UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

QJAVA: SETAS QUÂNTICAS EM JAVA

DISSERTAÇÃO DE MESTRADO

Bruno Crestani Calegaro

Santa Maria, RS, Brasil

2013

QJAVA: SETAS QUÂNTICAS EM JAVA

Bruno Crestani Calegaro

Dissertação apresentada ao Curso de Mestrado Programa de
Pós-Graduação em Informática (PPGI), Área de Concentração em
Computação, da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profª. Dra. Juliana Vizzotto

Santa Maria, RS, Brasil

2013

Crestani Calegaro, Bruno

QJava: Setas Quânticas em Java / por Bruno Crestani Calegaro. – 2013.

90 f.: il.; 30 cm.

Orientadora: Juliana Vizzotto

Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2013.

1. Computação Quântica. 2. Java. 3. Mônadas. 4. Setas. 5. ANTLR. 6. Analisar sintático. I. Vizzotto, Juliana. II. Título.

E-mail: bcalegaro@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

QJAVA: SETAS QUÂNTICAS EM JAVA

elaborada por
Bruno Crestani Calegaro

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Juliana Vizzotto, Dra.
(Presidente/Orientadora)

André du Bois, Dr. (UFPEL)

Eduardo Piveta, Dr. (UFSM)

Santa Maria, 27 de Agosto de 2013.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer à pessoa mais importante da minha vida, minha amada noiva Bruna. Obrigado por estar sempre ao meu lado fazendo o possível para melhorar meus dias. Ninguém melhor para aturar meu gênio, e vice-versa, do que essa pessoa que sempre traz alegria no final cansativo de cada dia. Agradeço pelo amor, carinho e atenção.

Aos meus pais, João Getúlio e Maria Idanir, que me apoiaram e me incentivaram a atingir meus objetivos. Não se sintam distantes de mim apenas pelo fato de não morarmos mais juntos, isso é apenas uma ilusão geográfica. Se tudo der errado, eu volto para casa, mas esperamos que não.

Aos meus irmãos, obrigado pelas lições de vida, boas ou ruins, que passamos juntos. Quando o irmão mais velho se transforma em pai, percebemos que não somos mais crianças, e que agora temos que servir de exemplos para a nova geração tendo que assumir uma postura mais adulta. Mas de qualquer forma isso não impede que venham muitas festas e rock and roll!

Agradeço a todos os meus amigos e colegas que me ajudaram durante essa trajetória. Em especial ao Rodrigo Tonin, pela parceira no mestrado, com nossos famosos coofee breaks, e sempre discutindo temas interessantes. Ao Vinícius Alves, super ausente, mas ao mesmo tempo demostrando muita garra em alcançar seus objetivos, servindo de imenso exemplo. Aos, não tão distantes, Marcus Garcia e João Gabriel Piccoli, por se esforçarem em voltar para Santa Maria apenas para visitar os amigos e sendo a maior prova de que a amizade vale por toda uma vida.

À minha orientadora Juliana K. Vizzotto, por ser compreensiva e paciente em me orientar em momentos conturbadores mesmo tendo que dar atenção ao pequeno Caio. Aos demais professores e colegas do PPGI que de alguma forma contribuíram para essa experiência. Agradeço também à UFSM e à CAPES pelo apoio durante a realização deste trabalho.

“Qualquer coisa que você faça será insignificante,
mas é muito importante que você faça.”
— MAHATMA GANDHI

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

QJAVA: SETAS QUÂNTICAS EM JAVA

AUTOR: BRUNO CRESTANI CALEGARO
ORIENTADORA: JULIANA VIZZOTTO

Local da Defesa e Data: Santa Maria, 27 de Agosto de 2013.

A computação quântica é uma tecnologia emergente e, atualmente, encontra-se no desafio de desenvolver linguagens de programação segundo as regras da mecânica quântica para a criação, análise, modelagem e simulação de algoritmos quânticos de *alto nível*.

Particularmente, o foco é na investigação de *novos modelos semânticos* para elaborar linguagens de programação para a computação quântica.

Nesse contexto, uma das alternativas é utilizar um modelo semântico de mônadas e setas capaz de abstrair tanto estados quânticos puros quanto mistos e ainda expressar operações de medidas. Esse modelo foi implementado como uma biblioteca para a linguagem funcional Haskell, contudo nem todo programador está familiarizado.

Dessa forma, o presente trabalho objetiva oferecer uma ferramenta universal de alto nível para a programação quântica, apresentando uma biblioteca para o Java.

Essa biblioteca foi implementada utilizando os novos recursos de closures presentes na versão 8 do JDK (Java Development Kit), já disponibilizados na prévia de desenvolvedores.

Além disso, esse trabalho apresenta uma sintaxe específica para a biblioteca para facilitar a elaboração de algoritmos quânticos de forma clara e estruturada, descrita de uma maneira similar a notação-do do Haskell. A sintaxe criada opera em conjunto com um tradutor desenvolvido com a ferramenta ANTLR.

Palavras-chave: Computação Quântica. Java. Mônadas. Setas. ANTLR. Analisar sintático.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

QJAVA: QUANTUM ARROWS IN JAVA

AUTHOR: BRUNO CRESTANI CALEGARO
ADVISOR: JULIANA VIZZOTTO

Defense Place and Date: Santa Maria, August 27st, 2013.

Quantum computing is an emerging technology that, currently, has the challenge of developing programming languages, according to the rules of quantum mechanics, to support the creation, analysis, modeling and simulation of *high-level* quantum algorithms. Particularly, the focus is on the investigation of *new semantic models* to develop programming languages for quantum computing. In this context, one alternative is to use the semantic model of monads and arrows that abstracts both pure and mixed quantum states and also can express measures. This model however, was originally implemented as a library for the functional language Haskell, which not every programmer is family. This way, this study aims to provide a universal tool for high-level quantum programming, providing a library for Java. This library was implemented using the new features of closures present in the version 8 of the JDK (Java Development Kit), already available in developers preview. In addition, we present a specific syntax for the library to facilitate the development of quantum algorithms with a clearly structured notation. This syntax is described in a notation similar to the do-notation of Haskell and operates in conjunction with a parser implemented by ANTLR tool.

Keywords: Quantum Computing, Java, Monads, Arrows, ANTLR, Parser.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CQ	Computação Quântica
Qubit	Bit Quântico
Cbit	Bit Clássico
EPR	Einstein, Podolsky e Rosen
QJava	Quantum Java
ANTLR	ANother Tool for Language Recognition

SUMÁRIO

1 INTRODUÇÃO

A *Computação Quântica* (CQ) é uma tecnologia emergente, com potencial computacional maior que os computadores clássicos, que necessita de uma abordagem diferente, tanto em hardware quanto em software. Essencialmente, a CQ baseia-se no pressuposto de que um computador pode acessar e manipular uma informação codificada em um sistema físico quântico. Assim, o processamento da informação é realizado fisicamente através de um sistema físico quântico de escala atômica (MERMIM, 2007). Em suma, a CQ é caracterizada por um paradigma de processamento de informação onde a informação/dados (e possivelmente controle de fluxo) podem ser codificados no estado de um sistema físico *quântico*.

A primeira visão de um computador baseado nos efeitos da mecânica quântica foi teoricamente introduzida por Richard Feynmam (1982). Um pouco depois, David Deutsch (1985) provou formalmente, por meio de uma máquina de Turing quântica, que um computador quântico não poderia ser simulado em tempo polinomial através de um computador clássico.

Em 1994, através de um computador quântico hipotético, Peter Shor, o pioneiro na computação quântica, criou um algoritmo quântico para resolver o problema da fatoração de números inteiros grandes em tempo polinomial (SHOR, 1994), enquanto os algoritmos clássicos levam tempo exponencial (POMERANCE, 1996). A aplicabilidade prática do algoritmo de Shor em um computador quântico real permitiria a solução rápida de problemas como algoritmos baseados em fatoração, como a criptografia RSA (RIVEST; SHAMIR; ADELMAN, 1977), curvas elípticas e algoritmos discretos. De fato, a invenção desse algoritmo motivou o tema da computação quântica e estimulou demais pesquisadores na investigação quanto à criação de computadores quânticos práticos e de novos algoritmos quânticos aplicados a áreas diferentes.

Os algoritmos quânticos são descritos através de formalismos matemáticos e exigem conhecimentos avançados de matemática (álgebra linear) e física (mecânica quântica) (YANOFSKY; MANNUCCI, 2008). Existe também uma representação simplificada que é, em uma analogia a computação clássica, um circuito lógico onde cada transformação (operação lógica) sobre um bit do estado quântico é representada como um porta quântica (*quantum gate*). Do ponto de vista da Computação, dentre as alternativas disponíveis, essa abordagem de *baixo nível* é a que mais se assemelha com a realidade dos programadores.

Sendo assim, atualmente os pesquisadores da área de CQ têm seus esforços voltados ao

desenvolvimento de linguagens de programação de modo dar suporte a criação, análise, modelagem e simulação de algoritmos quânticos com uma abordagem de *alto nível*. Particularmente, o foco é na investigação de *novos modelos semânticos* para elaborar linguagens de programação para a computação quântica.

No trabalho de Vizzotto et al (2006), os autores apresentam um modelo semântico para computação quântica, implementando na linguagem funcional Haskell, usando *mônadas* para representar estados quânticos puros e operações reversíveis, e *setas* para estados mistos e superoperadores (AHARONOV; KITAEV; NISAN, 1998). Mônada (MOGGI, 1989) é um conceito da teoria das categorias, utilizado para modelar efeitos colaterais no contexto de linguagens de programação funcional puras, tais como estado global e entrada/saída. E *Seta* (Arrow) (HUGHES, 2000) é uma generalização de mônada utilizada para criar abstrações mais gerais, como por exemplo, uma abstração para processar mais de uma entrada ou apenas parte dela.

O modelo semântico de mônadas e setas permite modelar de maneira elegante as características ímpares da computação quântica (como superposição e medidas), no contexto de linguagens de programação tradicionais (não quântica). Esse modelo semântico de mônadas e setas pode ser utilizado para se criar uma linguagem de programação quântica de propósito geral capaz de expressar algoritmos quânticos, incluindo operações reversíveis e medidas, com a conexão entre dados clássicos e quânticos (VIZZOTTO; ROCHA COSTA; SABRY, 2008).

O interesse na pesquisa do presente trabalho surgiu a partir do intuito de oferecer uma ferramenta universal de alto nível para a programação quântica, voltada ao público geral da Computação. Para essa tarefa foi escolhida a linguagem de programação que mais se assemelha com a realidade dos programadores, a linguagem Java.

Java, por sua vez, é uma linguagem de propósito geral, orientada a objetos, segura, multiplataforma e capaz de desenvolver software de alta performance. Recentemente ela incorporou em suas funcionalidades, através do *Lambda Project* (ORACLE, 2013a), funções anônimas (closures), o que permite agora a implementação de mônadas e setas. Assim, o presente estudo apresenta uma biblioteca para programação quântica criada para essa linguagem, chamada *QJava*, baseada no modelo de setas quânticas.

No decorrer do desenvolvimento da referida biblioteca, foi observado que era utilizada uma sintaxe extensa devido à composição de funções com closures. Em razão disso, foi produzido também um tradutor, *parser*, utilizando a ferramenta ANTLR (PARR, 2007) para atuar

sobre uma sintaxe similar a notação-do presente no Haskell (PATERSON, 2001). Essa notação é uma forma de estruturar as computações de um dado estado global sequenciando as computações uma por uma como sentenças e, opcionalmente, atribuindo o valor dessas computações à variáveis que podem ser referenciadas em outras sentenças. Assim, seguindo a abordagem do interpretador no Haskell, o tradutor desenvolvido lê o código de um bloco escrito na nova sintaxe e monta automaticamente o código extenso necessário para a composição de setas fazendo o uso das chamadas de funções da biblioteca QJava.

Por fim, entende-se que a pesquisa é viável. Sua aplicação permite a elaboração de algoritmos quânticos no contexto da linguagem de programação orientada a objeto Java, facilitando a compreensão e desenvolvimento de algoritmo quânticos para os programadores.

1.1 Trabalhos Relacionados

A utilização da linguagem Java na programação quântica não é nenhuma novidade. Existem diversos simuladores quânticos em Java, como por exemplo, o *Zeno* (CABRAL et al, 2004), uma ferramenta gráfica para projeto e simulação de circuitos quânticos, o *jQuantum* (VRIES, 2010), um programa que simula um computador quântico, oferecendo uma interface gráfica para desenhar circuitos quânticos mas também para ilustrar o estado de um registrador quântico, e o *Quantomatic* (KISSINGER et al, 2012), uma ferramenta para descrever as computações quântica utilizando uma linguagem baseada em gráficos.,

Tais trabalhos utilizam sempre uma representação de *baixo nível*, utilizando circuitos quânticos para representar os algoritmos. Sendo assim, o presente trabalho visa preencher essa lacuna, disponibilizando para a linguagem Java uma ferramenta que ofereça primitivas de mais alto nível como chamadas de funções e atribuições, de modo que os algoritmo quânticos possam ser representados de uma maneira mais similar a programação.

1.2 Objetivos e Contribuições

O objetivo geral do presente trabalho é criar uma biblioteca para a computação quântica com a linguagem de programação Java, usando o modelo semântico de setas quânticas, e um tradutor para uma sintaxe similar a notação-do. Como objetivos específicos, podem ser citados:

- Especificar a arquitetura geral do modelo semântico de setas quânticas em uma linguagem orientada a objetos

- Prototipar setas quânticas e demais recursos necessários para a implementação
- Verificar a integridade do projeto (operações quânticas, medidas, etc) através de exemplos
- Definir as regras para o tradutor segundo uma gramática paro o ANTLR
- Realizar testes de tradução e de verificação da consistência das operações
- Publicação dos resultados

Nesse sentido, as contribuições deste trabalho são:

- Melhor entendimento de computação quântica, mônadas e setas
- Desenvolvimento de uma biblioteca para a programação quântica em Java
- Desenvolvimento de um tradutor para essa biblioteca

1.3 Organização do Texto

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os principais conceitos da computação quântica, bem como alguns algoritmos quânticos. O objetivo desse capítulo, além de servir como fundamentação teórica para este estudo, é oferecer como fonte de pesquisa para futuros discentes interessados no assunto.

São revisados no Capítulo 3, de forma mais abrangente, os principais conceitos sobre cálculo-λ, mônadas e setas.

O Capítulo 4 tem como principal conteúdo informar os modelos semânticos que serão usados para manipular a informação em estados quânticos.

O Capítulo 5 descreve a biblioteca *QJava*. Além da estrutura, são apresentadas as funcionalidades e características de cada componente. Por fim, seguem exemplos de algumas operações quânticas fazendo o uso da biblioteca.

Uma introdução do funcionamento das transformações de programas, bem como a apresentação da ferramenta ANTLR, é feita no Capítulo 6. Posteriormente, são definidas regras da gramática para uma sintaxe mais sofisticada de setas e os modelos (*templates*) que serão utilizados pelo tradutor. Ainda, são demostrados como exemplos da biblioteca QJava e o tradutor, a implementação dos Algoritmos de Toffoli e da Teleportação.

A conclusões e os trabalhos futuros encontram-se Capítulo 7,

2 COMPUTAÇÃO QUÂNTICA

A Computação Quântica é um paradigma de processamento de informação, onde as informações, dados e possivelmente o controle de fluxos, podem ser codificados no estado de um sistema físico *quântico*. Neste capítulo, será feita uma breve revisão do formalismo necessário para compreender os princípios básicos da programação quântica. A revisão apresentada aqui está baseada no livro *Quantum Computation and Quantum Information* (NIELSEN; CHUANG, 2000).

2.1 Bits Quânticos

A unidade básica de informação usada na computação clássica é o bit, um sistema físico clássico binário. O bit pode ser representado como um escalar que pertence ao conjunto $\{1, 0\}$. Em contraste, na computação quântica a unidade básica de informação é representada pelo *bit quântico*, ou qubit, um sistema físico *quântico* binário. O qubit é um vetor usualmente representado usando a notação de Dirac, chamada *braket* [1]:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

Essa notação possui como benefício rotular as bases do espaço vetorial de forma explícita. Nesse exemplo, o vetor ψ é descrito em função dos estados básicos $|0\rangle$ e $|1\rangle$, que podem ser explicados como uma analogia ao bit clássico, ou seja, formam um sistema de dois níveis e são uma base ortonormal para o espaço vetorial onde está o qubit (geralmente chamado de padrão ou bases computacionais). Os coeficientes, também conhecidos como *amplitudes de probabilidade*, α e β, são números complexos, tal que $|\alpha|^2 + |\beta|^2 = 1$. Em outras palavras, o qubit pode ser formalizado como um vetor em um espaço vetorial complexo bidimensional (Espaço de Hilbert), com norma (tamanho) igual a 1, cuja representação geométrica é ilustrada na Figura 2.1.

O qubit pode ser visto como um vetor coluna:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Nessa representação, é importante notar que existe uma ordem para as bases vetoriais

[1] O nome *braket* vem da convenção de que uma coluna do vetor é chamada de "ket", indicada pelo símbolo $|\ \rangle$, e uma linha do vetor é chamada de "bra", indicada pelo símbolo $\langle\ |$.

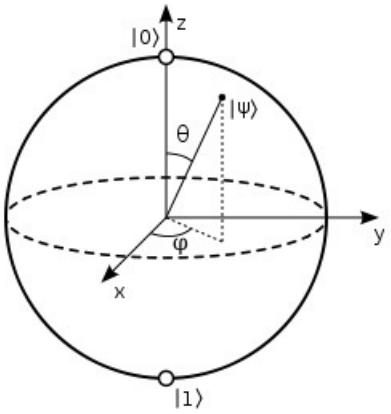


Figura 2.1 – Esfera de Bloch

relevante: o coeficiente para $|0\rangle$, que nesse caso específico é α, sempre aparece na primeira linha (a linha para $|0\rangle$) e o coeficiente para $|1\rangle$, que nesse caso específico é β, na segunda linha (a linha para $|1\rangle$). Essa convenção é ilustrada a abaixo.

$$\begin{matrix} |0\rangle \\ |1\rangle \end{matrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Como exemplo, o bit clássico 0 pode ser representado pelo estado básico $|0\rangle = 1|0\rangle + 0|1\rangle$ e o bit clássico 1 pode ser representado pelo estado básico $|1\rangle = 0|0\rangle + 1|1\rangle$. A representação desses dois estados como vetores em coluna pode ser vista da seguinte forma, respectivamente:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Qualquer outro estado com valores diferentes para α e β é dito estar em uma *superposição quântica* de $|0\rangle$ e $|1\rangle$, por exemplo o estado $1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$.

A interpretação das amplitudes de probabilidade α e β pode ser dada pelo seguinte: quando se *interage* ou se *mede* um estado quântico como $\alpha|0\rangle + \beta|1\rangle$ se obtém o estado $|0\rangle$ com probabilidade $|\alpha|^2$ e o estado $|1\rangle$ com probabilidade $|\beta|^2$.

Na prática, utilizam-se estados com vários qubits. Por isso se deve entender também como compor estados quânticos com dois ou mais qubits. Assim, para compreender a composição de estados quânticos, novamente em analogia com o bit clássico, considera-se um estado com dois bits. Tal estado pode estar em quatro valores possíveis: 00, 01, 10, 11. Logo, um estado quântico com dois qubits é uma combinação linear desses estados clássicos:

$$|\sigma\rangle = \alpha|00\rangle + \gamma|01\rangle + \delta|10\rangle + \beta|11\rangle \tag{2.2}$$

Esse estado de dois qubits pode também ser representado por um vetor coluna:

$$\begin{pmatrix} \alpha \\ \gamma \\ \delta \\ \beta \end{pmatrix}$$

Para estados com mais de um qubit também existe uma ordenação para as bases vetoriais: o coeficiente para $|00\rangle$, que nesse caso específico é α, sempre aparece na primeira linha (a linha para $|00\rangle$), o coeficiente para $|01\rangle$, que nesse caso específico é γ, na segunda linha (a linha para $|01\rangle$), o coeficiente para $|10\rangle$, que nesse caso específico é δ, na terceira linha (a linha para $|10\rangle$), e o coeficiente para $|11\rangle$, que nesse caso específico é β, na última linha.

Além disso, se $q = \alpha|0\rangle + \beta|1\rangle$ e $p = \gamma|0\rangle + \delta|1\rangle$ são dois bits quânticos independentes, então pode-se formar um estado composto usando o *produto tensorial* de seus espaços vetoriais, $\otimes$, definido como a seguir:

$$q \otimes p = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

Existem ainda algumas combinações de bits quânticos que não estão na forma $q \otimes p$. Esse tipo de estado *combinado* não pode ser descrito usando operações de produto tensorial. Tais estados são chamados de *estados emaranhados*. Por exemplo, a seguinte equação:

$$1/\sqrt{2}|00\rangle + 1/\sqrt{2}|11\rangle \tag{2.3}$$

não está na forma $q \otimes p$, para qualquer q e p. Essa característica específica de estados emaranhados permite que dois ou mais qubits estejam de alguma forma *ligados*, ou seja, não podem ser separados. Assim, uma operação de medida sobre o estado de um dos qubit influenciará imediatamente a sua contra-parte pois existe uma correlação muito forte entre seus subsistemas quânticos. Essa propriedade de estados emaranhados é descrita através do paradoxo EPR (EINSTEIN; PODOLSKY; ROSEN, 1935) por isso é comum chamar os membros dos estados emaranhados de par EPR. Os exemplos de estados emaranhados mais simples são descritos como os quatro possíveis valores da Tabela 2.1.

2.2 Operações Quânticas

Existem dois tipos de operações sobre bits quânticos: *transformações unitárias* e *medidas* (observações). O primeiro tipo corresponde a operações reversíveis que modificam o estado

$$|\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$$

$$|\beta_{01}\rangle = \frac{|01\rangle+|10\rangle}{\sqrt{2}}$$

$$|\beta_{10}\rangle = \frac{|00\rangle-|11\rangle}{\sqrt{2}}$$

$$|\beta_{11}\rangle = \frac{|01\rangle-|10\rangle}{\sqrt{2}}$$

Tabela 2.1 – Pares EPR

de um qubit sem perda de informação, o segundo, por sua vez, *observa* o estado para descobrir seu valor e causa um *colapso* no estado quântico, tornando impossível a recuperação do estado original após a observação.

2.2.1 Transformações Unitárias

Uma transformação unitária é capaz de mudar o estado de uma unidade vetorial sem perda de informação. Logo, como um qubit é uma unidade vetorial dentro de um espaço vetorial complexo de duas dimensões (normalmente gerado pelas bases computacionais $|0\rangle$ e $|1\rangle$), são usadas transformações unitárias para representar as operações reversíveis sobre os estados quânticos. Sendo assim, como um qubit é normalmente representado por um vetor coluna, uma transformação unitária é geralmente representada por uma matriz unitária [2]. Desse modo, para um estado de *n* qubits, é necessário uma matriz de tamanho $2^n \times 2^n$.

Com o estado $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$, a aplicação de uma transformação unitária T pode ser representada como a multiplicação normal entre matrizes e vetores:

$$T|\phi\rangle = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

A seguir, apresenta-se como exemplo de transformação unitária agindo sobre um qubit, onde deve ser considerada a transformação quântica, trocando as amplitudes de probabilidade dos coeficientes $|0\rangle$ e $|1\rangle$, a chamada operação *NOT*. A aplicação do *NOT* quântico em um dado estado quântico $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, deve retornar: $|\psi'\rangle = \beta|0\rangle + \alpha|1\rangle$. Assim, a *matriz unitária*

[2] Uma matriz unitária é uma matriz quadrada ($n \times n$) complexa, U, satisfazendo a condição $U^\dagger U = UU^\dagger = I_n$, tal que $U^\dagger$ é o transposto conjugado (também chamado de adjunto Hermitiano) de U.

capaz de representar a transformação quântica *NOT* é a seguinte:

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Intuitivamente, a primeira coluna dessa matriz de tamanho 2×2 expressa o que ocorre com o vetor de entrada $|0\rangle$, e a segunda, o que ocorre com o vetor de entrada $|1\rangle$. Logo, a aplicação dessa operação sobre $\alpha|0\rangle + \beta|1\rangle$ resulta em

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$$

isto é, $\beta|0\rangle + \alpha|1\rangle$.

Outra importante transformação unitária agindo sobre um qubit é a operação quântica *Hadamard*, que gera um estado quântico em *superposição coerente*:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Pode-se notar que a regra das colunas é a mesma: a primeira expressa o que acontece com o vetor de entrada $|0\rangle$, retornando $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, e a segunda demostra o que ocorre com o vetor de entrada $|1\rangle$, retornando $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Analisando o resultado, pode-se dizer que essa operação gera um qubit em uma *superposição coerente*, onde *coerente* significa que o estado tem a mesma amplitude de probabilidade para $|0\rangle$ e para $|1\rangle$, ou seja, $\frac{1}{\sqrt{2}}$.

Existem ainda outras operações importantes, para citar uma delas, tem-se a operação que modifica o fator fase do qubit. Na mecânica quântica, o fator fase é um coeficiente complexo que multiplica o vetor $|\psi\rangle$. Em termos clássicos, observa-se que o fator fase de um qubit não tem significado físico. Porém, este fator fase é uma quantidade importante ao ser considerada a *interferência*. A transformação de troca de fase é descrita como:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Há também operações quânticas atuando sobre mais de um qubit. Um exemplo interessante é a operação do *NOT* controlado, chamada de *CNOT*. Essa transformação quântica controlada exige uma matriz 4×4 e opera da seguinte forma: o primeiro qubit é de chamado de controlador, pois o *CNOT* só aplica o *NOT* no segundo se o primeiro for igual a 1, caso contrário, o segundo qubit não é alterado.

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Novamente, é interessante notar a regra de colunas. A pequena diferença aqui é no estado de dois qubits: a primeira coluna expressa o que acontece se o vetor de entrada for $|00\rangle$, a segunda, o que ocorre se o vetor de entrada for $|01\rangle$. A terceira coluna diz respeito quando o vetor de entrada for $|10\rangle$, e, por fim, a última coluna, se refere ao que ocorre se o vetor de entrada for $|11\rangle$. Diante da entrada estar em um estado de superposição, a saída deverá ser uma combinação de cada caso. Observa-se também, a existência de uma ordem para os coeficientes nas linhas (onde cada linha representa uma base vetorial), que é idêntica aquela explicada nos exemplos anteriores.

É importante notar que a *condição de ortogonalidade* se aplica a cada coluna, já que todas as operações quânticas devem ser unitárias/reversíveis. Isto é, cada coluna da matriz deve representar uma base ortogonal vetorial com respeito às outras. Isso significa que, a soma dos módulos dos coeficientes de probabilidade elevados ao quadrado de cada a coluna, deve ser sempre igual a 1.

2.2.2 Medidas

Em suma, as transformações unitárias caracterizam um tipo de *rotação reversível* no vetor que representa um estado de sistema quântico. Já as *medidas*, representam outro tipo de transformação no vetor. A medida é a única operação capaz de extrair alguma informação clássica de um estado quântico. Contudo, elas causam um *colapso* nesse estado, tornando impossível recuperar os valores originais, o que faz com que a medida seja a única operação não reversível da computação quântica.

Essencialmente, a medida em um estado como $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, resultaria em $|0\rangle$ (ou simplesmente 0), com a probabilidade $|\alpha|^2$, ou em $|1\rangle$ (ou simplesmente 1), com a probabilidade $|\beta|^2$. Tratando-se de um estado quântico modificado após uma medida, após a medir o estado ψ, o novo estado será $|0\rangle$ se o valor observado for 0 ou $|1\rangle$ se o valor observado for 1. Dessa forma, imediatamente após a medida, não será mais possível recuperar o estado original, especificamente quanto aos valores dos coeficientes α e β.

No exemplo anterior, explicou-se a medida em uma *base computacional* $|0\rangle$ e $|1\rangle$. Entretanto, um estado quântico poderá ser medido considerando qualquer base ortonormal para um qubit. Por exemplo, os estados $|+\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle)$ e $|-\rangle = 1/\sqrt{2}(|0\rangle - |1\rangle)$ também podem formar uma base ortonormal para o qubit. Assim, reescreve-se o estado nessas bases como: $\alpha'|+\rangle + \beta'|-\rangle$. Logo, uma medida em um qubit poderá ser apreciada nessas bases, resultando

em $|+\rangle$ com probabilidade $|\alpha'|^2$, e $|-\rangle$ com probabilidade $|\beta'|^2$. Considere a medição de um estado $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ nas bases $|+\rangle$, $|-\rangle$. Primeiro expressa-se $|\psi\rangle$ nas bases $|+\rangle$, $|-\rangle$:

$$\begin{aligned} |\psi\rangle &= \alpha|0\rangle + \beta|1\rangle \\ &= \alpha\tfrac{1}{\sqrt{2}}(|+\rangle + |-\rangle) + \beta\tfrac{1}{\sqrt{2}}(|+\rangle - |-\rangle) \\ &= \tfrac{1}{\sqrt{2}}((\alpha+\beta)|+\rangle + (|\alpha-\beta)|-\rangle). \end{aligned}$$

Em seguida, a probabilidade de medir $|+\rangle$ é $|\frac{1}{\sqrt{2}}(\alpha+\beta)|^2 = |\alpha+\beta|^2/2$ e a probabilidade de medir $|-\rangle$ é $|\alpha+\beta|^2/2$.

A medida também é possível em um estado quântico composto. Considerando que dois qubits estão em um estado $|\psi\rangle$ e são medidos, a probabilidade do primeiro qubit estar em um estado i, e do segundo estar em j é $P(i,j) = \langle ij|\psi\rangle$, onde $\langle|\rangle$ é a operação tradicional do produto interno de um espaço vetorial. O produto interno retorna uma quantidade escalar para um par de vetores, onde formalmente, é dado pela multiplicação de um vetor linha por um vetor coluna. Assim, após a medida, esse estado de dois qubits é dado como $|\psi'\rangle = |ij\rangle$.

Não obstante, ainda poderá ser medido apenas o primeiro qubit de um sistema quântico de dois estados. Nesse caso, o resultado é o mesmo que medir ambos qubits, isto é, a probabilidade de o primeiro estar no estado i é $P(i) = \sum_j^{0,1}\langle ij|\psi\rangle$. Assim, o novo estado após a medida do primeiro qubit, vai consistir em uma superposição dos termos que são consistentes com o resultado dela, normalizados. Essa medida parcial, representa uma *projeção* do vetor dentro do subespaço gerado pelos estados $|ij\rangle$, tal que $j \in \{0,1\}$.

Emaranhamento também pode ser explicado usando medidas. Considerando o estado da Equação 2.3, ao aplicar uma medida no primeiro qubit, o estado do outro qubit é dependente do resultado dessa medida. Por exemplo, com a probabilidade $\frac{1}{2}$ obtém-se $|00\rangle$ como resultado da medida, e nesse caso, o estado após a medida será $|00\rangle$. Essa característica prevê o estado após a medida de forma útil na computação quântica. Diversos algoritmos quânticos usufruem desse recurso, como no Algoritmo Quântico da Teleportação, onde estados emaranhados são fundamentais para o funcionamento do algoritmo.

2.3 Matrizes de Densidade e Super-Operadores

Na mecânica quântica existe o chamado *estado puro* que é aquele estado quântico que é representado por apenas um vetor dentro de um espaço de Hilbert, digamos ψ. Mas também existe o chamado *estado misto*, que é aquele estado quântico que está em uma perspectiva estatística de diferentes vetores estados. Por exemplo, uma estado quântico que tem 50% de

probabilidade de estar em um vetor estado ψ_1 e 50% de probabilidade de estar em um estado ψ_2 é um estado misto.

A representação de um estado quântico como um estado *puro* nem sempre é o suficiente para demostrar o comportamento de um qubit, como por exemplo, um estado após uma medida. Para tanto devem ser usados estados *mistos*. Contudo, o modelo tradicional da computação quântica está limitado a representar apenas estados como vetores puros.

Dessa forma, para representar estados mistos deve ser utilizado um modelo mais generalizado para a computação quântica, baseado em *matrizes de densidade* para representar os estados computacionais e *super-operadores* para as computações quânticas (AHARONOV; KITAEV; NISAN, 1998).

Seguindo o formalismo das matriz de densidade, um estado quântico $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ pode ser representado pelo *produto externo* do vetor estado que o representava, de tal forma que as amplitudes de probabilidade da matriz se transformam em uma espécie de distribuições de probabilidade dos vetores estados usados no produto externo. Por exemplo, os estados $|\rho_1\rangle = 1|0\rangle + 0|1\rangle$, $|\rho_2\rangle = 0|0\rangle + 1|1\rangle$ e $|\rho_3\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ podem ser descritos como as seguintes matrizes de densidade:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}$$

A grande vantagem de se usar matriz de densidade é que elas podem representar tanto estados *puros* como estados *mistos*, o que torna possível expressar medidas. Por exemplo, uma medida no estado $|\rho_3\rangle$ resulta $|\rho_1\rangle$ com probabilidade $\frac{1}{2}$ ou $|\rho_2\rangle$ com probabilidade $\frac{1}{2}$. Essa informação não pode ser descrita usando vetores mas sim com uma matriz de densidade. Uma matriz de densidade capaz de representar essa medida é uma matriz que representa um estado *misto* cujos coeficientes correspondem a soma (e normalização) dos dois resultados obtidos pela observação. Nesse caso específico, uma matriz de densidade capaz de representar essa medida pode ser vista como:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1/2 \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

sendo as duas primeiras matrizes correspondem aos resultados possíveis da medida em $|\rho_3\rangle$, e última correspondente aos coeficientes da soma e normalização desses resultados.

Neste modelo de computação quântica baseado em matrizes de densidade, é necessário que todas as transformações unitárias sejam convertidas em operadores lineares compatíveis com as matrizes de densidade. Na física, esses operadores são chamados de *super-operadores*.

Os super-operadores são um tipo de operador linear agindo sobre um espaço vetorial de operadores lineares. Dessa forma, um super-operador pode ser obtido através da conversão dos operadores lineares na forma de vetor.

2.4 Algoritmos Quânticos

Em geral, utiliza-se a notação de circuitos quânticos para descrever algoritmos quânticos. Em uma analogia com um circuito clássico, um circuito quântico é constituído por linhas e portas lógicas. As linhas representam qubits e as portas lógicas operações unitárias e também medidas. Em diversos livros sobre computação quântica, relacionando com as portas lógica da computação clássica, as transformações unitárias são chamadas de *portas quânticas*, conforme poderá ser verificado através do exemplo do circuito quântico na Figura 2.2. Esse circuito utiliza a porta Hadamard juntamente com uma porta controlada em sistema de dois qubits, onde o fluxo de valores inicia-se da esquerda para a direita em etapas correspondentes às portas quânticas. As portas conectadas de um ponto para outra linha representam operações controladas.

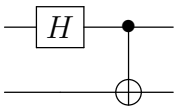


Figura 2.2 – Exemplo de um circuito quântico

Como na CQ todas as operações são reversíveis, com exceção das medidas, muitas portas lógicas clássicas não podem ser implementadas simplesmente como uma analogia de qubits pois é impossível reconstruir os valores de entrada a partir do resultado. Por exemplo, a porta clássica *NAND*, cuja tabela verdade é descrita na Tabela 2.4, não é uma operação reversível. Existe, no entanto, uma maneira de simular o efeito dessas operações com circuitos quânticos usando um qubit adicional, numa porta de três qubits chamada porta de *Toffoli*. A porta de Toffoli é uma porta *CNOT* duplamente controlada, ou seja, quando os dois qubits de controle estão, ambos, no estado $|1\rangle$ a porta de Toffoli aplica a operação de negação ao terceiro qubit. O circuito quântico que descreve esse porta, também chamado de algoritmo quântico de Toffoli, é ilustrado na Figura 2.3.

A porta de Toffoli tem um comportamento que permite simular várias operações clássicas. No caso da porta *NAND*, por exemplo, ela pode ser simulada usando dois qubits de controle q e p e um qubit alvo iniciado com valor $|1\rangle$. Considerando que saída da porta Toffoli é a des-

Entrada	Saída
00	1
01	1
10	1
11	0

Tabela 2.2 – Tabela Verdade da Porta clássica NAND

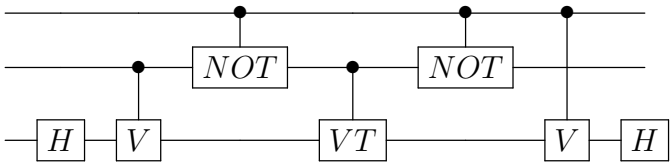


Figura 2.3 – Algoritmo Quântico de Toffoli

crita na Tabela 2.4, nesse exemplo específico tem-se que o qubit alvo só altera seu valor se os dois qubits de controle forem igual a $|1\rangle$ o que condiz com a tabela verdade da porta *NAND*.

O uso da porta quântica de Toffoli permite que computadores quânticos consigam simular qualquer circuito clássico usando um circuito reversível equivalente. Isso significa que os computadores quânticos são capazes de realizar qualquer tipo de cálculo que um computador clássico pode fazer.

Esse tipo de algoritmo é mais fácil de ser elaborado pois utiliza apenas uma sequência de porta quânticas sem precisar realizar cálculos mais complexos, envolvendo medidas por exemplo. O que não é o caso do algoritmo quântico da teleportação, que além de medidas usa outras propriedades interessantes da mecânica quântica. Esse algoritmo, é usado como uma técnica para mover estados quânticos, mesmo na ausência de um canal de comunicação quântico entre a origem e o destino. Em essência, a *teleportação quântica* permite a conexão de um estado

Entrada	Saída
000	000
001	001
010	010
011	011
100	100
101	101
110	111
111	110

Tabela 2.3 – Tabela Verdade da Porta Toffoli

quântico desconhecido através de um par EPR, compartilhado anteriormente (BENNETT et al., 1993).

2.4.1 Algoritmo Quântico da Teleportação

Para melhor esclarecer o funcionamento desse algoritmo, nomeia-se dois estados quânticos como Alice e Bob a fim de demonstrar o funcionamento por etapas do algoritmo. Esses dois estados quânticos compartilham um estado emaranhado entre si, isto é, cada um possui um dos qubit do par. Esse estado pode ser um dos possíveis valores da Tabela 2.1.

O objetivo de Alice é enviar um qubit, $|\psi\rangle$, para Bob. Porém, ela não tem o conhecimento sobre o estado desse qubit, de forma que somente poderá enviar informações clássicas para Bob. Tal situação é caracterizada por Alice não poder descobrir o conteúdo do estado $|\psi\rangle$ que ela precisa enviar para Bob pois se executar uma medida, esse estado entrará em *colapso* e perderá todas as informações originais.

A solução proposta para Alice é ter um par emaranhado compartilhado previamente com Bob. Assim, Alice pode utilizar o seu par para enviar $|\psi\rangle$ ao Bob, com um pequeno custo de comunicação clássica. Esse processo pode ser visto pela seguintes etapas: primeiramente, Alice interage seu qubit $|\psi\rangle$ (o estado a ser movido) com metade do par EPR, através da operação *CNOT*. Em seguida, ela executa uma medida nesses dois qubits e observa um dos quatro resultados clássicos, $00,\ 01,\ 10,\ 11$. Na sequência, Alice envia a informação clássica para Bob, e, dependendo da mensagem recebida, Bob executa uma de quatro possíveis operações na sua metade do par EPR. Através desse procedimento, poderá reconstituir o estado original $|\psi\rangle$.

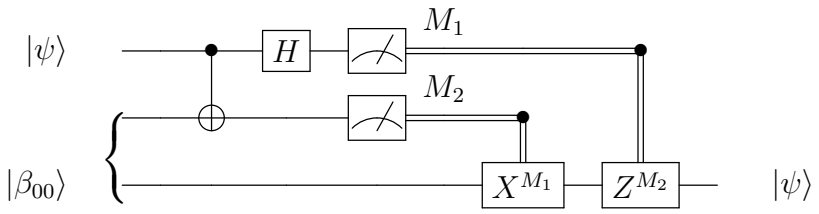


Figura 2.4 – Circuito quântico do algoritmo da teleportação (MERMIM, 2007)

O circuito quântico que representa esse algoritmo é mostrado na Figura 2.4. As primeiras duas linhas representam o sistema quântico de Alice, enquanto que a última linha o de Bob. As linhas duplas denotam a transmissão de bits clássicos e as simples representam as transmissões dos bits quânticos. Por exemplo, supondo que Alice queira enviar o estado $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ para Bob, onde α e β são amplitudes desconhecidas, a entrada do circuito é:

$$|\psi_0\rangle \quad = |\psi\rangle|\beta_{00}\rangle$$

$$= \tfrac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|00\rangle + |11\rangle)]$$

Aqui adota-se a convenção de que os dois primeiros qubits (a partir da esquerda) pertencem à Alice e o terceiro a Bob, e também que o par EPR é formado pelo segundo qubit de Alice e o de Bob seja o par β_{00} descrito na Tabela 2.1. Dessa forma, quando Alice aplica a porta *CNOT* no seu primeiro e segundo qubit obtém-se o seguinte resultado:

$$|\psi_1\rangle \quad = \tfrac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|10\rangle + |01\rangle)].$$

Consequentemente, aplicar a porta Hadamard no primeiro qubit desse novo estado, gerará o seguinte estado em superposição:

$$|\psi_2\rangle \quad = \tfrac{1}{2}[\alpha(|0\rangle + |1\rangle)(|00\rangle + |11\rangle) + \beta(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)].$$

que pode ser esclarecida na seguinte forma agrupada:

$$|\psi_2\rangle \quad = \tfrac{1}{2}[(\alpha|0\rangle + \alpha|1\rangle)(|00\rangle + |11\rangle) + (\beta|0\rangle - \beta|1\rangle)(|10\rangle + |01\rangle)]$$

$$= \tfrac{1}{2}[(\alpha|000\rangle + \alpha|011\rangle) + (\alpha|100\rangle + \alpha|111\rangle)$$
$$+(\beta|010\rangle + \beta|001\rangle) - (\beta|110\rangle + \beta|101\rangle)]$$

$$= \tfrac{1}{2}[|00\rangle(\alpha|0\rangle + \beta|1\rangle) + |01\rangle(\alpha|1\rangle + \beta|0\rangle)$$
$$+|10\rangle(\alpha|0\rangle - \beta|1\rangle) + |11\rangle(\alpha|1\rangle - \beta|0\rangle)]$$

A última etapa do algoritmo é executar uma medida em Alice e enviar a informação clássica observada para Bob [3]. A partir do valor observado por ele, poderá ser obtido o estado original $|\psi\rangle$ aplicando uma porta lógica apropriada no seu qubit, pois uma medida no último estado de Alice pode ter apenas esses quatro resultados possíveis:

$$00 \mapsto |\psi_3(00)\rangle \equiv [\alpha|0\rangle + \beta|1\rangle]$$
$$01 \mapsto |\psi_3(01)\rangle \equiv [\alpha|1\rangle + \beta|0\rangle]$$
$$10 \mapsto |\psi_3(10)\rangle \equiv [\alpha|0\rangle - \beta|1\rangle]$$
$$11 \mapsto |\psi_3(11)\rangle \equiv [\alpha|1\rangle - \beta|0\rangle]$$

sendo os dois primeiros bits o resultado de uma medida no sistema quântico de Alice e o estado $|\psi_3\rangle$ o respectivo qubit obtido para Bob. Dessa forma, se for observado o valor $|00\rangle$, o sistema de Bob estará no estado original $|\psi\rangle$ e não será preciso modificá-lo. Caso fosse observado o valor $|01\rangle$, Bob estaria no estado $\alpha|1\rangle + \beta|0\rangle$, necessitando executar uma operação *NOT* para recuperar o estado original $|\psi\rangle$.

$$\begin{pmatrix} \beta \\ \alpha \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

[3] Esse é o único fato que impede que a teleportação seja utilizada para transmitir informação em uma velocidade mais rápida que a luz pois exige um canal clássico

Se Alice observasse o valor $|10\rangle$, Bob estaria no estado $\alpha|0\rangle - \beta|1\rangle$, precisando executar uma operação *Z* para recuperar o estado original $|\psi\rangle$.

$$\begin{pmatrix} \alpha \\ -\beta \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Por fim, quanto ao valor $|11\rangle$, Bob estaria no estado $\alpha|1\rangle - \beta|0\rangle$, e seria necessário executar uma operação *NOT* e *Z* para recuperar o estado original $|\psi\rangle$.

$$\begin{pmatrix} -\beta \\ \alpha \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} \alpha \\ -\beta \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

A computação quântica busca implementar os seus algoritmos através das *transformações unitárias* nos estados quânticos (PALAO; KOSLOFF, 2002). Porém, esse exemplo de circuito quântico do algoritmo da teleportação serve para demostrar que, além utilizar as portas lógicas da computação quântica, é fundamental explorar os recursos de paralelismo quântico, medidas e estados emaranhados, a fim de elaborar algoritmos quânticos eficientes. Outros exemplos de algoritmos famosos são: o de Deutsch (1985) que usa o paralelismo quântico para comprovar que os algoritmos quânticos são mais rápidos que os clássicos; o de Shor (1994), para encontrar os fatores primos de números inteiros grandes em tempo polinomial (onde o melhor tempo de um algoritmo clássico é exponencial); e o algoritmo de busca de Grover (1996) com eficiência na ordem de O($\sqrt{n}$).

3 CÁLCULO-λ, MÔNADAS, ARROWS

O Cálculo *lambda*, ou Cálculo-λ, é o núcleo das linguagens de programação funcionais. Sua versão original envolve apenas definições de funções e suas aplicações, não oferecendo recursos de alto nível. Em linguagens funcionais de propósito geral, onde existem alguns tipos de *efeitos computacionais*, além de funções, são utilizadas versões estendidas do cálculo-λ tradicional. Pode-se citar alguns desses efeitos como entrada/saída, não-determinismo, exceções ou atribuições (estado global).

Na comunidade de linguagens funcionais, existe um longo debate sobre como estruturar a semântica para esses efeitos computacionais. A maneira como os efeitos são estruturados dividem-as em duas vertentes: linguagens funcionais *puras* e linguagens funcionais *impuras* (SABRY, 1998; WADLER, 1995). Nesse sentido, Moggi (1989) apresentou uma abordagem monádica para modelar efeitos computacionais no contexto de linguagens de programação funcional *puras* (WADLER, 1992), desencadeando uma série de estudos sobre mônadas e sua aplicabilidade (MOGGI, 1991; LIANG; HUDAK; JONES, 1995; BENTON; HUGHES; MOGGI, 2000).

Mônadas são uma maneira de abstrair computações e *mônadas* estão presentes na modelagem de efeitos computacionais para linguagem funcionais como o *Haskell* (HUDAK; PEYTON JONES; WADLER, 1992). Devido a sua flexibilidade e modularidade, elas também podem ser utilizadas como solução para problemas em diferentes áreas da computação. Por exemplo, no trabalho (HARRIS et al., 2005) se usa mônadas para modelar memórias transacionais para programação concorrente, no trabalho (PETERSON; HAGER, 1999) se usa mônadas para uma linguagem de domínio específico para a construção de controladores robóticos.

Entretanto, existem casos nos quais as *mônadas* não são suficientemente gerais para expressar certas computações. Em (SWIERSTRA; DUPONCHEEL, 1996), por exemplo, é apresentada uma biblioteca para analisar gramáticas LL-1, cujo *parser* utiliza um componente estático (independente da entrada) com a análise do programa até o momento. Tal definição não pode ser abstraída como uma mônada, pois não era possível combinar computações já que a resposta dependia tanto do componente estático quanto da função a ser combinada. Assim, Hughes (2000) demostrou que mônadas podem ser generalizadas para setas (*arrows*) relacionando entradas e saídas. Dessa forma, o uso de setas permite expressar, além de abstrações monádicas, abstrações mais gerais, como por exemplo, uma abstração para processar múltiplas

entradas ou apenas parte dela (componente estático).

No restante deste capítulo será abordada os conceitos sobre o cálculo-λ e a importância de mônadas para a programação funcional, bem como uma descrição de o que são mônadas e setas.

3.1 Cálculo-λ Lambda

O Cálculo-λ criado por Alonzo Church na década de 40, como parte de uma investigação dos fundamentos matemáticos, mas só foi incorporado a Ciência da Computação na década de 60 quando Landim conseguiu provar que uma linguagem de programação complexa poderia ser formulada reduzindo todas a computações como operações básicas de definição de funções e aplicações (CHURCH, 1940) (LANDIM, 1964).

Além de ser utilizado como o núcleo das linguagens de programação funcionais, atualmente, o Cálculo-λ é muito utilizado para especificar características de linguagens de programação e no estudo de sistemas de tipos (PIERCE, 2002). A sintaxe completa do Cálculo-λ é composta por apenas três termos (categoria sintática T), conforme visualizado na Figura 3.1.

Sintaxe Cálculo-λ

$t ::=$	Termos
x	variável
$\lambda\, x.\, t$	abstração
$t\, t$	aplicação

Figura 3.1 – Sintaxe do Cálculo Lambda (PIERCE, 2002)

Nessa notação, usamos o símbolo λ, lambda, para representar uma função. Por exemplo, uma função $f : \ x \mapsto x + y$, que recebe um argumento x e deve retornar a soma de x com um valor pré-definido y, deverá ser escrita na seguinte forma: $\lambda\, x.\, x + y$.

Quando se trata do escopo de variáveis, uma variável pode estar *livre* ou *ligada* no escopo da abstração. Uma ocorrência de um variável x é dita *ligada* quando ela aparece dentro do corpo T de uma abstração $\lambda\ x.\ T$. Uma ocorrência de uma variável x é dita *livre* quando ela aparece em uma posição em que não está ligada por uma abstração (PIERCE, 2002). Por exemplo, na função anterior $\lambda\, x.\, x{+}y$, y é uma variável livre, enquanto x é ligada pela abstração.

Note ainda que, nessa mesma função, x é utilizada dentro do corpo de T. Uma *aplicação*

dessa função, por exemplo $f(2)$ por ser escrita como:

$$\overbrace{\underbrace{(\lambda x.x + y)}_{T}\underbrace{(2)}_{T}}^{\text{aplicação}} \tag{3.1}$$

Para explicar como se chega ao resultado da aplicação das abstrações primeiro é preciso entender como funciona a semântica do cálculo-λ.

3.1.1 Semântica Operacional

Uma semântica operacional define a forma como são computadas as expressões descritas através da sintaxe de uma linguagem de programação. Essa semântica é representada pelo comportamento de uma *máquina abstrata* que processa um estado atual para um estado final. Esses estados são os termos da sintaxe de uma linguagem de programação, e podem, de acordo com o comportamento da semântica, sofrer uma mudança para outro estado, um termo mais simplificado, ou interromper a máquina. Em suma, a semântica operacional define a maneira como se avaliar as expressões de uma linguagem.

Na forma mais simples do Cálculo-λ, como se tem apenas a definição de funções e suas aplicações, somente é preciso computar a aplicação dessas funções em argumentos, esse passo é chamado de *substituição*. Dessa forma, se aplica o componente a direita como o argumento dessa abstração, substituindo a variável ligada no corpo da abstração pelo componente a direita, como descrito na seguinte equação:

$$(\lambda x.\ t_1)\ t_2 \rightarrow [x \mapsto t_2]\ t_1 \tag{3.2}$$

Onde $[x \mapsto t_2]\ t_1$ significa o termo obtido pela substituição de todas ocorrências de x em t_1 por t_2. Por exemplo, o termo $(\lambda\ x.\ x)\ y$ avalia para y pois $[x \mapsto y]\ x$ é igual a y. Outro exemplo é o termo $(\lambda\ x.\ x(\lambda\ x.\ x))\ (u\ r)$ que avalia para $u\ r\ (\lambda\ x.\ x)$.

Church chamou de *redex* ou expressão redutível um termo na forma $(\lambda x.\ t_1)\ t_2$.A operação de reescrever um redex de acordo com a regra acima é chamada beta redução (*beta-reduction*). O processo de passo a passo da redução da Equação 3.1 pode vista como:

$$(\lambda x.\ x + y)\ (2) \tag{3.3a}$$

$$([x \mapsto 2]\ (\lambda x.x + y)) \tag{3.3b}$$

$$2 + y \tag{3.3c}$$

Na Equação 3.3a encontra-se a expressão que será reduzida. Em 3.3b é demostrado como será a beta-redução, que nesse caso específico é substituir o valor de x por 2. E por fim, em 3.3c está o resultado.

Na sua forma original, o *cálculo-*λ possui apenas definição de funções e aplicações. Não existem recursos de alto nível como *arrays*, operações condicionais ou *laços*. Para simular essas operações mais complexas, é preciso descrevê-las como funções lambda e aplicações, mantendo a mesma semântica operacional descrita acima. Por exemplo, a operação condicional if é descrita como uma função que recebe um argumento e verifica se ele é verdadeiro ou falso, mapeando o resultado para a respectiva saída.

Esse argumento deve ser uma valor booleano, mas o cálculo-λ puro não possui um sistema de tipos para representar esse valor, o que acontece é que esses valores são descritos como funções usando a sintaxe do cálculo-λ. Os valores $True$ e $False$, também chamados de *Booleanos de Church*, são descritos, respectivamente, como as seguintes funções:

$$True = \lambda\, t.\lambda\, f.\, t \tag{3.4a}$$

$$False = \lambda\, t.\lambda\, f.\, f \tag{3.4b}$$

A função $True$ retorna o primeiro argumento, t, e a função $False$ o segundo argumento, f. Assim, a operação if pode ser descrita como a seguir:

$$if = \lambda\, bool.\, \lambda\, a.\, \lambda\, b.\, bool\ a\ b \tag{3.5}$$

O primeiro argumento, $bool$, deve ser uma das funções booleanas, o segundo argumento, a, é o que acontece se a condição for verdadeira e o terceiro argumento, b, o que acontece se a condição for falsa. A operação if recebe três argumento e simplesmente chama a função $bool$ passando os argumento a e b, logo, o resultado dependerá do valor booleano. Por exemplo, $if\ True\ v\ w$ resultaria em v, $if\ False\ v\ w$ resultaria em w. A beta redução, passo a passo, deste tipo de expressão pode ser vista como:

$$\begin{aligned}
&if\ True\ v\ w \\
&((\lambda\, a.\, \lambda\, b.\, True\ a\ b)\ v\ w) \\
&([a \mapsto v]\ (\lambda\, b.\, True\ a\ b)) \\
&((\lambda\, n.\, True\ v\ b)\ w) \\
&([b \mapsto w]\ (True\ v\ b))
\end{aligned}$$

$$\begin{aligned}
&((True\ v\ w)) \\
&((\lambda\, t.\, \lambda\, f.\, t)\ v\ w)) \\
&([t \mapsto v]\ (\lambda\, f.\, t))) \\
&((\lambda\, f.\, v)\ w) \\
&([f \mapsto w]\ v) \\
&v
\end{aligned}$$

No primeiro passo se traduz a expressão segunda a definição da função if. Em seguida, se substitui através da beta redução as variáveis a e b pelos respectivos argumentos v e w. No próximo passo se traduz a expressão $True$ para a sua forma de função. Por fim, t recebe o argumento v e o coloca no corpo da função, mas f recebe o argumento w e não faz nada. Dessa forma, o resultado da avaliação é apenas v.

3.1.2 Cálculo Lambda Estendido e Transparência Referencial

Como tipos são muito relevantes para a discussão sobre adicionar efeitos monádicos a linguagem, será apresentado o cálculo-λ puro estendido com tipos simples e com regras para números e adição. A seguinte gramática gera um conjunto de tipos simples sobre o tipo de números naturais:

$$A \quad ::= \quad \mathbb{N} \;||\; A \to A$$

Isto é, um tipo para números naturais e um tipo para funções. Considera-se agora a sintaxe para expressões descrita na Figura 3.2.

Sintaxe Cálculo-λ Estendido

$t ::=$	Termos
x	variável
$\lambda\, x : A.\, t$	abstração
$t_1\; t_2$	aplicação
n	números naturais
$t_1 + t_2$	soma

Figura 3.2 – Sintaxe do Cálculo Lambda Estendido (PIERCE, 2002)

A regra de derivação diz que as expressões da linguagem são constituídas de variáveis simples. Dessa forma, na abstração a variável é ligada a um tipo explícito o que significa que o argumento será explicitamente do tipo A.

A relação de tipos do cálculo-λ puro é definida por um conjunto de regras de inferência atribuindo tipos a expressões, resumida na Figura 3.3. A relação de tipos é escrita como $\Gamma \vdash e : A$ e lida no ambiente Γ como a expressão t é do tipo A, onde Γ é o conjunto de suposições da variáveis livres em t.

Com a adição de novos termos ao cálculo-λ, a semântica operacional descrita na seção anterior, não é o suficiente para descrever o modo como serão avaliadas as expressões. Aqui, deve ser usada uma semântica equacional, que em contraste a semântica operacional, descreve

o significado dos termos matematicamente como valores ou funções. Os seguintes axiomas dão a semântica equacional para a linguagem:

$$\begin{array}{rcll} (\lambda x : A.t)\ t' & = & t[t'/x] & (\beta) \\ \lambda x : A.t\ x & = & t & (\eta) \\ (t_1 + t_2) + t_3 & = & t_1 + (t_2 + t_3) & (S) \\ t_1 + t_2 & = & t_2 + t_1 & (C) \\ n_1 + n_2 & = & n \qquad \text{where } n = n_1 + n_2 & (A) \end{array}$$

As duas primeiras equações são usualmente chamadas de β e η redução. A notação $t[t'/x]$ representa a substituição, dizendo que o final da expressão será t com as ocorrências livres de x substituídas por t'. As três restantes equações representam a associatividade, comutatividade e avaliação da adição, respectivamente. Com a seguinte regra de inferência;

$$\frac{t_1 = t_2}{C[t_1] = C[t_2]}$$

para qualquer contexto de avaliação C. O contexto de avaliação, ou simplesmente contexto, foi introduzido para a semântica de redução de linguagens de programação por Felleisen (92) e controla onde a avaliação ocorre. Um contexto é uma expressão ou sub-expressão com um espaço reservado (representado por um $\Box$) para onde a próxima etapa da avaliação pode ter lugar. Por qualquer contexto quer dizer que a ordem de avaliação de uma (sub)expressão é irrelevante.

Por exemplo, considere a avaliação da expressão $(\lambda x.x + x)\ (4 + 2)$. Dependendo do contexto da avaliação se considera que pode ter diferentes ordem de avaliação. Considerando $C = \Box$, o contexto vazio, e $t_1 = (\lambda x.x + x)\ (4 + 2)$ se pode ter a ordem de avaliação descrita abaixo, e considerando $C = (\lambda x.x + x)\ \Box$ e $t_1 = (4 + 2)$ se pode ter a outra ordem de avaliação descrita abaixo.

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ \text{Tipo Cons} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{Tipo Var} \qquad \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.e : A_1 \to A_2}\ \text{Tipo Abs}$$

$$\frac{\Gamma \vdash t_1 : A_{11} \to A_{12} \qquad \Gamma \vdash e_2 : A_{11}}{\Gamma \vdash t_1\ t_2 : A_{12}}\ \text{Tipo App}$$

$$\frac{\Gamma \vdash t1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash t_1 + t_2 : \mathbb{N}}\ \text{Tipo Sum}$$

Figura 3.3 – Cálculo-λ Puro Simplesmente Tipado

$$
\begin{array}{lll}
 & (\lambda x.x + x)\ (4 + 2) & \\
=^{(by\ \beta)} & (4 + 2) + (4 + 2) & \\
= & 6 + (4 + 2) & \\
= & 6 + 6 & \\
= & 12 &
\end{array}
\qquad
\begin{array}{lll}
 & (\lambda x.x + x)\ (4 + 2) & \\
=^{(by\ C)} & (\lambda x.x + x)\ (2 + 4) & \\
= & (\lambda x.x + x)\ 6 & \\
= & 6 + 6 & \\
= & 12 &
\end{array}
$$

Esse exemplo também serve para ilustrar a propriedade da *transparência referencial*, isto é, as expressões carregam o mesmo valor a cada vez que são chamadas independente da ordem de avaliação.

Agora, considere a adição na sintaxe da linguagem de uma localização global implicitamente inicializado como 0, e as expressões get e inc:

Sintaxe Cálculo-λ Estendido com estado global

$t ::=$	Termos
x	variável
$\lambda x.t$	abstração
$t\ t$	aplicação
n	números naturais
$t + t$	soma
get	
inc	

Figura 3.4 – Sintaxe do Cálculo Lambda Estendido com estado global

onde get retorna o conteúdo da localização global e inc retorna o conteúdo atual da localização global e incremente ele como um efeito colateral.

Nesse sentido, a avaliação da expressão $(\lambda x.x + x)\ inc$ resultaria em:

$$
\begin{array}{lll}
 & (\lambda x.x + x)\ inc & \\
=^{(by\ \beta)} & inc + inc & \\
= & 0 + inc & \\
= & 0 + 1 & \\
= & 1 &
\end{array}
\qquad
\begin{array}{lll}
 & (\lambda x.x + x)\ inc & \\
=^{(by\ inc)} & (\lambda x.x + x)\ 0 & \\
= & 0 + 0 & \\
= & 0 &
\end{array}
$$

Novamente aqui, podem ser consideradas as mesmas duas ordens de avaliação citadas acima. Contudo, nesse exemplo, com o acesso a uma *variável global*, a *ordem de avaliação é muito relevante*. Dependendo da ordem, o resultado será diferente. Isso acontece porque foi adicionada alguma noção de *efeito computacional* a linguagem. Na verdade, inc não é simplesmente uma função que retorna um valor, a chamada de inc pode retornar em resultados diferentes a cada vez dependendo do estado global.

Uma solução elegante para o problema de modelar *efeitos computacionais* em linguagens funcionais é usar *mônadas*. Com esse tipo de abstração é possível manter as propriedades funcionais da linguagem intactas, incluindo a *transparência referencial*.

3.2 Mônadas

Mônadas foram aplicadas pela primeira vez na Computação em (MOGGI, 1989a), onde o autor apresenta uma *semântica categórica para noções de computações*. Como um exemplo dessa abordagem semântica para computações, Moggi mostrou um modo geral de estruturar várias *noções de computações* no *cálculo-λ computacional*. O cálculo-λ computacional é uma modificação do cálculo-λ tipado, que é correto para realizar provas de equivalência de programas independentes de qualquer tipo específico de *noção de computação*.

Por *noção de computação* (ou também chamado *efeito computacional*) ele quer dizer uma descrição quantitativa de denotações de *programas*. Exemplos de noções de computações são: computações com efeitos colaterais, onde um programa denota um mapeamento de um registro para um par, valor ou registro modificado; computações com exceções, onde um programa denota tanto um valor quando uma exceção; computações parciais, onde um programa denota tanto um valor ou uma divergência; computação com não-determinismo, onde um programa denota um conjunto de valores possíveis.

3.2.1 Mônadas na Teoria das Categorias

Ao considerar a semântica categórica para computações pode se considerar diferentes níveis de abstração. Ao lidar com mônadas, se considera a *visão denotacional*, onde programas são vistos como um morfismo de uma categoria, cujos objetos são tipos.

Uma simplificação da *visão denotacional* mostra que os programas podem ser identificados como funções de valores para valores. Contudo, essa é uma simplificação muito simples como dito em (MOGGI, 1989b). De fato, um programa pode incluir noções de efeitos computacionais, como entrada/saída, exceções, efeitos colaterais, etc. Nesse sentindo, um programa pode ser entendido como uma *função de valores para computações*. Por exemplo, um programa pode receber um *valor* e *imprimir* numa saída padrão.

Dessa forma, toma-se uma categoria C como um modelo para funções e desenvolve-se em cima dela uma compreensão geral do *tipo* referente a valores para computações. Como

explicado por Moggi (1989b), usa-se uma operação unária T nos objetos de C, que mapeia um objeto A, visto como um conjunto de valores do tipo τ, para um objeto TA correspondendo ao conjunto de computações do tipo τ. Assim, um programa de A para B, isto é, que recebe um valor de entrada do tipo A e após realizar alguma computação retorna um valor do tipo B, pode ser identificado como um morfismo de A para TB em C. Finalmente, tem um conjunto mínimo de requisitos sobre os valores e computação para que os programas sejam um morfismo de uma categoria adequada.

Essa discussão informal sobre noções de computações e teoria das categorias leva ao uso das triplas de Kleisli para modelar noções de computações e a categorias de Kleisli para modelar categoria de programas.

Definition 3.2.1 *Uma **tripla de Kleisli** sobre uma categoria C é uma tripla $(T, \eta, \_^*)$, onde $T : Obj(\mathsf{C}) \to Obj(\mathsf{C})$ (tal que, $Obj(\mathsf{C})$ corresponde a classe de objetos da categoria C), $\eta_A : A \to TA$, $f^* : TA \to TB$ para $f : A \to TB$, sustentada pelas seguintes equações:*

- $\eta_A^* = id_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Intuitivamente η_A é a *inclusão* de valores para computações e f^* é a *extensão* de uma função f de valores para computações para um função de computações para computações. Os axiomas equivalem a dizer exatamente que os programas formam um categoria, a categoria de Kleisli C_T, onde o conjunto $\mathsf{C}_T(A, B)$ de morfismos de A para B é $\mathsf{C}(A, TB)$, a identidade sobre A é η_A e a composição de $f : A \to TB$ seguida de $g : B \to TC$ é $f; g^*$.

Mônadas são equivalentes as triplas de Kleisli, onde, como visto acima, são facilmente justificadas computacionalmente. Há uma correspondência de um-para-um entre as triplas de Kleisli e mônadas. Contudo, a definição formal de mônadas é dada em termos de *functors* e transformações naturais. Não será mostrado a definição formal de mônadas, pois a definição alternativa de Kleisli é o suficiente no contexto da semântica denotacional de efeitos computacionais em linguagens de programação. Informações adicionais a respeito de mônadas podem ser encontradas no trabalho de Barr e Wells (1995). .

3.2.2 Modelando efeitos com mônadas

O estado da arte em linguagens funcionais puras usa o conceito de mônadas para modelar efeitos computacionais. Com mônadas é possível separar os efeitos da linguagem pura através do sistema de tipos. Além disso, elas impõem modificações mínimas na linguagem para sequenciar esses efeitos: é preciso ter um operador de composição, que é associativa, e uma unidade de composição. De fato, mônadas provêm um tratamento genérico para uma grande classe de efeitos computacionais.

Dessa forma, considerando o problema do estado global mostrada na Seção 3.1.2, manipulado com os termos get e inc, a seguinte solução com mônadas é proposta. De acordo com a definição de uma mônada segundo uma tripla de Kleisli, é preciso adicionar a tripla $(T, \eta, \_^*)$ para uma localização global da linguagem. Neste caso, um functor $T\ A = (A \times S)^S$, onde S é um conjunto de estados. Isto é, o functor mapeia um tipo A de valores para computações que são funções de estados/registros para um par de um valor junto com um estado/registro modificado: $S \to (A \times S)$. A transformação natural η é a de mapear $a \mapsto (\lambda s : S.\langle a, s\rangle)$ e se $f : A \to TB$ e $c \in TA$, então $f^*(c) = \lambda s : S.(let\langle a, s'\rangle = c(s)\ in\ f(a)(s'))$. Note que f^* é a função de passagem de estado.

Na comunidade de linguagem de programação, o η é usualmente chamado de uma função $return :: A \to TA$ e a composição, _*, é usualmente pronunciada "bind", $\gg\!=:: TA \to (A \to TB) \to TB$, uma função que recebe duas entradas, um efeito computacional, TA, e uma função de $A \to TB$, e retorna um efeito computacional, TB [4]. Dessa forma, a adição dessas duas construções a linguagem pode ser vista como:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x : A.e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ & & return\ e \mid e_1 \gg\!= e_2 \mid inc \mid get \end{array}$$

Com as seguintes regras de tipos:

$$\frac{}{\Gamma \vdash inc : TA}\ \text{Tipo Inc} \qquad \frac{}{\Gamma \vdash get : TA}\ \text{Tipo Get}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash return\ e : TA}\ \text{Tipo Return} \qquad \frac{\Gamma \vdash e_1 : TA \quad \Gamma \vdash e_2 : a \to TB}{\Gamma \vdash e_1 \gg\!= e_2 : TB}\ \text{Tipo Bind}$$

Figura 3.5 – Tipos para as Funções Monádicas

Agora será ilegal escrever o programa como $(\lambda x.x + x)\ inc$. O valor da localização

[4] Note que o conjunto de argumentos do $\gg\!=$ e _* são os mesmos só com uma pequena mudança de ordem.

global deverá ser propagado e mantido pela mônada. A forma correta dessa expressão é utilizar as funções monádicas para poder sequenciar o efeito. O que pode ser de duas formas:

$$
\begin{array}{l}
inc \ >>= \lambda \, x_1 \\
inc \ >>= \lambda \, x_2 \\
return \ (x_1 + x_2)
\end{array}
\qquad\qquad
\begin{array}{l}
inc \ >>= \lambda \, x_1 \\
return \ (x_1 + x_1)
\end{array}
$$

A expressão à esquerda, $x_1 + x_2$, se reduz a 1, e a expressão à direita, $x_1 + x_1$, se reduz a 0. As duas expressões são avaliadas seguindo a mesma semântica e preservam a propriedade de transparência referencial. O resultado final depende apenas da intenção do programador em escolher qual alternativa utilizar.

Sobre os axiomas e regras de inferência, a abordagem monádica alega que deverá ser mantido todos os axiomas da linguagem pura, ser adicionado os axiomas genéricos das mônadas (as leis monádicas que refletem as equações da Definição 3.2.1), e também os axiomas referentes ao efeito em questão, nesse caso get e inc.

Os axiomas genéricos na definição da tripla de Kleisli podem ser reescritos nos termos das funções monádicas, $return$ e $\gg\!=$, como:

1. Identidade a Esquerda:

   $return \ a \gg\!= f \quad \equiv \quad f \ a$

2. Identidade a Direita:

   $m \gg\!= return \quad \equiv \quad m$

3. Associatividade

   $(m \gg\!= f) \gg\!= g \quad \equiv \quad m \gg\!= (\ x \rightarrow f \ x \gg\!= g)$

Os axiomas para este efeito em particular, get e inc, são definidos como:

$$
\begin{array}{lcll}
\langle inc \gg\!= \lambda \, x.t, n\rangle & = & \langle t[n/x], n+1\rangle & (inc) \\
\langle get \gg\!= \lambda \, x.t, n\rangle & = & \langle t[n/x], n\rangle & (get)
\end{array}
$$

Note que está sendo explicitamente mostrado o que acontece com a localização global (gl) por meio da notação $\langle e, gl\rangle$. Para ilustrar a redução em etapas da avaliação das expressões com uma localização global explicitamente inicializado como 0, observe os passos a seguir:

$$
\begin{array}{ll}
 & \langle inc \ >>= \lambda \, x_1.\, return \ (x_1 + x_1), 0\rangle \\
=^{(by \ inc)} & \langle return(0+0), 1\rangle \\
=^{(by \ A)} & \langle return(0), 1\rangle
\end{array}
$$

3.2.3 Mônadas em Haskell

Haskell é uma linguagem de programação funcional pura. Independentemente de ser pura [5], Haskell tem cálculo-λ puro como seu núcleo de linguagem, o que oferece mecanismo de raciocínio funcional simples e transparência referencial. Como uma linguagem de propósito geral, Haskell oferece uma variedade de efeitos computacionais modelados com mônadas, como entrada/saída (a mônada *IO*), listas (a mônada *List*), e falha (a mônada *Maybe*). A abordagem usada no design da linguagem é a mesma explicado nas seções anteriores.

Além disso, no Haskell, o usuário pode também declarar um novo tipo como uma mônada. Isto é, qualquer efeito computacional de interesse pode ser adicionado a linguagem e tratado como uma mônada. Para fazer isso, o usuário precisa definir:

1. Um construtor de tipo M

   ```
   data M
   ```

2. Uma definição de um função return:

   ```
   return :: a -> M a
   ```

3. Uma definição de um função bind:

   ```
   >>= :: M a -> ( a -> M b) -> M b
   ```

Mais precisamente, um construtor de tipo pode ser considerado como um modelo para funções com efeitos. O mapeamento trivial de valores para computações é dado na definição da função `return`. A função $\gg=$ deve refletir como sequenciar/transformar efeitos. Note que a transformação de uma computação $M\ a$ em uma computação $M\ b$ deve ser definida nos termos de uma função do tipo $a \rightarrow M\ b$.

Além disso, para construir corretamente uma *mônada*, as funções `return` e $\gg=$ devem trabalhar juntas de acordo com os três *axiomas monádicos* apresentados na seção anterior.

3.3 Setas

Setas (*Arrows*) são uma generalização de mônadas e por isso possuem uma aplicabilidade mais ampla. Introduzidas por Hughes (HUGHES, 2000), as *setas* são capazes de estruturar

[5] Não será discutido sobre linguagens funcionais puras e impuras nesse trabalho, para uma boa leitura veja (SABRY, 1998).

uma classe maior de efeitos computacionais pois permitem expressar, além de abstrações monádicas, abstrações mais gerais. Como por exemplo, uma abstração para processar mais de uma entrada ou apenas parte dela (uma parte estática).

As *setas* foram primeiramente definidas como uma solução para o problema da biblioteca de *parsers* de Swierstra e Duponcheel (1996) pois ela não podia ser modelada com mônadas. Basicamente, a biblioteca proposta pelos autores incluía um componente estático para o *parser* com algumas informações sobre os *tokens* que poderiam ser aceitos primeiro pela entrada. Swierstra e Duponcheel não conseguiram implementar $(\gg\!=) :: Parser\ s\ a \rightarrow (a \rightarrow Parser\ s\ b) \rightarrow Parser\ s\ b$ pois o componente estático, s de $Parser\ s\ b$ dependente de *ambos* o primeiro e segundo argumentos. Veja que pela definição de $\gg\!=$, enquanto se tem acesso ao componente estático do primeiro argumento, o do segundo argumento será inacessível sem aplicar algum valor do tipo a. A solução de Hughes para esse problema era mudar a representação da função do tipo $a \rightarrow Parser\ s\ b$ para um tipo abstrato, que ele chamou de uma *seta* de a para b. Essa representação com setas permitiu que o acesso as propriedades estáticas do *parser* pudesse ser feito imediatamente.

Na realidade, não é preciso lidar com dois tipos abstratos, um tipo monádico e um outro tipo seta. Em vez disso, Hughes trabalhou somente com setas (arrows) e generalizou um tipo *arrows* para um tipo parametrizado com dois parâmetros em uma analogia com o $return$ e $\gg\!=$ das mônadas. Logo, assim como vemos uma mônada $m\ a$ representando uma computação que resulta em um tipo a, nós temos uma tipo seta (arrow) $a\ b\ c$ representando uma computação com entrada do tipo b resultando em um tipo c.

3.3.1 Setas em Haskell

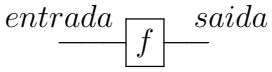
Assim como em Haskell uma mônada pode ser definida como uma classe de tipos *Monad*, uma arrow em Haskell também pode ser definida uma classe de tipos *Arrow* com os seus operadores. Diferente das mônadas, as setas explicam uma dependência na entrada. Sendo assim, enquanto o operador `return`, do tipo $a \rightarrow m\ a$, convertia valores para computações nas mônadas, nas setas considera-se um operador `arr`, do tipo $(b \rightarrow c) \rightarrow a\ b\ c$, que converte *uma função de entrada para saída* em uma computação. Assim como nas mônadas, as setas apresentam um operador de composição de computações e mais um operador `first`.

A implementação de setas em Haskell é definida como:

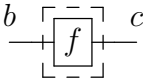
```
class Arrow a where
        arr      :: (b -> c) -> a b c
```

```
(>>>) :: a b c -> a c d -> a b d
first    :: a b c -> a (b,d) (c,d)
```

Para uma melhor explicação do funcionamento das setas, elas podem ser ilustradas graficamente como:

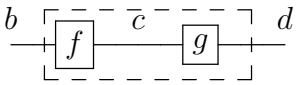


onde as linhas representam os valores de um respectivo tipo e os retângulos as computações (funções). Assim, o operador `arr` pode ser representado como:

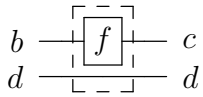


`b` é um valor de entrada que sofre alguma computação `f` que resulta em algum valor do tipo `c`. Sendo assim, operador `arr` recebe uma função de `b` para `c` e monta uma seta mapeando um valor de entrada `b` para uma saída `c`.

O operador de composição $\ggg$ recebe uma seta de `b` para `c` e uma seta de `c` para `d` retornando a combinação dessas computações. Ele retorna uma seta que mapeia uma entrada `b` para uma saída `d`, como ilustrado a seguir:



O operador `first` converte uma seta de b para c em uma seta de pares. Esse operador é usado para processar parte da entrada, pois aplica uma computação apenas no primeiro componente do par deixando o valor do segundo componente inalterado. Isso significa que o valor do segundo componente pode ser propagado através das combinações de computações sem sofrer mudanças. Sua ilustração demostrada como é processado apenas o primeiro componente de uma seta:



Essas três operações: `arr`, $\ggg$ e `first`, definem o comportamento da seta e devem satisfazer seguintes axiomas, também chamados de *arrows laws*:

```
                arr id >>> f = f
                f >>> arr id = f
            (f >>> g) >>> h = f >>> (g >>> h)
                arr (g . f) = arr f >>> arr g
              first (arr f) = arr (f x id)
            first (f >>> g) = first f >>> first g
  first f >>> arr (id x g) = arr (id x g) >>> first f
       first f >>> arr fst = arr fst >>>= f
first (first f) >>> arr assoc =  assoc >>> first f
```

a função `fst` extrai o primeiro componente de uma tupla, e as funções `assoc` e `x` são definidas como:

```
(f x g) (a,b) = (f a, g b)
assoc ((a,b),c) = (a, (b,c)
```

Essas equações acima garantem que as operações das setas sejam bem definidas mesmo com permutações arbitrárias e mudança na associatividade, preservando a combinação de composições.

4 MODELANDO EFEITOS QUÂNTICOS COM MÔNADAS E SETAS

Conforme visto no Capítulo 2, o modelo tradicional da computação quântica é baseado em espaços vetoriais formados por bases ortonormais, ou seja, *vetores normalizados*. Em tal modelo, os estados computacionais são representados por *vetores unitários* e as computações quânticas por *transformações unitárias*.

Consideram-se *estados quânticos fechados* onde a informação evolui de forma reversível através de transformações unitárias reversíveis (medidas não estão consideradas). Desse modo, o processo da computação quântica pode ser definido uma "caixa preta", onde se envia uma informação como entrada, aplicam-se operações reversíveis e se recebe uma saída no final do processo referente à informação processada.

Mu e Bird (2001) foram os primeiros autores a suspeitar que este modelo puro da computação quântica baseado em vetores e transformações unitárias podia ser modelado com *mônadas*. Baseados nessa ideia, no trabalho (VIZZOTTO; ALTENKIRCH; SABRY, 2006) os autores apresentam um modelo monádico para computações quânticas puras. Para trabalhar com um modelo geral da computação quântica, os autores também apresentaram um modelo para medidas quânticas e estados mistos, usando o conceito de setas (arrows). Ambos os modelos foram implementados na linguagem funcional Haskell. O presente capítulo resume esse trabalho.

4.1 Mônada Quântica para Estados Puros

A abordagem monádica para o modelo tradicional de computação quântica, instância um tipo `Vec` para vetores de estado quântico puro como uma mônada, de tal maneira que associa cada elemento da base a uma amplitude de probabilidade complexa.

4.1.1 Vetores

Em Haskell, um conjunto finito `a` pode ser representado por uma instância da classe `Basis`, cujo construtor cria uma lista dos possíveis elementos da base. Tais elementos devem ser distinguidos um do outro, por isso é utilizada a constante `Eq a`. A definição de um vetor e suas bases é descrita como:

```
class Eq a => Basis a where Basis :: [a]
type K = Complex Double
```

```
type Vec a = a -> K
```

O tipo `K` é utilizado para representar as amplitudes de probabilidade. O tipo `Vec` representa um função que recebe um valor do tipo `a` e retorna uma amplitude de probabilidade, $a \mapsto \mathcal{C}$.

Como os vetores só podem ser construídos sobre algum *conjunto*, isto é, uma instância da classe `Basis`, as computações acabam tendo uma restrição adicional pois elas são indexadas por `Basis`. Dessa forma, o construtor de tipo `Vec` corresponde a uma mônada indexada cuja representação genérica em Haskell é dada como:

```
classe IMonad m where
        return :: (F a) => a -> m a
        (>>=) :: (F a, F b) => m a -> (a -> m b) -> m b
```

Sendo assim, uma instância da classe `IMonad` para representar vetores é definida como:

```
instance IMonad Vec where
        return :: (Basis a) => a -> Vec a
        return a b = if a = b then 1.0 else 0
        (>>=) :: (Basis a, Basis b) => Vec a -> (a -> Vec b) -> Vec b
        va >>= f = \ b -> sum [(va a) * (f a b) | a <- basis]
```

onde `return` mapeia valores para vetores, e o `bind`, dado um operador linear representado por uma função `a -> Vec b`, e dado um vetor `Vec a`, retorna um vetor `Vec b` que corresponde a aplicação da transformação no vetor de entrada. Essa mônada indexada com `Vec` satisfaz os axiomas monádicos da Seção 3.2.

As mônadas indexadas também podem ter certas propriedades abstraídas pela versão indexada da `MonadPlus`:

```
classe IMonad m => IMonadPlus m where
        mzero :: (F a) => a -> m a
        mplus :: (F a) => m a -> m a -> m a
```

O uso de uma instância dessa classe permite a adição de dois métodos: `mzero`, que prove uma computação `zero`, e `mplus` que soma duas computações. Do ponto de visto de vetores, essas funções podem ser úteis para representar vetores nulos ou somas de vetores. Inclusive, um método `mminus` pode ser implementado como uma analogia a soma, a subtração.

```
instance IMonadPlus Vec where
        mzero  :: (Basis a) => a -> Vec a
        mzero = const 0.0
        mplus  :: (Basis a) => Vec a -> Vec a -> Vec a
        mplus v_1 v_2 a = v_1 a + v_2 a
        mminus :: (Basis a) => Vec a -> Vec a -> Vec a
        mminus v_1 v_2 a = v_1 a + v_2 a
```

Como está sendo modelado espaços vetoriais em Haskell, será útil definir certas operações sobre vetores, tais como: produto *escalar*, produto *tensorial* e produto *interno*.

```
($*) :: K -> Vec a -> Vec a
pa $* v = \ a -> pa * (v a)
( <*> ) :: Vec a -> Vec b -> Vec (a, b)
v1 <*> v2 = \ (a, b) -> (v1 a) * (v2 b)
( <.> ) :: Basis a => Vec a -> Vec a -> K
v1 <.> v2 = sum [conjugate (v1 a) * (v2 a) | a <- basis]
```

Exemplos de vetores que podem ser construídos usando as funções descritas em cima de um conjunto de valores booleanos, são mostrados a seguir.

```
instance Basis Bool where
        basis = [False, True]

qFalse, qTrue, qFT, qFmT :: Vec Bool
qFalse = return False
qTrue  = return True
qFT    = (1/raiz_2) $* (qFalse 'mplus' qTrue)
qFmT   = (1/raiz_2) $* (qFalse 'mminus' qTrue)
```

Os dois primeiros representam os estados computacionais básicos, e os dois últimos uma superposição sobre esses estados. Isso significa que, `qFalse` equivale ao estado quântico $||0\rangle$, `qTrue` equivale ao estado quântico $||1\rangle$, `qFT` equivale a superposição $||\psi\rangle = (1/\sqrt(2)) * (||0\rangle + ||1\rangle$ e `qFmT` equivale a superposição $||\psi\rangle = (1/\sqrt(2)) * (||0\rangle - ||1\rangle$.

Estados multidimensionais podem ser descritos com a ajuda da função do produto tensorial:

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

p1,p2,p3 :: Vec Bool
p1 = qFT lan*ran qFalse
p2 = qFalse lan*ran qFT
p3 = qFT lan*ran qFT
```

4.1.2 Operadores Lineares

Dados dois conjuntos A e B, um operador linear pode ser visto como uma função $f :: A \mapsto B$, que mapeia um vetor sobre A para um vetor sobre B. Segundo a definição do `bind` é possível aplicar qualquer operador linear definido através de um tipo `(a -> Vec b)`, que na verdade é uma representação de uma *matriz*. Dessa forma, os operadores lineares são definidos em Haskell como:

```
type Lin a b = a -> Vec b
fun2lin :: (Basis a, Basis b) => (a -> b) -> Lin a b
fun2lin f a = return (f a)
```

A função `fun2lin` converte uma função clássica (reversível) para um operador linear. Por exemplo, a operação quântica *NOT* pode ser vista como a aplicação da função de negação do Haskell:

```
qnot :: Lin Bool Bool
qnot = fun2lin not
```

Pode se definir alguns operadores importantes para computação quântica seguindo o mesmo modelo:

```
phase :: Lin Bool Bool
phase False = return False
phase True  = (0:+1) $* return True

hadamard :: Lin Bool Bool
hadamard False = qFT
hadamard True  = qFmt

zgate :: Lin Bool Bool
zgate False = qFalse
zgate True = -1 $* qTrue
```

Essas definições dos operadores lineares especificam qual transformação será aplicada a cada elemento da base, isto é, a aplicação de um operador linear pode ser vista como uma multiplicação de matrizes e vetores. Para aplicar uma operações linear `f` em um vetor `v`, se usa a operação `bind` como `v >>= f`. Por exemplo, `qFT >>= hadamard` aplica a operação quântica hadamard em um vetor `qFT`, o que resulta no vetor `qFalse`, e pode ser ilustrada passo a passo como a seguir:

```
qFT >>= hadamard
 = \ b -> sum [(qFT a) (hadamard a b) | a <- [False,True]]
 = \ b -> if b == False then sum [(qFT False) (hadamard False False) +
                                  (qFT True) (hadamard True False)]
                        else sum [(qFT False) (hadamard False True)  +
                                  (qFT True) (hadamard True True)]
 = \ b -> if b == false then 1.0 else 0.0
```

No caso de *operações controladas*, é possível definir um método genérico para transformar operadores lineares em novos operadores lineares controlados por algum valor booleano.

```
controlled :: (Basis a) => Lin a a -> Lin (Bool,a) (Bool,a)
controlled f (b,a) = (return b) lan*ran (if b then f a else return a)
```

O operador linear modificado retorna um par cujo primeiro componente é o valor usado para o controle. O segundo componente é passado por uma transformação `f` somente se o

valor de controle é verdadeiro, senão ele é deixado intacto. Por exemplo, a operação controlada *CNOT* pode ser definida através de `controlled not`.

Os operadores lineares também podem ser combinados e transformados de diferentes formas. Dessa forma, as seguintes funções visam auxiliar a manipulação dos operadores lineares.

```
adjoint :: (Basis a, Basis b) => Lin a b -> Lin b a
adjoint f b a = conjugate (f a b)

( >*< ) :: (Basis a, Basis b) => Vec a -> Vec b -> Lin a b
(v1 >*< v2 ) a b = (v1 a) * (conjugate (v2 b))

linplus :: (Basis a, Basis b) => Lin a b -> Lin a b -> Lin a b
linplus f g a = f a 'mplus' g a

lintens :: (Basis a, Basis b, Basis c, Basis d ) =>
           Lin a b -> Lin c d -> Lin (a, c) (b, d )
lintens f g (a, c) = f a lan*ran g c

o :: (Basis a, Basis b, Basis c) => Lin a b -> Lin b c -> Lin a c
o f g a = (f a) >>= g
```

A função >*< gera um operador linear correspondente ao produto externo de dois vetores. As funções `linplus` e `lintens` são funções que correspondem a soma e produto tensorial sobre vetores. E a função `o` compõe dois operadores lineares.

4.2 Modelagem Quântica para lidar com Medidas

O modelo tradicional de vetores e operadores lineares não consegue expressar medidas de uma maneira eficiente. Para tanto, é preciso utilizar o modelo mais generalizado da computação quântica descrito na Seção 2.3. Com algumas mudanças, é possível adaptar o tipo `Vec` para representar uma matriz de densidade criando um novo tipo:

```
type Dens a = Vec (a,a)

pureD :: Basis a => Vec a -> Dens a
pureD v = lin2vec ( v>*<v)

lin2vec :: (a -> Vec b) -> Vec (a,b)
lin2vec = uncurry
```

A função `pureD` recebe um estado puro e o transforma na representação matriz de densidade. A chamada da função `uncurry` do próprio Haskell, é apenas uma conveniência para ordenar os argumentos numa forma parecida com uma matriz. Logo, os estados `qFalse`, `qTrue` e `qFT`, podem ser construídos como:

```
qFalseD,qTrueD,qFTD :: Dens Bool
qFalseD = pureD qFalse
qTrueD  = pureD qTrue
qFTD    = pureD qFT
```

A construção dos super-operadores pode ser descrita em *Haskell* como:

```
type Super a b = (a,a) -> Dens b

lin2super :: (Basis a, Basis b) => Lin a b => Super a b
lin2super f (a1,a2) = lin2vec (f a1 >*< f a2)
```

A função `lin2super` recebe um operador linear que pode ser aplicado em um vetor e retorna uma matriz de densidade que representa o seu respectivo super-operador. Por exemplo, um super-operador para representar a transformação quântica de *Hadamard* pode ser montado como:

```
hadamardS :: Super Bool Bool
hadamardS = lin2super hadamard
```

4.2.1 Medidas

Uma medida em um estado representado por uma matriz de densidade é representada colocando-se os coeficientes de probabilidade dos resultados clássicos apenas na diagonal principal. Isto é, setar como zero os outros valores e deixar apenas os valores referentes ao resultado da medida na diagonal com índices iguais. Essa *medida* pode ser expressada com a ajuda de um super-operador capaz de "esquecer", *projetar* ou *traçar* parte do estado quântico.

Trata-se de um tipo de medida baseado em *projeção*, que é descrito como um grupo de projeções sobre subespaços ortogonais entre si (VIZZOTTO; ROCHA COSTA; SABRY, 2008). Esse tipo de medida requer um operador para realizar a observação e outro para *projetar* o resultado. O primeiro deve retornar um valor clássico e o estado após a medida. E o segundo, deve "esquecer"o estado em colapso e retornar apenas o resultado clássico da medida. O primeiro operador pode ser visto como:

```
meas ::  Basis a => Super a (a,a)
meas (a1,a2) = if a1 == a2 then return ((a1,a1),(a1,a1)) else mzero
```

O operador `meas` consegue envolver ambos os resultados, retornando o resultado da medida junto com o estado após a medida. Esse operador considera uma matriz na forma $|m\rangle\langle m|$, como uma forma de representar um valor clássico m. Assim, como o resultado do operador `meas` retorna dois componentes, pode se entender porque é necessário projetar parte desse estado quântico. O segundo operador pode ser visto como:

```
trL ::  (Basis a, Basis b) => Super (a,b) b
trL ((a_1,a_1),(a_2,b_2)) = if a_1 == a_2 then return (b1,b2) else mzero
```

Dessa forma, uma medida pode ser expressada como um sequência desses operadores em um estado quântico. Por exemplo, o estado `qFTD` representado pela matriz de densidade

$$\begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}$$

pode ser medido através da sequência:

```
qFTD >>= meas >>= trL
```

o que resulta, assim como mostrado anteriormente, na matriz de densidade:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

4.2.2 O porquê de usar setas e não mônadas nesse modelo

Seguindo a abordagem de mônadas como vetores, é possível trocar a sintaxe para lidar matriz de densidade ao invés de vetores. Dessa forma, uma mônada cujo tipo se refere a uma matriz de densidade `Dens` é definida sem problemas como:

```
class Monad m where
return :: (Basis a) => a -> Dens a
return = pureD.return
```

Todavia, é impossível implementar um método do tipo *bind* para aplicar superoperadores nessas matrizes de densidade porque os argumentos requeridos para essa computação não serão compatíveis com a entrada. Por exemplo, um método genérico capaz de aplicar super-operadores na mônada seria:

```
(>>=) :: (Basis a, Basis b) => Dens a -> ((a,a) -> Dens b) -> Dens b
da >>= s = \ (b_1,b_2) -> sum [ ( da (a_1,a_2) * (s (a_1,a_2) (b_1,b_2) )
                                | a_1 <- basis, a_2 <- basis ]
```

Esse tipo `s` consome múltiplas entradas (neste caso `(a,a)`) e por isso esse modelo não pode ser abstraído como uma mônada. De certa forma, essa observação lembra a mesma motivação que levou a *generalização de mônadas para setas* (HUGHES, 2000). Sendo assim, Hughes além de criar uma nova noção de procedimento que pode realizar efeitos computacionais, definiu que as setas podem ter um componente estático independente da entrada ou que podem aceitar mais de uma entrada. Desse modo, as setas superam a limitação das mônadas e oferecem um solução elegante para modelar o modelo de matrizes de densidade e super-operadores para a computação quântica.

4.3 Setas Quânticas

A vantagem de usar o modelo de matrizes de densidade e super-operadores, é que é possível expressar as transformações quânticas e as medidas no mesmo formalismo. No entanto, essa modelagem não pode ser abstraída por mônadas, mas sim uma generalização delas chamada de *setas*. A interface das setas é definida como:

```
class Seta a where
arr   :: (Basis a, Basis b) => (b -> c) -> Super b c
(>>>) :: (Basis b, Basis c, Basis d) => Super b c -> Super c d -> Super b d
first :: (Basis b, Basis c, Basis d) => Super b c -> Super (b,d) (c,d)
```

A função `arr` recebe um função e retorna super-operador. O operador de composição de setas $\ggg$ é simplesmente uma composição de operadores lineares. A função `first` aplica o super-operador `f` ao primeiro componente e mantém o segundo inalterado. A definição dessas funções pode ser vista como:

```
arr :: (Basis b, Basis c) => (b -> c) -> Super b c
arr f = fun2lin (\ (b_1,b_2) -> (f b_1, f b_2))

(>>>) :: (Basis b, Basis c, Basis d) => Super b c -> Super c d -> Super b d
(>>>) = o

first :: (Basis b, Basis c, Basis d) => Super b c -> Super (b,d) (c,d)
first f ((b1,d1),(b2,d2)) = permute ((f (b_1,b_2)) <*> (return (d_1,d_2)))
        where permute v ((b_1,b_2),(d_1,d_2)) = v ((b_1,d_1),(b_2,d_2))
```

Essas funções com superoperadores foram implementadas de forma a satisfazer as requeridas equações de setas. Sendo assim, a função `arr` constrói o super-operador aplicando a função `f` no vetor e seu dual. O operador $\ggg$ apenas faz a chamada da função `o` descrita na Seção 4.1.2, e função `first` calcula cada parte separadamente, permutando o resultado para combinar com o tipo requerido para o retorno.

A implementação de circuitos quânticos faz o uso dessas três operações básicas das setas. Como as portas quânticas são modeladas para operar sobe apenas um qubit, o uso da operação `first` se faz necessário para computar a operação quântica apenas em parte da entrada, ou seja, aplica uma porta quântica apenas no primeiro componente. Para tanto, primeiro se deve montar uma seta com a operação `arr` e só depois combinar as computações. Por exemplo, a seta para aplicar de uma porta Hadamard no segundo qubit de um sistema de dois qubits é definida como:

```
had2_2 :: Super (Bool,Bool) (Bool,Bool)
had2_2 =
```

```
 let hadS = lin2super hadamard
 in  arr ( \ (a_0, b_0) -> (b_0, a_0)) >>>
     (first hadS >>> arr (\ (b_1, a_0)) -> ((a_0, b_1)))
```

A primeira operação `arr` monta uma seta de `(a,b)` para `(b,a)` para que na sequência, ela seja combinada com uma operação `first hadS` para aplicar a porta *Hadamard* no segundo qubit. Nesse ponto, poderiam ser adicionadas outras operações mas nesse caso como chegou ao fim da computação apenas monta-se a seta a seu estado de origem colocando o valor de `a` no primeiro componente do par e `b` no segundo.

Outro exemplo de aplicação das setas, são os primeiros passos do circuito de Toffoli da Figura 2.3:

```
toffoli :: Super (Bool,Bool,Bool) (Bool,Bool,Bool)
toffoli =
 let hadS = lin2super hadamard
     cphaseS = lin2super (controlled phase)
     cnotS = lin2super (controlled qnot)
 in  arr ( \ (a_0, b_0, c_0) -> (c_0, (a_0, b_0))) >>>
     (first hadS >>> arr (\ (c_1, (a_0, b_0)) -> ((b_0, c_1), a_0))) >>>
     (first cphaseS >>> arr (\ ((b_1, c_2), a_0) -> ((a_0, b_1), c_2))) >>>
     (first cnotS >>> arr (\ ((a_1,b_2), c_2) -> ((b_2, c_2), a_1))) >>> ...
```

Nesses dois exemplos, pode-se notar que a sintaxe utilizada pelas setas precisa manipular e permutar os componentes de todo estado quântico manualmente. Dessa forma, a fim de oferecer uma sintaxe mais prática e clara para a notação das setas, Paterson (2001) apresentou uma extensão para o Haskell para incorporar uma sintaxe melhorada para setas, similar a notação-do das mônadas. Assim, pode-se reescrever o primeiro exemplo como:

```
had2_2 :: Super (Bool,Bool) (Bool,Bool)
had2_2 = proc (a,b) -> do
 b_1 <- lin2super hadamard -< b
 returnA (a,b_1)
```

A notação-do serve para sequenciar as ações dentro de um corpo, similar ao que acontece em linguagens imperativas. A função `returnA` é o equivalente para setas da função `return` das mônadas. O comando `proc` representa a abstração de uma seta, o símbolo `<-` é usado para atribuir o valor de uma expressão em uma variável e o símbolo `-<` é a aplicação de uma expressão em uma seta.

Utilizando essa nova notação é possível criar códigos mais claros e rápidos de escrever. O algoritmo de Toffoli pode ser reescrito como:

```
toffoli :: Super (Bool, Bool, Bool) (Bool, Bool, Bool)
toffoli = let hadS = lin2super hadamard
              cnotS = lin2super (controlled qnot)
```

```
    cphaseS = lin2super (controlled phase)
    caphaseS = lin2super (controlled (adjoint phase))
in proc (a0,b0,c0) -> do
   c1 <- hads -< c0
   (b1,c2) <- cphaseS -< (b0,c1)
   (a1,b2) <- cnotS -< (a0,b1)
   (b3,c3) <- caphaseS -< (b2,c2)
   (a2,b4) <- cnotS -< (a1,b3)
   (a3,c4) <- cphaseS -< (a2,c3)
   c5 <- hadS -< c4
   returnA -< (a3,b4,c5)
```

5 QJAVA: SETAS QUÂNTICAS EM JAVA

Neste capítulo é descrita a implementação da biblioteca em Java para programação quântica, chamada de *QJava*. Além da modelagem de mônadas e setas quânticas para uma linguagem orientada a objetos, são também apresentados os outros componentes auxiliares necessários para construção da biblioteca.

5.1 Visão Geral

O objetivo da biblioteca é oferecer uma ferramenta de alto nível para a programação quântica. Essa necessidade surge porque ainda não existem fortes ferramentas voltadas a programadores que facilitem a elaboração e compreensão de algoritmos quânticos. Por esse motivo, a biblioteca foi implementada na linguagem de programação orientada a objetos Java, por ser uma das mais utilizadas pela comunidade da Computação.

Para conseguir abstrair os conceitos da Computação Quântica, o sistema foi implementado com o modelo de mônadas e setas quânticas demostrado no Capítulo 4 . Essa implementação só foi possível em Java, pois recentemente, através do *Lambda Project* (ORACLE, 2013a), foram incorporados em seus recursos funções anônimas (closures), permitindo a implementação de mônadas e setas no contexto de uma linguagem orientada a objetos. Sendo assim, fazendo o uso de Java Closures, o sistema projetado é capaz de expressar algoritmos quânticos, incluindo operações reversíveis e medidas, através de mônadas e setas quânticas.

Primeiramente neste Capítulo serão descritas as características da linguagem Java, com ênfase em funções anônimas. E em seguida, será apresentada a arquitetura da biblioteca de setas quânticas em Java e seus componentes.

5.1.1 Java Closures

Denomina-se closure uma expressão lambda que teve suas variáveis livres fechadas por um ambiente léxico (LANDIN, 1964), isto é, uma *closed expression.* É um tipo de *função anônima* tipicamente utilizado em linguagens funcionais Com closures se pode fazer a passagem de um trecho de código como argumento para uma função e criar de funções dinâmicas, em tempo de execução.

Em termos convencionais um closure pode ser definido como um tipo de função anônima

com uma sintaxe mais compacta, que também permite a omissão de modificadores, tipo de retorno, e, em alguns casos, tipos de parâmetros. Sua sintaxe básica em Java é:

```
(parâmetros) -> expressão
```

ou

```
(parâmetros) -> { declarações;}
```

Exemplos:

```
1 (int x, int y) -> x + y
2 (x, y) -> x - y
3 () -> 42
4 (String s) -> System.out.println(s)
5 x -> 2 * x
6 c -> { int s = c.size(); c.clear(); return s; }
```

A primeira função recebe dois valores inteiros e retorna a sua soma. A segunda recebe dois valores e retorna a sua diferença. A terceira não recebe nenhum valor e retorna o número 42. A quarta recebe uma string, imprime seu valor no console, e retorna nenhum valor. A quinta recebe um número e retorna o seu dobro. A sexta por fim, recebe uma coleção, limpa-a e retorna o seu tamanho anterior.

Note que tipos de parâmetros podem ser explicitamente declarados (ex. 1/4) ou implicitamente inferidos (ex. 2/5/6) entretanto, não podem ser misturados numa única expressão lambda. O corpo da função pode ser um bloco (cercado por chaves, ex 6) ou uma expressão (ex. 1 ao 5). Uma função pode ter um valor de retorno ou nada (void). Se o corpo é apenas uma expressão, ele pode retornar um valor (ex. 1/2/3/5) ou nada (ex. 4) sem precisar explicitamente declarar a instrução de retorno (return). Parênteses podem ser omitidos para um único parâmetro inferido do tipo (ex. 5/6).

Para atribuir uma função dinâmica a uma variável em Java e, posteriormente, usá-la para invocar o método criado ou apenas a passar como argumento para outra função, é necessário o uso de *interfaces funcionais*. Tais interfaces funcionais podem ser vistas como qualquer interface que possua exatamente um método abstrato declarado. Entre tantas, pode-se citar como interfaces funcionais as seguintes, disponibilizadas pela plataforma Java:

```
public interface Runnable { void run(); }
public interface Callable<V> { V call() throws Exception; }
public interface ActionListener {
   void actionPerformed(ActionEvent e);}
```

Como exemplo simples, considera-se uma função dinâmica que apenas imprime qualquer frase na tela do console. Essa função anônima é atribuída a uma variável local, que pos-

teriormente poderá invocar o novo método criado. Para esse caso, não é necessário criar uma nova interface funcional, pode-se usar:

```
Runnable c = () -> () -> { System.out.println("Ola mundo!"); };
c.run();
```

O resultado do trecho de código acima vai ser imprimir no console a frase "Olá mundo!". A chamada do closure é feita através do método *run*, o único método declarado dentro da interface funcional *Runnable*.

Um closure é capaz de capturar variáveis, ou seja, pode utilizar variáveis dentro de um escopo, mesmo que este último não esteja ativo no momento de sua chamada. Se ele é passado como um argumento para um método, vai continuar usando as variáveis do escopo onde foi criado. Em Java, apenas variáveis estáticas podem ser alteradas dentro do bloco de uma função anônima. Além disso, a captura de variáveis locais só é possível se forem efetivamente finais, isto é, seu valor inicial nunca é alterado, veja-se:

```
final String str = "Ola Mundo!";
Runnable c = () -> { System.out.println(str); };
c.run();
```

Nesse caso simples, a string "str"é uma variável local final capturada pelo closure "c". Se o closure for passado como argumento para alguma função, ele ainda poderá acessar o valor da string. Outra alternativa, é definir uma variável estática para a string, o que permitiria, além do acesso ao valor da variável, a possibilidade de alterar seu conteúdo. Essa limitação na captura de variáveis locais previne *bugs* na implementação de closures em Java (NAFTALIN, 2013).

5.2 Quantum Java

Quantum Java ou QJava é o nome atribuído para a biblioteca de programação quântica desenvolvida neste trabalho. Essa biblioteca é projetada sobre uma modelagem orientada a objetos que adapta a implementação em Haskell de mônadas e setas quânticas. Para tanto, foram utilizados os novos recursos de funções anônimas, já disponíveis nativamente dentro da prévia da JDK 8, e classes auxiliares para manipular tuplas e números complexos.

A biblioteca foi modularizada em uma série de pacotes: *QMonad*, *QArrows*, *Tuples* e *Basis*. Um pacote em Java é um mecanismo utilizado para organizar as classes em grupos como se fossem organizadas em diretórios. Dessa forma, os pacotes foram organizados da seguinte forma: o pacote *QMonad* contém as classes que representam as abstrações de uma

monada quântica; o pacote *QArrow* as classes que representam as abstrações da seta quântica, o pacote *Tuples* as classes de manipulação de um par de valores ou uma tupla de *n* valores[6] e o pacote *Basis* as classes de manipulação e construção das bases vetoriais usadas no modelo da computação quântica. A Figura 5.1 ilustra a arquitetura da biblioteca QJava organizada em pacotes. Serão descritos nas próximas seções os detalhes sobre cada classe presente dentro dos pacotes.

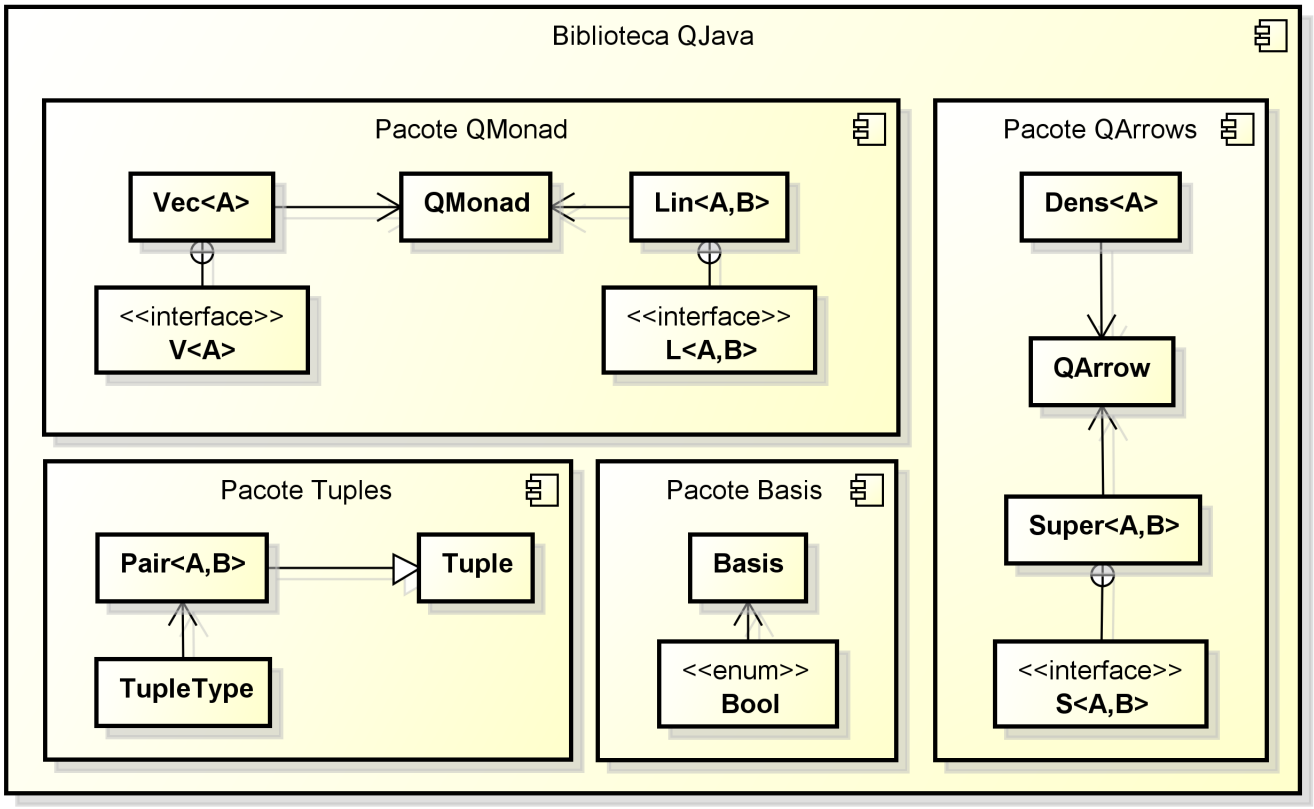


Figura 5.1 – Arquitetura da Biblioteca QJava

5.2.1 Pacote Tuples

Um tupla é um tipo de estrutura de dados contendo uma lista ordenada de *n* elementos que, ao contrário de visão tradicional de *arrays*, pode conter valores de tipos diferentes. Naturalmente, uma dupla ou par é uma tupla de dois elementos, uma tripla é uma tupla de três elementos, e um *n*-tuple, uma tupla de *n* elementos. Sua sintaxe utiliza a representação de parênteses para identificar cada membro da tupla, por exemplo, (42, "string") é uma tupla cujo primeiro elemento é um valor do tipo inteiro e o segundo um valor do tipo string.

[6] Pois em Java não existe suporte nativo a tuplas

Essa estrutura é indispensável para o presente estudo, pois manipula-se as bases vetoriais de um estado quântico através de um tipo *T*. Esse tipo pode ser um valor booleano (um estado de um qubit), mapeando *false* e *true* para seus respectivos coeficientes de probabilidade, ou uma dupla *(boolean, boolean)* (um estado de dois qubits), mapeando *(false, false)*, *(false, true)*, *(true, false)* e *(true, true)* para seus quatro respectivos coeficientes, ou uma tripla *(boolean, boolean, boolean)* (um estado de três qubits), mapeando *(false, false, false)*, *(false, false, true)*, *(false, true, false)*, *(false, true, true)*, *(true, false, false)*, *(true, false, true)*, *(true, true, false)*, *(true, true, true)* para seus respectivos coeficientes, e assim por diante.

As tuplas estão presentes tanto em linguagens funcionais (Haskell, Prolog) quanto imperativas (Ruby, Phyton), mas ainda não foram incorporadas em Java. Em razão disso, foram criadas três classes em Java, que em conjunto conseguem manipular abstrair o conceito de tuplas. Dessa forma, estão presentes no pacote Tuples as classes: *Pair*, *Tuple* e *TupleType*. A classe *Pair* é utilizada para manipular um par de valores, ou seja, um tupla de dois elementos, e a classe *Tuple* é utilizada para manipular uma tupla de *n* elementos. A classe *TupleType* está presente como atributo em todas as tuplas e é utilizada para manipular os diferentes tipos de valores presentes nelas.

A classe *Pair* recebe dois argumentos de tipos genéricos `A` e `B` e manipula seus valores através dos métodos `getA()` e `getB()`:

```
public class Pair<A,B>{
   private final TupleType type;
   private final A a;
   private final B b;

   public Pair(A a, B b){
      Class<?>[] types = new Class<?>[]{a.getClass(), b.getClass()};
      this.type = new TupleType(types);
      this.a = a;
      this.b = b;
   }

   public A getA(){
      return a;
   }

   public B getB(){
      return b;
   }
   ...
}
```

Seguindo o critério de que uma tupla é uma sequência de duplas, existe uma herança da classe *Pair* para uma classe *Tuple*. Isto significa que um par de valores pode ser tanto uma

instância da classe *Pair* quanto da classe *Tuple*. A diferença é que a classe Tuple estrutura os elementos como uma lista e manipula os valores com uma função `getNthValue(int i)`, sendo `i` a posição da tupla.

```
public class Tuple extends Pair{
    private final Object[] values;

    public Tuple(Object a, Object b) {
        super(a, b);
        values = new Object[]{a,b};
    }

    public Tuple(Object... values){
        super(values);
        this.values = new Object[values.length];
        System.arraycopy(values, 0, this.values, 0, values.length);
    }

        public <T> T getNthValue(int i) {
                return (T) values[i];
        }

    ...
}
```

A classe *TupleType* manipula os diferentes tipos de valores que estão presentes em uma tupla através de uma lista com os tipos. Ela impede que um tupla acrescente um novo valor que não seja de seus respectivos tipos. Por exemplo, uma tupla *(boolean, int, char)* apenas aceita um valor booleano para o elemento da primeira posição da tupla, um valor inteiro para a segunda posição e um caractere para a terceira posição.

Na Figura 5.4 encontra-se o resumo do pacote Tuples contendo as principais funções utilizadas para manipular tuplas.

5.2.2 Pacote Basis

Devido ao fato de que um vetor quântico é manipulado por um tipo *T*, torna-se necessário gerar todos os possíveis valores que esse tipo pode assumir para ser usado como as bases ortonormais. Em Haskell, devido ao suporte natural a tuplas, isso pode ser criado em poucas linhas, já em Java, é preciso criar uma série de funções auxiliares para realizar tal tarefa. Sendo assim, no pacote *Basis* estão presentes uma interface do tipo *enum Bool*, utilizada para conter os possíveis valores de um tipo booleano, e uma classe *Basis*, utilizada para criar e manipular as listas de bases de um referido tipo *T*.

Por exemplo, para criar uma lista das bases para um tipo *A*, primeiramente a classe

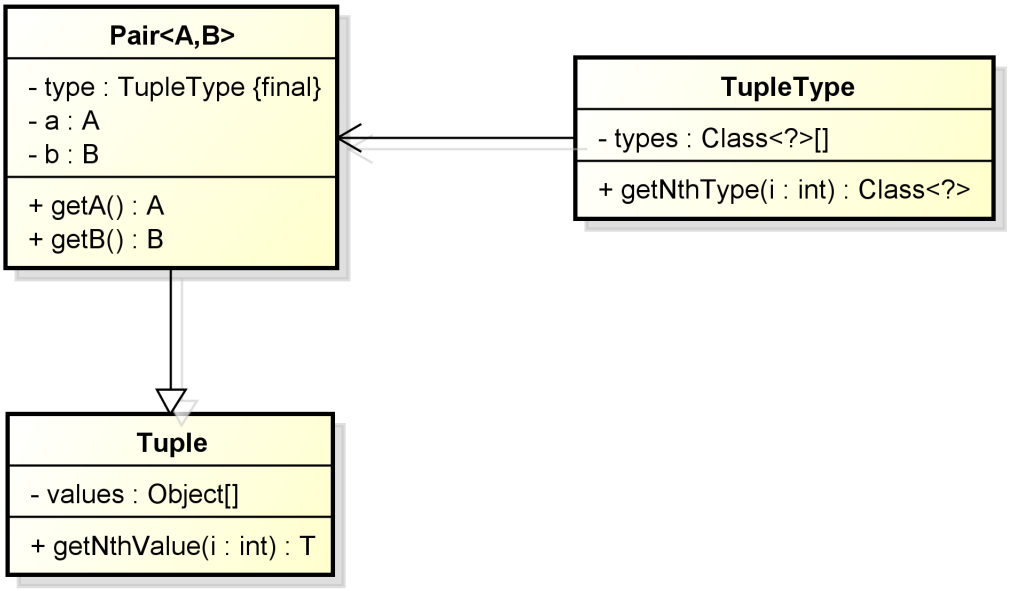


Figura 5.2 – Pacote Tuplas

Basis verifica se ele é um tipo booleano ou uma tupla. Na sequência, cria-se uma lista de todos os possíveis valores percorrendo a enumeração e usando permutação caso necessário. Assim, dado um valor básico booleano, ela retorna uma lista *[false,true]*, dado uma tupla *(booleano,booleano)* ela retorna uma permutação das listas *[false,true]* e *[false,true]*, isto é, *[(false,false), (false,true), (true,false), (true,true)]*, e assim por diante.

Por questões de otimização, a lista de bases é criada apenas uma vez. Assim, antes de gerar uma lista, é verificado se a lista de bases para um tipo T já foi criada. Em caso afirmativo, ela é retornada, se não, cria-se a lista de bases segundo a descrição acima e depois retorna a nova lista. A função que faz essa verificação e retorno é a `getBasis(A tipo)`, que por sua vez, tem aplicação direta no cálculo das transformações unitárias em estados quânticos, sendo ela a única forma de um método *bind* conseguir percorrer a lista de bases e combinar computações.

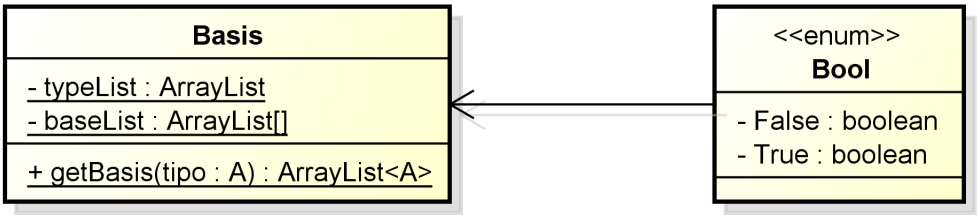


Figura 5.3 – Pacote Basis

Na Figura 5.3 encontra-se o diagrama de classes do pacote Basis.

5.2.3 Pacote QMonad

No pacote QMonad estão presentes as classes utilizadas para implementar as mônadas quânticas descritas na Seção 4.1. As funções da implementação em Haskell, foram moduladas para uma abordagem orientada a objetos. Sendo assim, existe uma classe `Vec`, para abstrair um vetor quântico e uma classe `Lin` para abstrair um operador linear. E as funções monádicas, `return` e `bind`. estão estruturadas dentro de uma classe *QMonad*. Além disso, estão presentes também as interfaces funcionais utilizadas para operar com Java Closures.

Uma interface funcional para descrever uma função anônima que recebe um tipo `A` e retorna um número complexo é definida como:

```
public interface V<A> {
   Complex invoke(A bool);
}
```

A classe `Vec` possui um atributo `vec` para representar um estado quântico descrito como um closure segundo a interface funcional acima. Por exemplo, o estado quântico $|1\rangle = 0 * |0\rangle + 1 * |1\rangle$ pode ser representado assimilando o seguinte valor a variável `vec`:

```
V<Boolean> vec = (Boolean bool) -> {
   return bool?Complex.ONE:Complex.ZERO;
}
```

Além dos métodos para abstrair um estado quântico, estão presentes dentro da classe `Vec` todos os métodos utilizados para realizar as operações vetoriais. Esses métodos foram implementados como métodos estáticos, isto é, métodos de classe que podem ser chamados sem precisar explicitamente de uma instância da classe. Existe um método `mzero(A tipo)` para retornar um vetor que mapeia todos os tipos de `A` para zero, um método `mplus` e `mminus` para fazer somar e subtrair dois vetores quânticos de mesma base `A`, um método `scalar`, que representa a multiplicação simples de um vetor por um número complexo, ou seja, multiplica todos os coeficientes do vetor pelo número informado, um método `tensor`, que recebe dois vetores de bases diferentes`Vec<A>` e `Vec<B>`, que também podem ser do mesmo tipo, e calcula o produto tensorial, e por fim, um método `outer` recebe dois vetores de mesma base e calcula o produto interno.

A classe *QMonad* possui apenas os métodos estáticos necessários para a manipulação da monada quântica. O método `qreturn`, que é usado para construir os vetores quânticos básicos, recebe um valor do tipo `A` e mapeia o valor do argumento para um e todos os outros possíveis valores para zero.

```
public static <A> Vec<A> qreturn(A base) {
   return new Vec<A>(base, (A bool) -> {
      return bool.equals(base)?Complex.ONE:Complex.ZERO;
   });
}
```

Assim, pode-se criar os estados quânticos básicos $|0\rangle$ e $|1\rangle$, da seguinte forma:

```
Vec<Boolean> vzero = qreturn(false);
Vec<Boolean> vone  = qreturn(true);
```

O outro método da classe das mônadas, `bind` [7], é o responsável por combinar computações. No caso da mônada quântica, esse método deve calcular o resultado de uma transformação unitária sobre um vetor, o que é feito através da noção de uma multiplicação de matrizes. O método de composição implementando no Java, recebe um vetor `va`, do tipo `Vec<A>` e uma transformação `f`, do tipo `Lin<A,B>`, e retorna um novo vetor do tipo `Vec<B>` contendo o resultado da aplicação da transformação `f` em `va`.

```
public static <A, B> Vec<B> bind(Vec<A> va, Lin<A, B> f) {
   return new Vec<B>(f.invoke(va.getBase()), (B b) -> {
      Complex soma = Complex.ZERO;
      for (A a : va.getBases())
         soma = soma.plus(va.invoke(a).times(f.invoke(a).invoke(b)));
      return soma;
   });
}
```

A classe `Lin<A,B>` descreve os operadores lineares como uma função `A` $\mapsto$ `Vec<B>` associada a uma variável `lin`. A interface funcional que descreve esse closure é definida como:

```
public interface Lin<A,B> {
   Vec<B> invoke(A bool);
}
```

Tais transformações lineares são matrizes unitárias, moduladas como uma função que recebe um tipo `A`, a linha da matriz unitária, e retorna um `Vec<B>`, o vetor da linha. Por exemplo, a operação `NOT` $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ é representada pelo closure:

```
L<Boolean,Boolean> lin = (Boolean bool) -> {
   !bool?QMonad.qreturn(true):QMonad.qreturn(false);
};
```

Por questões de projeto, as operações quânticas mais comuns já estão disponíveis para uso dentro da biblioteca. Elas podem ser chamadas através dos métodos estáticas `qnot()`,

[7] Aqui, os métodos `plus` e `times` são provenientes da biblioteca de números complexos do Java, e significam a soma e a multiplicação de dois números complexos respectivamente.

`qphase()` e `qhadamard()`, que retornam, respectivamente, a transformação linear representando a operação quântica `Not`, `Phase` e `Hadamard` descritas na Seção 2.2.1. Além disso, existe uma função genérica `controlled()`, que consegue retornar à transformação controlada de uma dada transformação linear. Podemos criar a operação `CNOT` por exemplo, com a chamada da função `controlled(qnot())`.

Como exemplo simples da aplicação de uma transformação em um estado quântico, considera-se o seguinte código:

```
Vec<Boolean> vzero = qreturn(false);
Vec<Boolean> result = bind(vzero, qnot());
```

Nesse caso, o resultado da aplicação da operação `Not` no vetor $|0\rangle$, por exemplo, é um vetor representando o estado quântico $|1\rangle$.

Assim como existem funções estáticas para lidar com as operações sobre vetores, a classe `Lin` apresenta uma série de métodos estáticos para manipular as operações sobre operadores lineares. Existe um método `fun2lin`, que recebe uma função qualquer de `A` $\mapsto$ `B` e converte em um operador linear, um método `adjoint`, que retorna a operação adjunta do operador linear passado como argumento, um método `outer`, que produz um operador linear correspondente ao produto interno de dois vetores, um método `linplus` e `lintens`, que correspondem a soma e produto tensorial sobre operadores lineares. E por fim, um método `compose` recebe dois operadores lineares e retorna um novo operador linear representando a composição dos dois operadores em um só.

É importante destacar que tanto a classe para encapsular um vetor quântico quanto a classe para manipular operadores lineares possuem uma lista contendo os possíveis valores de suas bases. Tal procedimento se mostrou necessário, porque os tipos genéricos em Java usam `type erasure`, ou seja, em tempo de execução é impossível descobrir qual o tipo específico de um tipo genérico `A`[8]. Por exemplo, dado uma instância da classe `Vec<A>`, não há possibilidade de extrair e verificar qual é a o tipo de `A` e, portanto, não se sabe qual base precisa ser usada para percorrer o vetor. Esse problema foi solucionado adicionando em cada construtor de um vetor ou operador linear a referida base que deve usada. Assim, o construtor cria uma lista dos possíveis valores de `A` usando o método `getBasis(tipo)` na criação de cada novo objeto. O mesmo vale para super operadores e matrizes de densidades, os quais serão analisados na próxima seção.

[8] Mesmo por reflexão, a única informação que se consegue extrair é a classe Object, mas todos os objetos em Java estão ligados a classe Object...

Na Figura 5.3 encontra-se o diagrama de classes com as principais funções e atributos do pacote das mônadas.

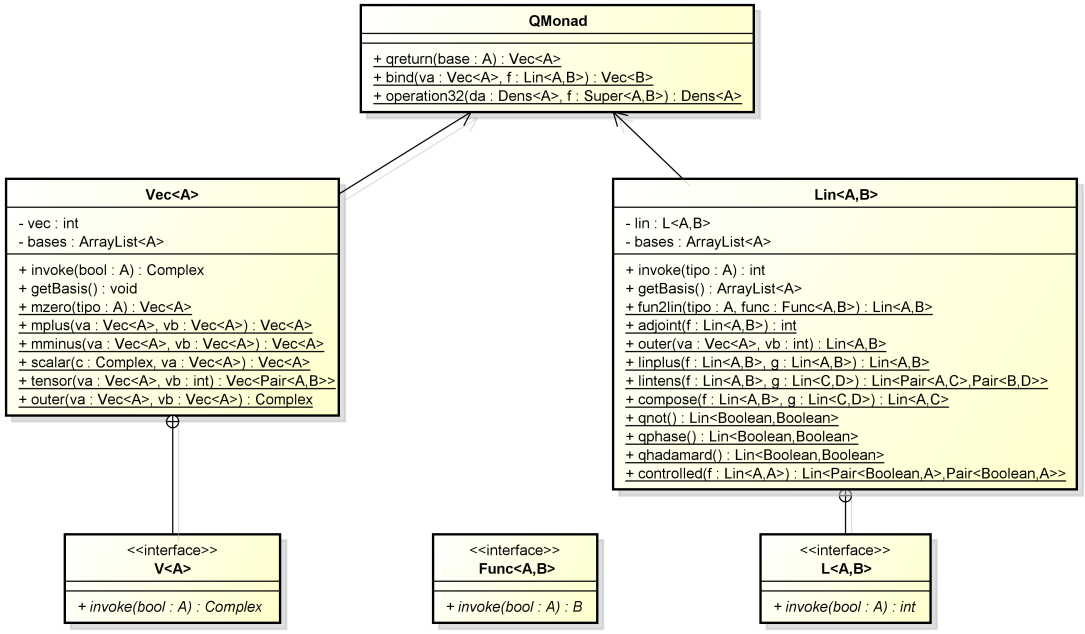


Figura 5.4 – Pacote QMonad

5.2.4 Pacote QArrows

No pacote *QArrows* estão presentes as classes utilizadas para implementar as setas quânticas descritas na Seção 4.3, que é modelo baseado em matrizes de densidade, para representar estados quânticos, e super operadores, para representar as transformações unitárias. Nesse pacote estão presentes as classes *Dens*, *Super*, *QArrows* e as interfaces funcionais necessárias.

No caso da representação de uma matriz de densidade com closures, não é necessário criar uma nova interface funcional. Pode-se reutilizar a mesma interface dos vetores quânticos mas como uma pequena modificação, como demonstrado a seguir:

```
public class Dens<A> {
    public final Vec<Pair<A,A>> dens;
    private ArrayList<Pair<A,A>> bases;

    public Dens (A tipo, V<Pair<A,A>> v){
        bases = Basis.getBasis(new Tuple(tipo,tipo));
        this.dens = new Vec<Tuple>(t,v);
    }
    ...
}
```

Uma matriz de densidade é descrita como uma função anônima que mapeia um tipo `Pair<A,A>` para um número complexo. Dessa forma, uma matriz de densidade é apenas um nova classe, encapsulando um vetor `Vec<Pair<A,A>>`. Para converter um vetor puro na representação de matriz de densidade, a classe `Dens` oferece os seguintes métodos estáticos:

```
   public static <A> Dens <A> pureD (Vec<A> v){
      return new Dens<>((lin2vec (Lin.outer(v, v))));
}

   public static <A,B> Vec<Pair<A,B>> lin2vec(Lin<A,B> f){
      A a = f.getBase();
      B b = f.invoke(a).getBase();
      return new Vec<> (
         new Pair<A,B> (a,b),
         (Pair<A,B> p) -> {
            return f.invoke(p.A()).invoke(p.B());
         }
      );
   }
```

A função `pureD` recebe um vetor quântico `Vec<A>` e retorna a sua representação por uma matriz de densidade, usando uma função auxiliar `lin2vec`, que recebe o produto externo de um vetor puro com ele mesmo (formando um novo operador linear) e o converte para a forma de vetor. Por exemplo, a matriz de densidade que representa o vetor quântico $|0\rangle$ pode ser criada desta forma:

```
   Dens<Boolean> dens = pureD(QMonad.qreturn(false));
```

ou ainda, utilizando a variável criada na seção anterior:

```
   Dens<Boolean> dens = pureD(vzero);
```

Os super operadores necessários para executar as computações em um estado quântico, representado por matrizes de densidade, podem ser obtidos convertendo a transformações lineares descritas na seção anterior. Um superoperador é representando como uma função que mapeia um `Pair<A,A>` para um vetor `Vec<Pair<B,B»` descrito através da interface funcional:

```
   public interface S<A, B> {
      Dens<B> invoke(Pair<A,A> bool);
   }
```

A conversão de um operador linear em um super operador é atingida através da função `lin2super`:

```
public static <A, B> Super<A, B> lin2super(Lin<A, B> f) {
    return new Super<>(
    f.getBase(),
    (Pair<A,A> p) -> {
        Vec<Tuple> novo = Dens.lin2vec(Lin.outer(f.invoke(p.getNthValue(0)),
        f.invoke(p.getNthValue(1))));
        return new Dens<>(novo);
    }
);
```

Os métodos das setas `arr`, `bind` e `first`, são implementados na classe *QArrows* como métodos estáticos. O método `arr` recebe um função de `B` para `C` e retorna um super operador convertendo a função para um operador linear[9].

```
public static <B,C> Super<B,C> arr (B tipo, Func<B,C> func) {
    Lin<Pair<B,B>,Pair<C,C>> novo = Lin.fun2lin(
        new Tuple(tipo,tipo),
        (Pair<B,B> parBB) -> {
            return new Pair<C,C>(
                func.invoke(parBB.getA()),
                func.invoke(parBB.getB())
            );
        }
    );
    return new Super<>(tipo, cast(novo));
}
```

A composição de setas, `bind`, é simplesmente a computação de dois operadores lineares:

```
public static <B,C,D> Super<B,D> bind (Super <B,C> a1, Super <C,D> a2){
    Lin<Pair<B,B>,Pair<C,C>> l1 = new Lin<> (a1.tipo, cast(a1));
    Lin<Pair<C,C>,Pair<D,D>> l2 = new Lin<> (a2.tipo, cast(a2));
    Lin<Pair<B,B>,Pair<D,D>> l3 = LinearOperations.compose(l1,l2);
    return new Super<>(l3.getBase().getNthValue(0), cast(l3));
}
```

O método `first`[10] aplica uma transformação `f` no primeiro componente de uma seta e deixa o segundo componente intacto.

```
public static <B,C,D> Super<Pair<B,D>,Pair<C,D>> first (D tipo,Super<B,C> f){
    return new Super<>(
        new Pair<B,D> (f.tipo.getA(),tipo), (Pair<Pair<B,D>,Pair<B,D>> p) -> {
            Vec<Pair<C,C>> va = f.invoke(new Pair<>
             (p.getA().getA(),p.getB().getA())).dens;
            Vec<Pair<D,D>> vb = Monad.qreturn(new Pair<> (p.getA().getB(),
             p.getB().getB()));
            Vec<Pair<Pair<C,C>,Pair<D,D>>> vv = Vec.tensor(va,vb);
            return new Dens<Pair<C,D>> (permute(vv));
        }
    );
}
```

[9] Onde `cast` é uma função que realiza a conversão de vetor `Vec<Pair<A,A>>` em uma matriz de densidade `Dens<A>`. Em Haskell, ele consegue deduzir que as duas funções são parecidas mas em Java um objeto Vec é diferente de um Objeto Dens e por isso dá erro de execução.

[10] O método `permute` apenas faz a permutação das bases recebidas para o resultado esperado. Ele recebe um tupla *((B1,B2), (D1,D2))* e retorna a conversão para *((B1,D1), (B2,D2))*

Esse método, em conjunto com o `arr`, e usado para montar as setas de acordo com a operação que se deseja realizar. Por exemplo, para aplicar a porta `Not` no segundo qubit de um estado de dois qubits, representado por uma matriz de densidade, basta combinar a seta:

```
Pair bool_bool = new Pair(false, false);
Super<Boolean, Boolean> hadS = Super.lin2super(Lin.qhadamard());

Super<Pair<Boolean,Boolean>, Pair<Boolean,Boolean>> had1_2 = QArrows.bind(
   QArrows.arr(
      bool_bool,
      (Pair<A,B> ab) -> {
         return new Pair(ab.getB(),ab.getA());
      }),
      QArrows.bind(
         QArrows.first(true, hadS),
         QArrows.arr(
            bool_bool,
            (Pair<A,B> ba) -> {
                        return new Pair(ba.getB(),ba.getA());
            }
         )
      )
);
```

Como uma das motivações que levaram ao uso de setas, ao invés de mônadas para representar estado quânticos, é a possibilidade de expressar medidas, a biblioteca QJava oferece em suas funcionalidades dos métodos estáticos `trL` e `meas`, incorporados na classe `Super`, para a realização das medidas sobre estado quântico. O primeiro projeta (*traces out*) parte de um estado quântico, o segundo, por sua vez, executa a medida também em uma parte de um estado quântico. Essas duas operações só podem ser expressadas através de super-operadores e são o grande diferencial deste modelo semântico para a computação quântica.

```
   public static <A,B> Super<Pair<A,B>,B> trL (Vec<Pair<A,B>> va){
      return new Super<> (va.tipo, (Pair<Pair<A,B>,Pair<A,B>> pp) -> {
      if (pp.geA().getA().equals(pp.getB().getA()))
         return new Dens<B> (Monad.qreturn(new Pair<B,B>
         (pp.getA().getB(),pp.getB().getB())));
      return new Dens<B> (Vec.mzero(
         new Pair<B,B>(va.tipo.B(),va.tipo.B())));
      }
   );
}

   public static <A> Super<A,Pair<A,A>> meas (Vec<Pair<A,A>> va){
      return new Super<> (va.tipo.getValue0(), (Pair<A,A> p) -> {
         if (p.getA().equals(p.getB()))
            return new Dens<> (Monad.qreturn(new Pair<Pair<A,A>,Pair<A,A>>(
               new Pair<>(p.getA(),p.getA()),
               new Pair<>(p.getB(),p.getB())
            ))
         );
```

```
        return new Dens<>(Vec.mzero(new Pair<Pair<A,A>,Pair<A,A>>(p,p)));
        }
    );
}
```

Por exemplo, dado um estado em superposição entre $|0\rangle$ e $|1\rangle$, `qFT`, que é construído com as operações de soma e produto escalar sobre vetores. Uma matriz de densidade que representa esse estado pode sofrer uma operação de medida na seguinte forma:

```
public static void testMedida() {
    Vec<Boolean> qFalse, qTrue, qFT;
    qFalse = QMonad.qreturn(false);
    qTrue = QMonad.qreturn(true);
    qFT = Vec.scalar(Complex.valueOf(1 / Math.sqrt(2), 0),
          Vec.mplus(qFalse, qTrue));

    Dens<Boolean> ex = Dens.pureD(qFT);
    Dens<Pair<Boolean,Boolean>> teste = QMonad.bind(ex, Super.meas(ex.dens));
    Dens<Boolean> medida = QMonad.bind(teste, Super.trL(ex.dens));
}
```

O resultado de uma medida nesse estado deve ser o valor *zero* com probabilidade 1/2 e no valor *um* com probabilidade 1/2. Essa informação, que não pode ser expressada usando vetores, é representada pela matriz de densidade $\begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$ que é o resultado da sequência de operações acima.

Na Figura 5.5 está o diagrama de classes do pacote setas (arrows).

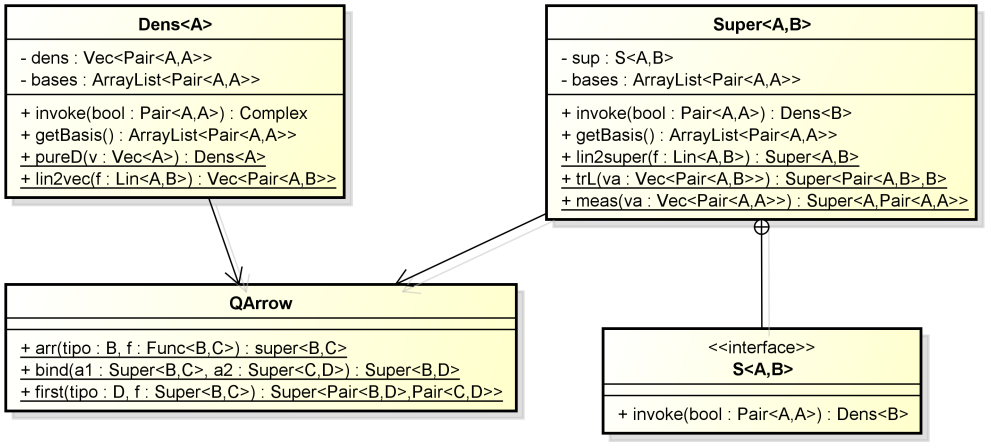


Figura 5.5 – Pacote QArrow

A biblioteca QJava desenvolvida oferece um ambiente de programação para os algoritmos quânticos fazendo uso das diversas funções implementadas para o modelo de mônadas e setas quânticas. Contudo, apresenta algumas restrições. Na questão de desempenho, uma

sequência de chamadas das funções de setas para representar os algoritmo quânticos demostrou ter um custo exponencial de processamento, o que sugere que essa biblioteca passe por otimizações.

Além disso, a sequenciação de operações quânticas dos algoritmos quânticos, e consequentemente a composição de funções com closures, utilizam uma sintaxe extensa e falha no propósito do trabalho de oferecer uma ferramenta de alto nível para a programação quântica, motivo pelo qual levou este trabalho a implementar também um tradutor para atuar sobre uma sintaxe mais sofisticada, tornando os códigos mais estruturados e claros. A implementação do tradutor será descrita no próximo capítulo.

6 TRADUTOR PARA UMA SINTAXE DE ALTO NÍVEL PARA A BIBLIOTECA QJAVA

O presente capítulo objetiva apresentar um tradutor para uma sintaxe em alto nível, utilizada para descrever setas quânticas de maneira clara e estruturada para um código executável em Java, sendo ele composto por setas, closures e demais chamadas à biblioteca da computação quântica apresentada neste trabalho, QJava.

A implementação do referido tradutor, teve como base o trabalho (BANDEIRA; BOIS; PILLA, 2011), onde os autores descrevem um tradutor para uma sintaxe similar a notação-do para operar sobre mônadas representando memórias transacionais. Para tanto, é utilizada uma ferramenta geradora de *analisadores sintáticos*, ANTLR.

Este capítulo é organizado da seguinte forma: em um primeiro momento, será exposta uma breve descrição acerca da transformação de programas, bem como a funcionalidade da ferramenta ANTRL. Posteriormente, serão definidas regras para a gramática do tradutor e classes auxiliares. Ao final, serão demostrados exemplos de algoritmos quânticos utilizando a sintaxe.

6.1 Geração e Transformação de Programas

A transformação de programas é o ato de transformar um programa em outro (VISSER, 2005). A forma mais comum de transformação de programas é a compilação de um código fonte para um arquivo executável, tendo como resultado, normalmente, um programa descrito em linguagem de máquina. Mas existem também transformações que podem ser executadas sobre programas e não necessariamente terem como resultado um código executável. É possível haver transformações de otimização, refatoração ou até mesmo geração de documentação.

As transformações de programas podem ser divididas em duas categorias, sendo a primeira com entrada e saída de mesma linguagem (*reexpressão*), e a segunda com linguagem de saída diferente à de entrada (*tradução*) .

De modo geral, a reexpressão busca o mesmo sentido, mas com descrições diferentes, o que acarreta na mudança semântica do programa. Esse tipo de transformação é utilizado para realizar refatoração, otimização, normalização e renovação de programas (VISSER, 2005).

No caso da tradução, deve-se transformar uma linguagem fonte em uma linguagem alvo. O alvo pode ser, por exemplo, uma linguagem de máquina para transformar um código fonte em um arquivo executável, ou também em uma outra linguagem de programação. O tradutor é,

portanto, um programa que faz leitura de uma entrada (*input*) e gera uma saída (*output*) (PARR, 2007). Assim, um tradutor recebe como entrada um código fonte cuja sequência de instruções é transformada em outra, resultando em uma saída traduzida com a sintaxe da linguagem alvo.

Essa tarefa é realizada através da decomposição de sentenças completas em subsentenças ou (sub-)expressões, mais fáceis de se gerenciar. Por exemplo, a definição de uma classe em Java é escrita como `class nome{corpo}`, onde o corpo pode conter tanto definições de funções quanto variáveis, ou seja, pode ser quebrado em um sub-sentença descrevendo o que fazer se for uma função e outra sub-sentença descrevendo o que fazer se for uma variável. Para descrever todas as sentenças e sub-sentenças possíveis, um tradutor conta com auxílio de uma *gramática*, que é uma maneira formal de descrever as sentenças e as regras de tradução. Através da gramática, o tradutor consegue mapear cada sequência de entrada para uma saída específica.

6.1.1 Compiladores

O tradutor mais comum é o compilador. Em essência, o compilador traduz um programa escrito em uma linguagem fonte para um programa semanticamente equivalente em outra linguagem (AHO et al., 2006). Além disso, através dele, é possível reportar qualquer erro detectado durante a tradução. O procedimento de compilação de um programa é seguido por uma série de fases, sendo estas divididas em duas grandes etapas (análise e síntese), conforme se demostra na Figura 6.1.

O primeiro passo da fase de análise é fazer a análise léxica (ou *scanning*) do programa fonte, onde é feita a leitura de todos os caracteres, verificando se pertencem ao alfabeto da linguagem, agrupando-os logicamente em *tokens* (identificadores, *keywords* e outras palavras reservadas). O analisador léxico inicia a construção da Tabela de Símbolos e envia mensagens de erros caso identifique unidades léxicas não aceitas pela linguagem em questão. O segundo passo se dá pela análise sintática (ou *parsing*), momento em que é verificado se a estrutura gramatical do programa está de acordo com as regras gramaticais da linguagem. O analisador léxico, ou somente *parser*, produz uma representação intermediária, chamada árvore de derivação, cuja finalidade é descrever a estrutura gramatical da sequência de entrada. O próximo passo é o instante em que a análise semântica utiliza a árvore sintática e a informação na tabela de símbolos para verificar a consistência semântica do programa com a definição da linguagem (PRICE; TOSCANI, 2008).

A fase de síntese constrói o programa na linguagem alvo a partir da representação in-

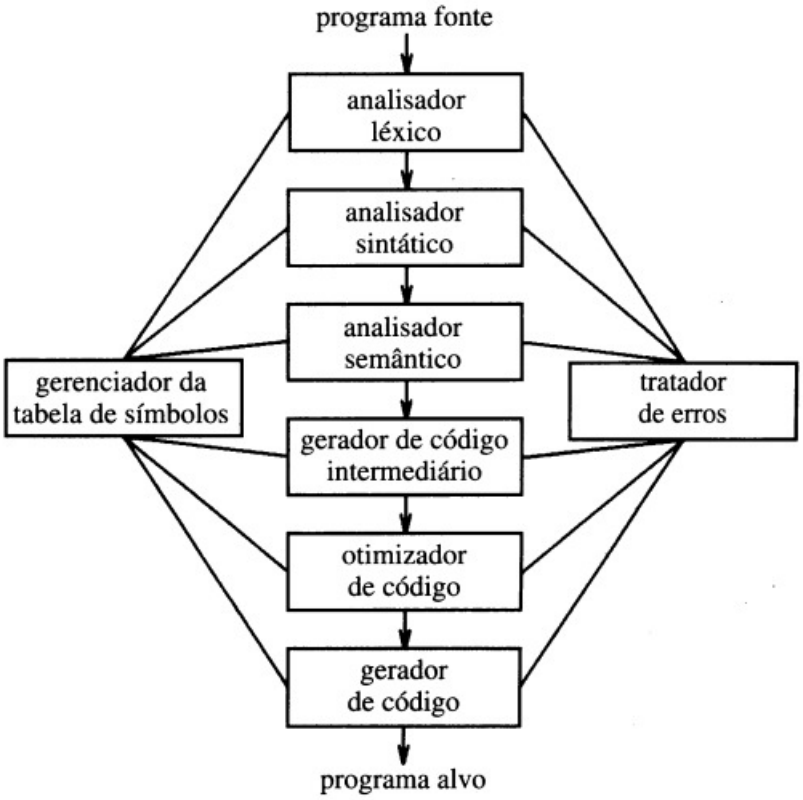


Figura 6.1 – Fases de um compilador (AHO et al., 2006)

termediária e das informações da Tabela de Símbolos fornecidas pela fase de análise. Consiste na geração de um código intermediário, que deve ser fácil de produzir e traduzir para a linguagem de máquina (AHO et al., 2006), seguida de uma otimização, buscando aplicar melhorias no código. E por fim, na geração do código final através do mapeamento da representação intermediária para a linguagem alvo em si.

O processo de transformar o código fonte de uma linguagem fonte em uma linguagem alvo é complexo e requer uma série de procedimentos. Felizmente, existe uma série de ferramentas que auxiliam no processo de construção de compiladores, tais como geradores de *parsers* e *scanners*.

6.2 Ferramenta ANTLR

ANTLR[11] (ANother Tool for Language Recognition) é um poderoso gerador de parser, utilizado para ler, processar, executar ou traduzir códigos estruturados ou arquivos binários. É

[11] Disponível em `http://www.antlr.org/`

uma ferramenta que pode ser usada para implementar interpretadores, compiladores e outros tradutores.

A referida ferramenta caracteriza-se por determinar se uma sequência de entrada está de acordo com a definição formal da linguagem, a gramática. Neste passo, uma gramática é definida como um conjunto de regras que descrevem subsentenças da linguagem, isto é, cada regra consiste em uma ou mais alternativas (PARR, 2007). Uma das vantagens do ANTRL é que, através da adição de alguns fragmentos de código à gramática, o reconhecedor torna-se um tradutor. Esses fragmentos de código são capazes de transformar uma expressão de entrada em uma expressão diferente de saída.

Uma gramática para a linguagem Java em ANTLR[12], por exemplo, é inicialmente definida como:

```
compilationUnit
    :    (    (annotations
             )?
             packageDeclaration
         )?
         (importDeclaration
         )*
         (typeDeclaration
         )*
    ;
```

A regra em que o *parsing* começa é chamada de regra inicial, o que, neste caso, define-se como *compilationUnit*. Um programa em Java é descrito como uma unidade formada por *annotations* (métodos sintáticos adicionados ao código), *packageDeclaration* (declaração de pacotes), *importDeclaration* (declarações de importações) e e *typeDeclaration* (declarações de tipos, por exemplo, a definição de uma classe). O símbolo *"?"* indica que a regra é opcional, e o "*", que pode haver nenhuma ou mais de uma ocorrência da regra. Um código fonte que não atender tais regras, ou as demais regras definidas na gramática, irá informar uma mensagem de erro.

O ANTLR suporta diversas linguagens alvo como Java, Phython, Ruby, C# entre outras. Ele automatiza ou ao menos formaliza muitas tarefas comuns. Suas gramáticas não são complexas, mas claras e eficientes na geração de algoritmos reconhecedores. A vantagem em trabalhar com a ferramenta ANTLR está na desnecessidade definir manualmente um *parser* e um *lexer* para uma gramáticaespecífica, o procedimento é feito automaticamente neste caso.

[12] Disponível em `http://www.antlr3.org/grammar/list.html`

6.2.1 String Template

O *String Template* é um motor de modelos (*templates*) em Java (com versões para C# e Python), com a característica de gerar código-fonte, páginas web, e-mails, ou qualquer outra saída de texto formatado. O motor de *templates*, é simplesmente um gerador de código que segue um modelo definido como um documento com lacunas em branco, podendo ser preenchidas com atributos, isto é, valores que são passados como parâmetros para o *template* (PARR, 2007).

Se o tradutor gera código fonte ou outro texto de saída, diretamente através de instruções de impressão (*prints*), ele não segue uma lógica bem estruturada, o que dificulta sua codificação, leitura e, posteriormente, modificação. Isso pode ser solucionado com a utilização de *templates*, pois se define a estrutura de saída separadamente, facilitando sua definição e edição. Outro benefício de utilizar o *String Template* em conjunto com o ANTLR, é que as *templates* são carregadas em tempo de execução, sendo possível alterá-las sem a necessidade de modificar sua gramática.

```
cria_classe(nome, pai, corpo) ::= <<
public class <nome> <if (pai)> extends <pai> <endif> {
   <corpo>
}
>>
```

Figura 6.2 – Template para a definição de uma classe em Java

Uma *template* simples, definida para gerar o código de criação de uma classe em Java, é descrita na Figura 6.2. Ela recebe três atributos: o nome da classe, uma super classe e seu corpo. A definição do corpo de uma *template* é delimitada por « e ». O texto de saída preenche as lacunas em branco, delimitadas por < e > e pelo referido valor recebido como parâmetro. No caso da expressão de condição, será preenchido o texto somente se for recebido um parâmetro para o valor referente à super classe. Essa *template* irá retornar um texto exatamente igual ao especificado na definição e com os devidos campos já preenchidos.

O uso de *StringTemplate* em conjunto com a ferramenta ANTLR forma um eficiente framework para a construção de tradutores. Na próxima seção, serão descritas as regras da gramática, bem como as *templates* criadas para a tradução de uma sintaxe de alto nível da representação de setas quânticas, para uma versão naturalmente compilável e executável em Java.

6.3 Tradutor para a biblioteca QJava

Como visto anteriormente, ao final da Seção 5.2, a forma de elaborar uma sequência de operações quânticas sobre os qubits utiliza as funções das setas quânticas para a composição dos closures. Tal tarefa utiliza uma sintaxe complexa e exige do programador saber, explicitamente, manipular o estado quântico e permutar seus valores. Sendo assim, no intuito de facilitar o procedimento de desenvolvimento de algoritmo quânticos, é definida uma sintaxe mais simples de se usar, similar a extensão da notação-*do* apresentada em por Paterson (2001), para ser utilizada em conjunto com um tradutor que automatiza a geração do código das setas.

A sintaxe proposta pelo presente trabalho é uma extensão para a linguagem Java e permite códigos simples e fáceis de estruturar. Por exemplo, a função `had1_2` apresentada na Seção 5.2.4, pode ser reescrita da seguinte forma:

```
Super hads = Super.lin2super(Lin.qhadamard());

Super had1_2 = new proc (a,b) (bool,bool) {
   (b2) <- hadS -< (b)
   returnA <- (a,b2)
}
```

A nova sintaxe para construir setas é descrita como uma sequência de instruções dentro de um bloco `proc`. O primeiro grupo de parênteses atribui nomes as bases que irão representar os qubits, e o segundo, os seus respectivo tipos, ou seja, o resultado do trecho de código acima é um super-operador para ser aplicado em um estado quântico de dois qubits cujas bases é representada pela tupla (boolean,boolean).

O símbolo $-<$ representa uma *aplicação* do valor de uma expressão em uma seta, o símbolo $\leftarrow$ indica a composição de duas setas através da operação bind e, por fim, a instrução `returnA` indica uma construção equivalente ao método *return* das mônadas.

O trecho de código descrito acima, representa a aplicação de uma operação *Hadamard* no segundo qubit de um estado quântico de dois qubits. A primeira instrução do bloco `proc` mapeia a entrada `b` para o primeiro componente da seta e aplica uma operação `hadS`, através da função `first`, alterando apenas o valor do qubit representado pela entrada `b`, que agora passa a se chamar de `b2`. Outras operações quânticas podem ser escritas na sequência, seguindo a mesma lógica, mas a ultima instrução deverá sempre ser a construção final da seta com a primitiva `returnA`, que nesse caso específico, é construir uma seta onde a entrada `a` é o primeiro elemento e `b2` o segundo.

O algoritmo de Toffoli, descrito na Seção 2.4, pode ser implementado através do código descrito como:

```
public static Super Toffoli (){
   Boolean bool = true;
   Super hadS = Super.lin2super(Lin.qhadamard());
   Super cnotS = Super.lin2super(Lin.controlled(Lin.qnot()));
   Super cphaseS = Super.lin2super(Lin.controlled(Lin.qphase()));
   Super caphaseS = Super.lin2super(Lin.controlled(Lin.adjoint(Lin.qphase())));

   return new proc (a,b,c) (bool,bool,bool){
      (c1) <- hads -< (c);
      (b1,c2) <- cphaseS -< (b0,c1);
      (a1,b2) <- cnotS -< (a0,b1);
      (b3,c3) <- caphaseS -< (b2,c2);
      (a2,b4) <- cnotS -< (a1,b3);
      (a3,c4) <- cphaseS -< (a2,c3);
      (c5) <- hadS -< (c4);
      returnA -< (a3,b4,c5);
   }
}
```

Para facilitar a legibilidade do referido código, são criadas variáveis auxiliares que representam o super operador de cada porta quântica usada no algoritmo, o qual opera sobre um estado de três qubits.

A sequência de instruções que foi elaborada segue uma maneira estruturada de descrever um algoritmo, esclarecendo o que acontece passo-a-passo. Assim, ao aplicar uma operação `hadS` em um valor `c`, significa aplicar a porta quântica *Hadamard* no terceiro qubit do sistema. E na sequência, aplicar uma operação controlada `cnotS` que recebe `(a0,b1)`, isto é, será aplicada a porta quântica controlada do *NOT* no primeiro (`a`) e segundo (`b`) qubit do sistema.

Ao final, o algoritmo montará uma seta com os valores computados durante todo o percurso do algoritmo. A operação descrita pela função acima, poderá ser combinada em um estado de três qubits usando a função *bind* ou até mesmo para compor outros algoritmos.

Para ilustrar o funcionamento de *medidas* e *tracing*, considera-se como exemplo o Algoritmo da Teleportação que é implementado como:

```
public static Super Alice (){
   return new proc (eprL,q) (true,true){
      (q1,e1) <- cnotS -< (q,eprL);
      (q2) <- hadS -< (q1);
      ((q3,e2),(m1,m2)) <- meas -< (q2,e1);
      (m3 ,m4) <- trL -< ((q3,e2),(m1,m2));
      returnA -< (m3,m4);
   }
}

public static Super Bob (){
```

```
    return new proc (eprR,m1,m2) (true,true,true){
        (m4,e1) <- cnotS -< (m2,eprR);
        (m3,e2) <- zS -< (m1,e1);
        (q) <- trL -< ((m3,m4),e2);
        returnA -< (q);
    }
}

public static Super Teleport (){
    return new proc (eprL,eprR, q) (true,true,true){
        (m1,m2) <- alice -< (eprL,q);
        (q) <- bob -< (eprR,m1,m2);
        returnA -< (q);
    }
}
```

O algoritmo é quebrado em três setas diferentes por questões de legibilidade. A função Alice é receber uma parte do estado EPR, o qual é compartilhado com Bob, e executar uma medida. Bob recebe o resultado da medida e aplica uma operação de acordo com o resultado da medida. Por fim, a seta *Teleport* apenas coordena a comunicação entre Alice e Bob, e retorna o qubit que foi teleportado de Alice para Bob.

As operações quânticas usadas no algoritmo são descritas pelas variáveis *cnotS*, para o *not* controlado, *hadS*, para a porta quântica *hadamard*, *meas* e *trL*, para a operação de *medida* e *tracing* sobre um específico estado quântico, e *zS*, para a transformação final do qubit a ser teleportado que é uma troca de fase.

A descrição em detalhes de como foi desenvolvido o tradutor, bem como as regras da gramática, são apresentadas no restante deste capítulo.

6.3.1 Gramática

A gramática criada para o tradutor foi construída como uma extensão daquela disponibilizada pelo ANTLR para a linguagem Java. Foram adicionadas regras sintáticas e léxicas para o correto funcionamento da abstração de setas quânticas. A partir dessa gramática o ANTLR gera um *parser* e um *lexer*, os quais reconhecem as novas expressões da linguagem e verificam a sua integridade emitindo mensagens de erros para expressões inválidas. O resultado é um código descrito na linguagem Java semanticamente equivalente à abstração das setas quânticas.

As novas regras da gramática são incorporadas junto com a primitiva *new*. Naturalmente em Java, uma instrução após um operador *new* pode ser a criação de uma instância de classe ou um *array*. Contudo, após o procedimento acima descrito, também será possível, através dela, construir uma *seta*. A chamada e a definição da regra inicial para a abstração de setas é definida

como:

```
creator
    :   'new' arrowProc -> template(id={$arrowProc.sttp}) "<id>"
    |   'new' classOrInterfaceType classCreatorRest
    |   arrayCreator
    ;

arrowProc returns [StringTemplate sttp]
    :   'proc'
            atributos
            tipos
        '{'
            bloco[$atributos.lista, $tipos.lista]
        '}'
        {$sttp = $ bloco.stmp;}
;
```

A primitiva `template` acessa a StringTemplate `sttp` a qual retornou das regras de derivação de `arrowproc` e e executa a tradução em si, pois o código anterior é substituído pelo recebido pelo identificador `id`. A tradução do bloco `proc` é feita recursivamente dentro do `bloco`, as regras pertencentes aos nós mais inferiores da árvore de derivação retornam sua tradução para os nós superiores, construindo assim, a tradução de acordo com as regras definidas.

A regra `atributos` e `tipos` computa os parâmetros, transformando-os em uma lista que será recebida por uma classe `Contexto`, cuja função é manipular todas as variáveis envolvidas dentro do processo de tradução.

As regras para o `bloco` e `arrowstat` são definidas como:

```
bloco [String lista, String tipos ] returns [StringTemplate stmp]
    scope {
        Context ctx;
        boolean isfinal;
    }
    @init {
        $bloco::isfinal = false;
        StringTemplate head = null, tail = null;
        $bloco::ctx = new Context(lista,tipos);
    }
    :
        (
            arrowStat [$bloco::ctx]
            {
                head = %block(begin={head}, end={$arrowStat.tp1});
                tail = %block(begin={$arrowStat.tp2}, end={tail});
            }
        )+
        {$stmp=%block(begin={head}, end={tail});}
    ;
```

```
arrowStat [Context ctx] returns [StringTemplate tp1=null,
                                 StringTemplate tp2=null]
    :
        arrowBind[$ctx]
        {$tp1 = $arrowBind.stpl; $tp2 = $arrowBind.tail;}
    |
        arrowReturn[$ctx]
        {$tp1 = $arrowReturn.stpl; $tp2 =
           $arrowReturn.tail;System.out.println(ctx);}
```

A regra `bloco` é responsável pela construção e concatenação das sentenças, que podem ser aplicações ou atribuições dentro de uma seta. Um bloco pode possuir uma ou mais sentenças, representado pelo símbolo +. Um `arrowstat` pode ser uma operação *bind* ou *return*.

Uma atribuição (*bind*) jamais poderá terminar um bloco e, portanto, nunca será última expressão descrita. Para melhor entendimento, utiliza-se como exemplo, neste caso, uma operação *bind* computa uma operação *first* em um seta. A seta pode ser uma simples função de construção de setas *arr* ou também uma outra sequência de computações (um outro bind). Assim, deve-se utilizar uma StringTemplate *head* para guardar a o começo das sentenças e *tail*, para que, posteriormente, ela possa ser fechada.

Como são criadas novas variáveis e tuplas de acordo com a sequência de operações descritas dentro de um bloco, foi necessário elaborar uma classe gerenciadora de contexto. Através dela, para cada operação das setas, é gerada uma StringTemplate da operação que recebe como parâmetros as informações contidas dentro da classe Contexto. Por exemplo, a instrução `returnA` deve retornar uma StringTemplate representando uma operação `arr`. A StringTemplate para essa operação é demostrada na Figura 6.3.

```
arrowReturn [Context ctx] returns [StringTemplate stpl, StringTemplate tail]
    @init
    {
        String lista = "";
    }
    :
        'returnA' atributos (';')
        {
             $stpl = %arr(
              type={ctx.getType(ctx.getContext())},
              constructor={ctx.getConstructor()},
              name={ctx.getNames()},
              retorno={ctx.getReturnOf2($atributos.lista)}
             );
        }
;
```

A ordem em que as alternativas das regras são declaradas é respeitada pelo parser. Isto é, o parser tentará derivar um `arrowstat` em um `arrowBind`, caso não consiga corresponder o

```
arr(type,constructor,name, retorno) ::= <<
QArrows.arr(<type>, (<constructor> <name>) -> {
    return <retorno>;
})
>>
```

Figura 6.3 – String Template para a criação de um método *arr*

fluxo de *tokens* da entrada com a definição da regra, buscará na próxima, e assim sucessivamente até a última. Dessa forma, as regras devem ser definidas em ordem crescente de abstração, o que ocorre com a regra `arrowBind`, pois além de operar sobre uma operação quântica normal, são definidas regras mais especificas para o caso da medida e tracing:

```
arrowBind [Context ctx] returns [StringTemplate stpl, StringTemplate tail]
    :
            atributos  '<-' 'meas' arrowFirst[$ctx]  (';')
        {
            ctx.renameMeas($arrowFirst.bases);
            StringTemplate x1 = $arrowFirst.stpl;
            StringTemplate x2 = %first(type={ctx.getType(ctx.getContext())},
             app={"meas"});
            x2 = %bind(x1={x2});
            $stpl = %bind_block(x1={x1},x2={x2});
            $tail = new StringTemplate("))");
        }
      |
        a1=atributos  '<-' 'trL' '-<' a2=atributos (';')
        {
            ctx.renameTracing($a1.lista,$a2.lista);
            StringTemplate x1 = %first(type={ctx.getType($a1.lista)}, app={"trL"});
            $stpl = %bind(x1={x1});
            $tail = new StringTemplate(")");

        }
      |
        atributos  '<-'  app1=IDENTIFIER  arrowFirst[$ctx]  (';')
        {
            ctx.rename($atributos.lista,$arrowFirst.bases);
            StringTemplate x1 = $arrowFirst.stpl;
            StringTemplate x2 = %first(type={ctx.getType($atributos.lista)},
             app={$app1.text});
            x2 = %bind(x1={x2});
            $stpl = %bind_block(x1={x1},x2={x2});
            $tail = new StringTemplate("))");
        }
;
```

Tal regra, em conjunto com a `arrowfirst` envolve todo o processo de tradução de atribuição de uma expressão a uma seta quântica.

```
arrowFirst [Context ctx] returns [StringTemplate stpl, StringTemplate tail,
 String bases]
```

```
    :
        '-<' atributos
        {
            $stpl = %arr(
             type={ctx.getType(ctx.getContext())},
             constructor={ctx.getConstructor()},
             name={ctx.getNames()},
             retorno={ctx.getReturnOf($atributos.lista)}
            );
            $stpl = %bind(x1={$stpl});
            $bases = $atributos.lista;
            $tail = new StringTemplate(")");
        }
;
```

Todas as regras da gramática envolvida na tradução retornam duas *StringTemplate*, as quais serão processadas pela regra bloco para a construção do corpo de *proc*. Além disso, a classe contexto é constada como atributo da regra bloco, entretanto, sofre mudança com a computação das regras de cada sentença. Essa classe oferece funções para a adição e remoção de variáveis, com regras específicas para o caso de operações de medida e tracing. Ademais, tem uso direto na construção das *templates*, tendo como principais métodos uma função *getType*, que retorna se o contexto é uma tupla ou outro tipo, *getConstructor* e *getNames*, que retorna respectivamente o construtor e uma nomenclatura para as variáveis dentro do contexto, e *getReturnOf*, que recebe um lista de variáveis como argumento e retorna como são acessadas dentro do contexto.

7 CONCLUSÃO

Um dos desafios da área da computação quântica é desenvolver linguagens de programação segundo as regras da mecânica quântica para suportar a criação, análise, modelagem e simulação de algoritmos quânticos de *alto nível*. Nesse contexto, o presente trabalho introduziu um modelo semântico, que usa mônadas e setas, capaz de expressar todos os tipos de computações quânticas e medidas.

A união entre computação quântica e setas, está presente neste estudo através de uma biblioteca chamada QJava. Implementada com a versão de nº 8 da prévia de desenvolvedores da JDK (Java Development Kit), essa biblioteca consegue elaborar algoritmo quânticos de qualquer tipo apesar de suas restrições, como na questão de desempenho. Contudo, a sintaxe utilizada pela biblioteca demostrou ser extensa, tornando confusa sua utilização devido a falta de clareza inerente ao código das setas.

Sendo assim, foi apresentada a estrutura de uma nova sintaxe para lidar com setas de uma maneira clara e estruturada, bem como um respectivo tradutor. Tal procedimento resultou em um código mais modular, facilitando a leitura e entendimento dos programas, assim como a manutenção e a reutilização do código.

O tradutor foi implementado através da ferramenta geradora de *parsers* ANTLR. A gramática criada bem como as *templates* usadas para a tradução do código, podem ser vistas como uma extensão da linguagem Java. Com a ajuda do tradutor, foi possível implementar algoritmos quânticos de uma maneira menos complexa, deixando a cargo do tradutor o papel de montar o código extenso necessário para a utilização da biblioteca QJava.

Espera-se, que o produto desenvolvido sirva de auxílio na implementação de outros algoritmos e operações da computação quântica, e também, que ofereça aos pesquisadores interessados uma base de estudo da computação quântica.

7.1 Trabalhos Futuros

A biblioteca QJava em conjunto com o tradutor é, de certa forma, uma linguagem de domínio específico (*DSL*). Assim, pode ser indicado como um trabalho futuro a geração de um sistema de tipos e um compilador para uma linguagem específica para a computação quântica criada sobre uma plataforma Java.

Ademais, estudos devem ser feitos acerca dos problemas de desempenho relacionados

com a composição de setas. Uma das possíveis soluções pode ser a de modificar a representação da computação quântica para não usar closures na representação de vetores e matrizes de densidade. Esta modelagem pode ser modificada para uma versão baseada em listas, ou a coleção *Map*, sem modificar a estrutura de uma mônada. Esta alternativa pode vir a trazer melhorias de desempenho no processamento do cálculo das seta, pois parte da ideia de que talvez seja mais rápido processar listas.

REFERÊNCIAS

AHARONOV, D.; KITAEV, A.; NISAN, N. Quantum circuits with mixed states. In: PROC. 30TH STOC. **Anais...** [S.l.: s.n.], 1998.

AHO, A. V. et al. **Compilers**: principles, techniques, and tools (2nd edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

BANDEIRA, R. L.; BOIS, A. R. du; PILLA, M. L. **Compilador para a linguagem CMTJava**. 2011.

BARR, M.; WELLS, C. **Toposes, Triples and Theories.** [S.l.]: Springer Verlag, 1895.

BENNETT, C. et al. **Teleporting an unknown quantum state via dual classical and EPR channels**. Bristol, UK, UK: [s.n.], 1993.

BENTON, N.; HUGHES, J.; MOGGI, E. Monads and Effects. In: IN INTERNATIONAL SUMMER SCHOOL ON APPLIED SEMANTICS APPSEM'2000. **Anais...** Springer-Verlag, 2000. p.42–122.

CABRAL, G. E.; LULA, B. J.; LIMA, F. A. **Uma Ferramenta para Projeto e Simulação de Circuitos Quânticos**. 2004.

CHURCH, A. A Formulation of the Simple Theory of Types. **Journal of Symbolic Logic**, [S.l.], v.5, n.2, p.56–68, 1940.

DEUTSCH, D. Quantum Theory, the Church-Turing Principle, and the Universal Quantum Computer. In: ROYAL SOCIETY OF LONDON. **Proceedings...** [S.l.: s.n.], 1985. v.47, p.97–117.

EINSTEIN, A.; PODOLSKY, B.; ROSEN, N. Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? **Phys. Rev.**, [S.l.], v.47, n.10, p.777–780, May 1935.

FELLEISEN, M. et al. The Revised Report on the Syntactic Theories of Sequential Control and State. **Theoretical Computer Science**, [S.l.], v.103, p.235–271, 1992.

FEYNMAN, R.; SHOR, P. W. Simulating Physics with Computers. **SIAM Journal on Computing**, [S.l.], v.26, p.1484–1509, 1982.

GROVER, L. K. A Fast Quantum Mechanical Algorithm for Database Search. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING. **Anais...** ACM, 1996. p.212–219.

HARRIS, T. et al. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, New York, NY, USA. **Proceedings...** ACM, 2005. p.48–60. (PPoPP '05).

HUDAK, P.; PEYTON JONES, S.; WADLER, P. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). **ACM SIGPLAN Notices**, [S.l.], v.27, n.5, May 1992.

HUGHES, J. Generalising monads to arrows. **Sci. Comput. Program.**, Amsterdam, The Netherlands, The Netherlands, v.37, n.1-3, p.67–111, May 2000.

KISSINGER, A.; MERRY, A.; SOLOVIEV, M. **Pattern Graph Rewrite Systems**. 2012.

LANDIM, P. J. The mechanical evaluation of expressions. **The Computer Journal (British Computer Society)**, [S.l.], 1964.

LANDIN, J. P. The mechanical evaluation of expressions. In: THE COMPUTER JOURNAL. **Anais...** [S.l.: s.n.], 1964.

LIANG, S.; HUDAK, P.; JONES, M. Monad Transformers and Modular Interpreters. In: IN PROCEEDINGS OF THE 22ND ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES. ACMPRESS. **Anais...** [S.l.: s.n.], 1995.

MERMIM, N. D. **Quantum Computer Science**. [S.l.]: Cambridge University Press, 2007.

MOGGI, E. Computational lambda-calculus and monads. In: FOURTH ANNUAL SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE. **Proceedings...** IEEE Press, 1989. p.14–23.

MOGGI, E. Computational lambda-calculus and monads. In: FOURTH ANNUAL SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE. **Proceedings...** IEEE Press, 1989a. p.14–23.

MOGGI, E. **An Abstract View of Programming Languages**. [S.l.]: Edinburgh University, 1989b.

MOGGI, E. **Notions of Computation and Monads**. 1991.

MU, S.-C.; BIRD, R. Functional Quantum Programming. In: IN PROCEEDINGS OF THE 2ND ASIAN WORKSHOP ON PROGRAMMING LANGUAGES AND SYSTEMS. **Anais...** [S.l.: s.n.], 2001.

NAFTALIN, M. **Maurice Naftalin's Lambda FAQ**. 2013.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information**. [S.l.]: Cambridge University Press, 2000.

ORACLE. **Project Lambda**. 2013a.

ORACLE. **JDK 8**. 2013b.

PALAO, J. P.; KOSLOFF, R. Quantum Computing by an Optimal Control Algorithm for Unitary Transformations. In: PHYSICAL REVIEW LETTERS. **Anais...** [S.l.: s.n.], 2002. v.89.

PARR, T. **The Definitive ANTLR Reference**: building domain-specific languages. [S.l.]: Pragmatic Bookshelf, 2007.

PATERSON, R. A new notation for arrows. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, New York, NY, USA. **Proceedings...** ACM, 2001. p.229–240. (ICFP '01).

PETERSON, J.; HAGER, G. Monadic Robotics. In: PROCEEDINGS OF THE WORKSHOP ON DOMAIN SPECIFIC LANGUAGES, USENIX. **Anais...** [S.l.: s.n.], 1999.

PIERCE, B. C. **Types and Programming Languages**. Cambridge, Massachusetts: MIT Press, 2002. 645p.

POMERANCE, C. A Tale of Two Sieves. **Notices Amer. Math. Soc.**, [S.l.], v.43, p.1473–1485, 1996.

PRICE, A. M. A.; TOSCANI, S. **Implementação de Linguagens de Programação**: compiladores. [S.l.]: Bookman, 2008.

RIVEST, R. L.; SHAMIR, A.; ADELMAN, L. M. **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**. [S.l.]: MIT, 1977. (MIT/LCS/TM-82).

SABRY, A. What is a Purely Functional Language? **Journal of Functional Programming**, [S.l.], v.8, p.1–22, 1998.

SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IN PROC. IEEE SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE. **Anais...** [S.l.: s.n.], 1994. p.124–134.

SWIERSTRA, S. D.; DUPONCHEEL, L. Deterministic, Error-Correcting Combinator Parsers. In: ADVANCED FUNCTIONAL PROGRAMMING, SECOND INTERNATIONAL SCHOOL-TUTORIAL TEXT, London, UK, UK. **Anais...** Springer-Verlag, 1996. p.184–207.

VISSER, E. A survey of strategies in rule-based program transformation systems. **J. Symb. Comput.**, Duluth, MN, USA, v.40, n.1, p.831–873, July 2005.

VIZZOTTO, J. K.; ALTENKIRCH, T.; SABRY, A. Structuring Quantum Effects: superoperators as arrows. **Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages**, [S.l.], v.16, p.453–468, 2006.

VIZZOTTO, J. K.; ROCHA COSTA, A. C. da; SABRY, A. Quantum Arrows in Haskell. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.210, p.139–152, July 2008.

VRIES, A. de. **jQuantum - Quantum Computer Simulator**. 2010.

WADLER, P. The essence of functional programming. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 19., New York, NY, USA. **Proceedings...** ACM, 1992. p.1–14. (POPL '92).

WADLER, P. Monads for Functional Programming. In: ADVANCED FUNCTIONAL PROGRAMMING, FIRST INTERNATIONAL SPRING SCHOOL ON ADVANCED FUNCTIONAL PROGRAMMING TECHNIQUES-TUTORIAL TEXT, London, UK, UK. **Anais...** Springer-Verlag, 1995. p.24–52.

YANOFSKY, N. S.; MANNUCCI, M. A. **Quantum Computing for Computer Scientists**. 1.ed. New York, NY, USA: Cambridge University Press, 2008.