no corpo da expressão [van Tonder 2004]. Nesse contexto, qubits só devem ser utilizados dentro de termos lineares. Essa distinção é necessária para que qubits não sejam descartados e consequentemente armazenados em $\mathcal{H}$. Uma vez que não há qubits no histórico, é impossível um qubit do termo atual estar entrelaçado com estados salvos [van Tonder 2004].

As regras de redução, ainda sem os qubits e portas lógicas, podem ser implementadas como:

```
reductionRun t =
    case t of
    (App t1 t2) | not (isValue t1) -> App (reductionRun t1) t2
                | not (isValue t2) -> App t1 (reductionRun t2)
    (App (LinAbs t1) v)
                | isValue v -> betaReduct v t1
    (App (NonLinAbs t1) (NonLinTerm v))
                | isValue v -> betaReduct v t1
    a -> a

reduction t =
  let t' = reductionRun t
  in if t' == t
        then t'
        else reduction t
```

4. Implementação do Cálculo Lambda Quântico

Nesta seção apresenta-se uma implementação do Cálculo Lambda quântico e uma adaptação para a execução deste sobre os qubits apresentados na Seção 1.

Durante operações dentro do Cálculo Lambda é necessário, muitas vezes, referir-se a qubits separadamente. Por exemplo, quando se quer aplicar uma porta lógica a apenas um qubit do grupo em questão. Como já mencionado, Sabry dispõe de valores virtuais para a realização desse tipo de separação.

Desta forma, operações definidas por Tonder [van Tonder 2004], tais como

```
let (x, y) = QV (0,1)
in  H x
```

precisam se utilizar de adaptadores que transformam valores virtuais. Isso pode ser compreendido, de forma bastante simplificada, como

```
let t = QV (0,1)
in  H (adapt t)
```

Tal abordagem se mostra suficiente para nossos objetivos.

O uso de uma função de adaptação em um valor virtual gera um novo valor virtual que possui uma tipagem diferente da anterior. Dessa forma, se torna muito custoso manter um termo dependente do tipo dos qubits, pois em um único termo podemos encontrar qubits virtuais cujas tipagens diferem. Em Haskell, uma forma de abstração de tipos

pode ser alcançada utilizando a *typeclass Typeable*. Esses tipos podem ser comparados em tempo de interpretação, o que possibilita a realização de operações seguras quanto ao casamento de tipos.

Desta forma, podemos criar estruturas intermediarias para a representação de qubits, portas quânticas e adaptadores, as quais abstraem a tipagem real. Estes novos tipos são utilizados em funções que definem a aplicação de uma porta lógica em um qubit; a adaptação de um valor virtual para outro; e o produto tensorial entre valores virtuais, representado por : &∗ :.

```
data CnstGate = forall a b.
    (Basis a, Basis b) => CnstGate (Qop a b)
data CnstValue = forall a b c.
    (Basis a, Basis b, Basis c) => CnstValue (Virt a b c)
data CnstAdaptor = forall a b c.
    (Basis a, Basis b, Basis c) => CnstAdaptor (Adaptor (a, b) c)
```

Para a manipulação dos novos tipos apresentados, são implementadas as seguintes funções: *cnstAdapt*, que aplica um adaptador, retornando um valor quântico com base diferente; *cnstApp* que aplica uma porta quântica a um valor; e *cnstTensor*, que define o produto tensorial para constantes. Caso haja uma incompatibilidade entre os tipos das bases das operações ou valores, uma mensagem de erro é enviada.

```
cnstAdapt :: CnstValue -> CnstAdaptor -> CnstValue
cnstAdapt (CnstValue (v :: Virt a rest u))
          (CnstAdaptor (ad :: Adaptor (f1, f2) t))
    = case eqT @a @t of
        Just Refl -> CnstValue (virtFromV v ad)
        _ -> error "Coundʼt match adaptor with value basis"

cnstApp :: CnstGate -> CnstValue -> IO ()
cnstApp (CnstGate (op :: Qop a1 a2))
        (CnstValue (v :: Virt a3 b c))
    = case eqT @a1 @a2 of
      Just Refl -> case eqT @a1 @a3 of
        Just Refl -> app1 op v
        _ -> error "`Op a a` must match `Val a b c`"
      _ -> error "need an `Op a a`, not a general `Op a b`"

cnstTensor :: CnstValue -> CnstValue -> IO CnstValue
cnstTensor (CnstValue (v1 :: Virt a1 b1 c1))
           (CnstValue (v2 :: Virt a2 b2 c2)) =
    do  composed <- virtTensor v1 v2
        return $ CnstValue composed
```

Para contemplar a manipulação dos elementos apresentados na Seção 2, é necessário finalizar a definição dos termos em Haskell. Além disso, por motivos de simplicidade, adiciona-se à definição, variáveis nomeadas (*Def*) e as *keyword let-in* (*Let*). Finalmente, a definição completa dos termos é:

```
data LLT =
```

```
    Var Int
  | NonLinAbs LLT
  | NonLinTerm LLT
  | App LLT LLT
  | LinAbs LLT
  | LValue CnstValue
  | LAdaptor CnstAdaptor LLT
  | LGate CnstGate
  | LLT :&*: LLT
  | Def VarName
  | Let (VarTable LLT) LLT
```

Para a avaliação de expressões contendo elementos quânticos, estende-se a definição da função de redução apresentada na Seção 3 para os seguintes casos, com operações monádicas suprimidas:

```
((LValue v1) :&*: (LValue v2))
    -> LValue (cnstTensor v1 v2)
(LAdaptor ad (LValue v))
    -> reductionRun vt $ LValue $ cnstAdapt v ad
(App (LGate q) t1)
    -> App (LGate q) (reductionRun vt t1)
(LAdaptor ad t1)
    -> LAdaptor ad (reductionRun vt t1)
(Def name) -> defToLLT vt name
(Let vt' ni)
    -> reduction (varAppend vt vt') ni
(v1 :&*: v2)
    -> reductionRun vt v1 :&*: reductionRun vt v2
```

O algoritmo de Deutsch é um importante resultado para a computação por ser um dos mais simples exemplos a demonstrar a superioridade da abordagem Quântica [Yanofsky and Mannucci 2008]. O algoritmo resolve o problema de descobrir se uma dada função é constante para todas as entradas. Simplificadamente, essa verificação é realizada através da avaliação da função em uma entrada em máxima sobreposição, uma vez que isso garante a avaliação da função sobre todos os valores básicos simultaneamente.

Abaixo, a implementação do algoritmo de Deutsch sobre os termos, utilizando a porta *Cnot* como função de entrada:

```
deutsch :: LLT
deutsch =
    Let [("x1", LGate (CnstGate hGate) `App` LValue cnst1),
         ("x2", LGate (CnstGate hGate) `App` LValue cnst0)] $
    Let [("x", LGate cnstCnot `App` (Def "x1" :&*: Def "x2"))] $
          LGate cnstH `App` LAdaptor adapt1 (Def "x")
```

Note que não é possível descrever o algoritmo como

```
deutsch :: LLT
deutsch =
    Let [("x1", LGate (CnstGate hGate) `App` LValue cnst1),
```

```
4           ("x2", LGate (CnstGate hGate) `App` LValue cnst0)] $
5     Let [("x", LGate cnstCnot `App` (Def "x1" :&*: Def "x2"))] $
6           LGate cnstH `App` (Def "x1")
```

por conta da avaliação preguiçosa. Isso se dá porque o segundo termo *Let* jamais será avaliado, implicando que a porta *Cnot* não é aplicada ao grupo de qubits.

O seguinte código é válido para retomar o valor da base inicial, caso queiramos voltar a realizar aplicações com ambos os qubits. Porém, leva a outro problema: *x2* está relacionado a *x1* por dois meios: um através do valor virtual que foi adaptado, e outro pelo novo produto tensorial. Além disso, *x2* desafia a definição das abstrações lineares apresentadas na Seção 3, uma vez que é encontrado duas vezes dentro da expressão.

```
1 deutsch :: LLT
2 deutsch =
3     Let [("x1", LGate (CnstGate hGate) `App` LValue cnst1),
4          ("x2", LGate (CnstGate hGate) `App` LValue cnst0)] $
5     Let [("x", LGate cnstCnot `App` (Def "x1" :&*: Def "x2"))] $
6     LGate cnstH `App` LAdaptor adapt1 (Def "x") :&*: Def "x2"
```

5. Conclusões e Trabalhos Futuros

Neste trabalho apresentou-se uma definição e implementação do Cálculo Lambda Quântico para uma implementação específica de qubits na linguagem Haskell.

Sabe-se, porém, das limitações da abordagem apresentada. Primeiramente, não foi apresentado um método válido de reversibilidade de operações como citado na Seção 2 e 3. Também é importante salientar a limitação de expressividade dentro dos termos, tendo em vista que o uso direto de adaptadores e verificações de tipos exige um tratamento específico para cada caso.

Como trabalhos futuros, pretende-se ampliar o poder de articulação dos termos e variáveis. A constante necessidade de aplicação de funções a qubits virtuais representa um desafio na construção deste projeto. Desta forma, uma implementação de uma interface intermediária para a manipulação de grupos de qubits se faz necessária para uma escrita mais amigável dos termos.

Em síntese, o desenvolvimento de linguagens de programação voltadas para a computação quântica se apresenta como um pilar fundamental no avanço tecnológico. Esta conclusão reitera que, ao proporcionar abstrações e ferramentas que se alinham às especificidades do mundo quântico, abrem-se portas para novas descobertas e transformações na área, além de permitir uma abordagem de alto nível e consequentemente mais acessível.

Referências

Pierce, B. C. (2002). *Types and Programming Language*. The MIT Press, 1st edition.

Sabry, A. (2003). Modeling quantum computing in haskell.

van Tonder, A. (2004). A lambda calculus for quantum computation.

Yanofsky, N. S. and Mannucci, M. A. (2008). *Quantum Computing for Computer Scientists*. Cambridge University Press, 1st edition.