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

UMA LINGUAGEM DE PROGRAMAÇÃO QUÂNTICA ORIENTADA A OBJETOS BASEADA NO FEATHERWEIGHT JAVA

DISSERTAÇÃO DE MESTRADO

Samuel da Silva Feitosa

Santa Maria, RS, Brasil

2016

UMA LINGUAGEM DE PROGRAMAÇÃO QUÂNTICA ORIENTADA A OBJETOS BASEADA NO FEATHERWEIGHT JAVA

Samuel da Silva Feitosa

Dissertação apresentada ao Curso de Mestrado do 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 Kaizer Vizzotto

Santa Maria, RS, Brasil

2016

Feitosa, Samuel da Silva

Uma Linguagem de Programação Quântica Orientada a Objetos Baseada no Featherweight Java / Samuel da Silva Feitosa. – 2016.
99 p.; 30 cm.

Orientadora: Juliana Kaizer Vizzotto
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2016.

1. Computação Quântica. 2. Mônada Quântica. 3. Featherweight Java. I. Vizzotto, Juliana Kaizer. II. Título.

E-mail: sfeitosa@inf.ufsm.br

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

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

UMA LINGUAGEM DE PROGRAMAÇÃO QUÂNTICA ORIENTADA A OBJETOS BASEADA NO FEATHERWEIGHT JAVA

elaborada por
Samuel da Silva Feitosa

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

COMISSÃO EXAMINADORA:

Juliana Kaizer Vizzotto, Dra.
(Presidente/Orientadora)

Eduardo Kessler Piveta, Dr. (UFSM)

Andre Rauber Du Bois, Dr. (UFPEL)

Santa Maria, 04 de Março de 2016.

À minha família, por sua capacidade de acreditar e investir em mim. Mãe, seu incentivo foi o que me deu força para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nesta caminhada.

AGRADECIMENTOS

Meu agradecimento especial a todas as pessoas que incentivaram e contribuíram direta ou indiretamente para o desenvolvimento deste trabalho.

À minha orientadora, professora Juliana Kaizer Vizzoto, pelas oportunidades proporcionadas, apoio e confiança durante esta jornada. Nossas conversas e discussões transformaram a minha visão da vida em diversos aspectos. Seu conhecimento e brilhantismo foram essenciais na condução deste projeto.

Agradecimento especial ao professor Eduardo Kessler Piveta, por compartilhar seu tempo e seu conhecimento, pelo incentivo, pelas oportunidades e pela parceria. Ao professor André Rauber Du Bois por contribuir com ideias e sugestões sobre o trabalho.

Aos meus colegas que estiveram presentes em algum momento do curso, muito obrigado pelas discussões e conversas que só me fizeram crescer. Um agradecimento especial aos meus amigos Fábiner Fugali, Rafael Rohden, Thiago Krug e Tiago Idalêncio pelas horas de diversão e de discussões sobre computação e tecnologia.

Aos meus pais, pelo incentivo e por proporcionarem a base para minha educação.

Finalmente, um agradecimento especial à minha esposa Cheila, pela sua compreensão, dedicação, apoio e carinho durante todo este tempo. Você é, com certeza, um dos motivos de estar onde estou hoje. Muito obrigado.

“If you think you understand quantum mechanics,
you don’t understand quantum mechanics”
— RICHARD FEYNMAN

RESUMO

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

UMA LINGUAGEM DE PROGRAMAÇÃO QUÂNTICA ORIENTADA A OBJETOS BASEADA NO FEATHERWEIGHT JAVA

AUTOR: SAMUEL DA SILVA FEITOSA
ORIENTADORA: JULIANA KAIZER VIZZOTTO
Local da Defesa e Data: Santa Maria, 04 de Março de 2016.

Com a aproximação do fim da Lei de Moore, onde não será possível melhorar a capacidade dos processadores baseados em silício, a computação quântica aparece como uma boa escolha para prover uma nova era da computação. A computação quântica pode ser entendida como a arte de transformar informação codificada no estado físico *quântico*. Esta codificação se dá através de bits quânticos (*qubits*), que podem estar em estados de *superposição* ou *emaranhados*, permitindo explorar uma propriedade conhecida como *paralelismo quântico*. Nesta dissertação é discutida a criação de uma linguagem de programação quântica que utiliza-se do paradigma da orientação a objetos (*OO*), fornecendo a possibilidade de manipular classes e objetos, onde os dados e os efeitos quânticos são manipulados através de uma abordagem monádica, sendo modelada como uma extensão da proposta Featherweight Java (*FJ*). Esta extensão é definida formalmente através da apresentação de sua *semântica operacional*, a qual é passível de implementação em qualquer linguagem de programação que forneça o mecanismo de *closures*. A formalização desta linguagem permitiu a criação de um interpretador, que implementa as fases de análise léxica, sintática e semântica, com foco especial no tratamento do sistema de tipos para embutir conceitos de computação quântica em uma linguagem clássica. Vários exemplos são fornecidos no decorrer do texto, mostrando formas de manipular a camada monádica para realizar transformações em informações quânticas.

Palavras-chave: Computação Quântica. Mônada Quântica. Featherweight Java.

ABSTRACT

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

A QUANTUM OBJECT-ORIENTED LANGUAGE BASED ON FEATHERWEIGHT JAVA

AUTHOR: SAMUEL DA SILVA FEITOSA
ADVISOR: JULIANA KAIZER VIZZOTTO
Defense Place and Date: Santa Maria, June 04st, 2016.

With the approaching end of Moore's Law, where will not be possible to improve the capacity of silicon based processors, the quantum computing appear to be a good choice to provide a new era of computation. Quantum computing can be understood as the art of transform information encoded in the state of a *quantum* physical system. This encoding is through the quantum bits (*qubits*), which can be on *superposition* or *entangled* states, enabling to explore the property called *quantum parallelism*. In this work is discussed the creation of a quantum programming language implementing the object-oriented paradigm (*OO*), allowing manipulation of classes and objects, where the quantum effects are handled through a monadic approach, extending the Featherweight Java (FJ) proposal. This language is formally defined through the *operational semantics*, which allow the implementation in any language that provides *closures*. That language formalization enables us to create an interpreter, implementing the steps of lexical, syntactic and semantic analysis, focusing in the type system to embedded quantum computing concepts in a classical language. Several examples are provided in the text, showing ways to handle the monadic layer in order to perform transformations in quantum information.

Keywords: Quantum Computing. Quantum Monad. Featherweight Java.

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

AST	*Árvore de Sintaxe Abstrata*
Bit	*Binary Digit*
BNF	*Backus-Naur Form*
CT	*Class Table*
DSL	*Domain Specific Language*
FJ	*Featherweight Java*
OO	*Orientação a Objetos*
Qubit	*Quantum Binary Digit*

SUMÁRIO

1 INTRODUÇÃO

Computação quântica é uma área de pesquisa que investiga aspectos da computação considerando a física quântica. Diferentemente do que acontece na computação convencional, a informação em um computador quântico pode apresentar características como *superposição* e *emaranhamento*, as quais permitem a manipulação de combinações de bits quânticos simultaneamente, assim possibilitando um crescimento exponencial no tempo computacional dos computadores. Embora não existam computadores quânticos de propósito geral atualmente, há uma comunidade de pesquisadores dedicando seu tempo e esforço para estudar as propriedades destes dispositivos do futuro e também investigar as novas possibilidades de processamento de informação quântica.

Sem um dispositivo prático a disposição, embora existam pesquisas e projetos de computadores quânticos que permitem manipular poucos *qubits* em laboratório, o trabalho dos pesquisadores ainda é bastante teórico, sendo realizado geralmente através do uso de formalismos desenvolvidos para auxiliar na pesquisa na área de computação quântica. A maioria destes formalismos auxilia no pensamento em relação à teoria da mecânica quântica, entretanto poucos deles são aplicáveis na prática para o projeto de algoritmos quânticos. Sendo assim, uma importante área de pesquisa diz respeito ao projeto de linguagens de programação de alto nível tendo como principal objetivo a descrição de algoritmos quânticos, além de facilitar a compreensão da computação quântica em geral.

Neste contexto, observou-se a possibilidade de utilizar mônadas para estruturar efeitos quânticos (VIZZOTTO; ALTENKIRCH; SABRY, 2006), que posteriormente foram aplicadas para simulação de algoritmos na linguagem Java, através de uma biblioteca que implementa mônadas quânticas utilizando *closures* (CALEGARO; VIZZOTTO, 2013). Seguindo este raciocínio, o objetivo deste trabalho é fornecer uma semântica monádica formal para computação quântica no âmbito de uma linguagem orientada a objetos. Para tal, é descrita a sintaxe, a semântica e o sistema de tipos através de uma semântica operacional, tomando como base o Featherweight Java (FJ) (IGARASHI; PIERCE; WADLER, 2001) - um pequeno cálculo da linguagem Java, com uma definição semântica rigorosa dos seus principais aspectos - transformando-o em uma linguagem quântica orientada a objetos (*FJQuantum*), com o objetivo de testar sua aplicabilidade na implementação de algoritmos quânticos.

Neste projeto o FJ foi utilizado por duas razões: por ser muito compacto, permitindo o

foco em aspectos formais da extensão quântica, e por ser uma representação de uma linguagem orientada a objetos, um dos paradigmas mais utilizados. O fato de utilizar este paradigma pode minimizar os esforços de aprendizado dos conceitos quânticos.

Este trabalho descreve uma extensão para a linguagem Java, que permite a implementação de programas para a computação quântica pura, ou seja, sem a operação de medida. Esta extensão funciona como uma linguagem de domínio específico embutida em Java, e permite a definição de estados quânticos como um tipo especial na linguagem. Como é utilizada uma abordagem monádica, é possível combinar estes estados e usá-los como blocos para construir computações encadeadas. A extensão proposta usa uma característica importante do Java 8: a habilidade de implementar *expressões lambda* e *closures*. Embora esta extensão do Java tenha sido descrita no contexto do FJ, ela poderia ser implementada em qualquer linguagem orientada a objetos que suporte *closures*, como exemplo, C#.

Como uma forma de validar as definições semânticas, também são apresentados os passos realizados para desenvolver um interpretador para a linguagem proposta, codificado na linguagem funcional *Haskell*, implementando a análise léxica, sintática e semântica (sistema de tipos e avaliador semântico) refletindo as especificações apresentadas ao longo deste trabalho.

1.1 Organização do Texto

Este trabalho está dividido da seguinte forma: o Capítulo 2 apresenta os principais conceitos referentes à computação quântica, incluindo *qubits*, operações quânticas e algoritmos quânticos. O Capítulo 3, apresenta algumas linguagens de programação quântica já desenvolvidas, bem como os principais conceitos utilizados para desenvolver este trabalho, cobrindo a abordagem monádica para computação quântica e a proposta original da semântica do FJ. O Capítulo 4 descreve as extensões utilizadas para preparar a linguagem FJ para manipular conceitos de computação quântica, como a adição de tipos genéricos, tipos primitivos, operadores, tuplas e *closures*. O Capítulo 5 apresenta a semântica monádica para possibilitar as definições e transformações dos estados quânticos, além de alguns exemplos de código aceitos pela linguagem proposta. O Capítulo 6 discute os principais aspectos da implementação do interpretador desta linguagem, apresentando as diversas etapas de seu desenvolvimento, como o analisador léxico, sintático e semântico. Por fim, o Capítulo 7 apresenta as conclusões e possíveis trabalhos a serem desenvolvidos no futuro.

2 COMPUTAÇÃO QUÂNTICA

A computação quântica é um campo da teoria da computação que tenta encontrar o que pode ser computado, levando em consideração a física quântica. Essencialmente, a computação clássica pode ser resumida como o processo de processar informações codificadas através do sistema físico clássico, regido pelas partículas em escala macroscópica. Deste ponto de vista, a computação quântica pode ser vista como a tarefa de processar informações codificadas em um sistema físico quântico, definido pelo comportamento das partículas em escala microscópica (YANOFSKY; MANNUCCI, 2008).

Este capítulo tem como objetivo explanar os diferentes aspectos da computação quântica de um ponto de vista algorítmico, sempre que possível fazendo uma relação com a computação clássica para facilitar a compreensão do leitor. Entretanto, alguns conceitos chave deste modelo de computação não possuem análogos na computação clássica, como o paralelismo quântico, emaranhamento, operação de medida, e serão apresentados diretamente.

2.1 Revisão Histórica

Há quase um século da descoberta da mecânica quântica, mais de meio século da invenção da teoria da informação e da chegada da computação digital, foi percebido que a física quântica altera profundamente as características do processamento da informação. A mecânica quântica provê um novo paradigma que não foi imaginado antes dos anos 80 e que o poder não foi completamente apreciado até metade dos anos 90 (MERMIN, 2007).

A primeira vez que pensou-se em utilizar física quântica na computação foi em 1982 (FEYNMAN, 1982), onde sugeriu-se que somente através da Computação Quântica seria possível simular com eficiência os efeitos da natureza. Nesta época sugeriu-se que sistemas de mecânica quântica são mais poderosos que a computação clássica (YANOFSKY; MANNUCCI, 2008). Então foi proposto um modelo básico de computador quântico que seria capaz de realizar essas simulações.

Em 1985 (DEUTSCH, 1985), propôs-se um modelo de computador quântico universal, com o intuito de prover um modelo mais poderoso de computação. Ainda é impossível afirmar com clareza se o modelo proposto é capaz de simular eficientemente um sistema físico arbitrário. Provar ou refutar esta conjectura é um dos maiores problemas abertos no campo de computação e informação quântica. Apesar disso, o esforço de Deutsch nos permite desafiar

a tese de Church-Turing (NIELSEN; CHUANG, 2000), considerando um simples exemplo de algoritmo quântico que sugere que, de fato, computadores quânticos podem ter poderes computacionais excedendo os computadores clássicos.

Em 1989, Deutsch descreveu um segundo modelo de computação quântica: o circuito quântico. Neste esforço, ele demonstrou que circuitos quânticos podem realizar qualquer computação clássica e também computar as mesmas informações passíveis de execução na descrição do computador quântico universal. Através do estudo de Andrew Chi-Chih Yao (YAO, 1993), foi demonstrado que os circuitos quânticos podem computar os mesmos problemas que uma máquina de Turing quântica, permitindo aos pesquisadores focarem em circuitos quânticos, pela sua facilidade de compreensão perante as construções das máquinas de Turing quânticas (YANOFSKY; MANNUCCI, 2008). Além disso, Deutsch foi o primeiro a publicar um algoritmo quântico em 1989, que utiliza os conceitos do paralelismo quântico, sendo capaz de processar duas operações simultaneamente, através da utilização de um *qubit* em superposição.

Este primeiro passo dado por Deutsch culminou em várias pesquisas na área de computação quântica, sendo demonstrado por Peter Shor em 1994 a resolução de outros importantes problemas da computação clássica, o problema de encontrar os fatores primos de um inteiro e o então chamado "logaritmo discreto", que poderiam ser resolvidos de forma eficiente por computadores quânticos. Shor desencadeou várias pesquisas nesta área, fazendo surgir os campos da computação, informação e complexidade quântica. Em 1995 apareceu outro nome importante no cenário, Lov Grover mostrou que pesquisas em espaços de busca não estruturados também podem ser melhoradas por um computador quântico (NIELSEN; CHUANG, 2000).

Nas últimas décadas também ocorreu certo progresso no desenvolvimento experimental de hardware quântico. O contexto para o desenvolvimento deste computador foi aprimorado. Um computador quântico não será uma versão mais rápida, maior ou menor que um computador atual. Ao invés disso, será um tipo totalmente diferente de máquina, construída para operar coerentemente com aspectos da mecânica quântica para diferentes aplicações (T. D. LADD F. JELEZKO; O'BRIEN, 2010).

Já foi demonstrado que é possível resolver ou melhorar problemas clássicos, porém, atualmente não é possível afirmar que um computador quântico será sempre mais rápido que um computador clássico. A computação clássica, apesar de também ser recente, possui vários campos de estudo, e o projeto de algoritmos já é bem conhecido. Na computação quântica, o projeto de algoritmos é complexo, pois os projetistas deparam-se com a intuição clássica que

está enraizada desde o início do aprendizado sobre a computação, e também que não basta apenas escrever um algoritmo quântico, este tem de ser melhor dos que os clássicos já apresentados (SHOR, 2005).

2.2 Processamento de Informação Quântica

Na computação clássica a menor unidade de informação é conhecida por *bit*, o qual pode representar apenas um dos valores 0 ou 1 em um dado momento. Já na computação quântica, a menor unidade de informação é chamada de *qubit*, que pode representar os valores 0, 1 ou ambos ao mesmo tempo, possibilitando explorar algumas características da física quântica, como a *superposição* e o *emaranhamento*, possibilitando um ganho computacional considerável.

A seguir serão expostos diversos conceitos matemáticos necessários para compreender o modelo matemático que compõe a teoria da computação quântica.

2.2.1 Espaço de Hilbert

Um espaço de *Hilbert* é uma generalização do espaço euclidiano que não precisa estar restrita a um número finito de dimensões. É um espaço vetorial dotado de produto interno, apresentando noções de distância e ângulos. Elementos no espaço de Hilbert são chamados de vetores, os quais na teoria quântica representam e mantêm informações completas sobre um estado físico, sendo apresentados através da notação Dirac, denotados por $|\alpha\rangle$ (NIELSEN; CHUANG, 2000).

2.2.2 Produtos Tensoriais

O estado de um sistema físico quântico composto de vários *qubits* é expresso pelo produto tensorial dos vetores que representam cada *qubit* individualmente. Podemos expressar o produto tensorial dos vetores $P \otimes Q$ conforme apresentado na Equação 2.1.

$$P \otimes Q = \begin{pmatrix} P11 \begin{pmatrix} Q11 \\ \vdots \\ Q1n \end{pmatrix} \\ \vdots \\ P1m \begin{pmatrix} Q11 \\ \vdots \\ Q1n \end{pmatrix} \end{pmatrix} \tag{2.1}$$

Supondo os vetores P e Q da Equação 2.2:

$$P = \begin{pmatrix} a \\ b \end{pmatrix} \qquad Q = \begin{pmatrix} c \\ d \end{pmatrix} \tag{2.2}$$

O produto tensorial é apenas a multiplicação de cada item do vetor P por todos os valores do vetor Q, conforme demonstrado na Equação 2.3.

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix} \tag{2.3}$$

Assim como os *qubits* são representados por vetores, as operações unitárias são representadas por matrizes, que também são passíveis de aplicação do produto tensorial. Supondo as matrizes P e Q da Equação 2.4:

$$P = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad Q = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{2.4}$$

O produto tensorial das matrizes $P \otimes Q$ é apresentado na Equação 2.5:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{12} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ a_{21} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{22} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \end{pmatrix} \tag{2.5}$$

Onde o resultado é apresentado na Equação 2.6.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{12} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix} \tag{2.6}$$

É necessário compreender a operação produto tensorial, para facilitar a compreensão dos algoritmos nos circuitos quânticos.

2.2.3 Notação Dirac

Paul Dirac (DIRAC, 1939) propôs uma notação padrão para descrever os estados quânticos, sendo utilizada para expressar os *qubits* de maneira concisa. Esta notação também é conhecida por *Bra-Ket*. Sendo assim, é denotado por *ket* um vetor coluna qualquer ψ por $|\psi\rangle$, e é denotado por *bra* seu transposto conjugado $\langle\psi|$.

Utilizando-se desta notação, pode-se expressar os análogos aos *bits* clássicos 0 e 1 através do *ket*, para os *qubits* $|0\rangle$ e $|1\rangle$ respectivamente e seus vetores coluna são demonstrados na Equação 2.7:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad e \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.7}$$

As informações nestes vetores coluna, representam as amplitudes de probabilidade de um *qubit*, sendo possível existir combinações lineares de $|0\rangle$ e $|1\rangle$, conforme apresentado na Equação 2.8:

$$\alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.8}$$

A seguir serão apresentados conceitos detalhados sobre os *qubits* e sua função na computação quântica.

2.2.4 Qubits

O *qubit* ou *bit* quântico é a menor unidade de informação de um sistema quântico, sendo o conceito análogo do *bit* da computação clássica. Um *bit* tem a capacidade de representar duas informações distintas, 0 ou 1, e qualquer informação na computação atual é mapeada através de combinações destes *bits*.

A computação quântica difere-se da clássica principalmente em virtude das características especiais dos *bits* quânticos. Estes não são restritos aos seus estados básicos, 0 e 1, que são chamados de estados puros, podendo estar em uma superposição de estados, significando que o *qubit* pode estar efetivamente em ambos os estados puros ao mesmo tempo. A interpretação destas características vêm dos estudos preliminares referentes à mecânica quântica (MERMIN, 2007).

Geralmente os *qubits* são descritos através da notação *Bra-Ket*, onde os estados puros ou estados base são descritos como $|0\rangle$ e $|1\rangle$. Como mencionado anteriormente, um estado quântico é descrito através de amplitudes de probabilidades de seus estados base, onde α e β indicam a possibilidade de um estado estar em $|0\rangle$ ou em $|1\rangle$, ou em uma *superposição*. É importante notar que estas probabilidades combinadas devem respeitar a regra: $|\alpha|^2$+$|\beta|^2$ = 1. Intuitivamente, pode-se afirmar que um *qubit* pode estar em $|0\rangle$, $|1\rangle$ ou em ambos os estados simultaneamente (NIELSEN; CHUANG, 2000).

A habilidade dos *qubits* de estarem em uma combinação linear dos estados básicos é responsável por uma das principais características da computação quântica, o *paralelismo quântico*. Essa característica é muito explorada na construção de algoritmos quânticos, e será abordada nas seções seguintes.

A representação matemática de um *bit* quântico se dá através de um vetor coluna, o qual é descrito através de suas amplitudes de probabilidade α e β, conforme apresentado na Equação 2.9:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.9}$$

Onde α representa a probabilidade do qubit estar em $|0\rangle$ e β representa a probabilidade do qubit estar em $|1\rangle$.

O *qubit* apresentado na Equação 2.10 está descrito com 100% de probabilidade do qubit estar em 0 e 0% de probabilidade de estar em 1, ou seja, está no estado puro $|0\rangle$.

$$|\psi\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2.10}$$

Já na Equação 2.11, o *qubit* está com 0% de probabilidade de estar em $|0\rangle$ e 100% de probabilidade de estar em $|1\rangle$, estando no seu estado puro $|1\rangle$.

$$|\psi\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.11}$$

Um estado quântico de N *qubits* pode ser expresso como um vetor em um espaço de dimensão 2^N. Essa combinação é dada através da aplicação do produto tensorial ($\otimes$) dos N vetores dos *qubits*. Por exemplo, pode-se escrever o estado de dois *qubits* $|0\rangle \otimes |1\rangle$, como $|01\rangle$. A representação matemática de dois *qubits* pode ser vista na Equação 2.12.

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.12}$$

Outra característica incomum dos estados quânticos é que suas amplitudes de probabilidade são inobserváveis, ou seja, uma verificação do estado atual de um registrador quântico incorre em um colapso, destruindo a superposição dos *qubits* e resultando apenas em um de seus estados básicos $|0\rangle$ e $|1\rangle$. Esta característica é contrária à computação clássica, que depende da observação de seus estados em cada leitura e gravação que é feita em memória (NIELSEN;

CHUANG, 2000). Um *qubit* em *superposição* pode ser representado pelo estado apresentado na Equação 2.13.

$$\frac{1}{\sqrt{2}} \mid 0\rangle + \frac{1}{\sqrt{2}} \mid 1\rangle \tag{2.13}$$

O estado mostrado acima é muito usado, e apresenta, quando medido o resultado 0 com 50% de probabilidade ($|1/\sqrt{2}|^2$) e resultado 1, também com 50% de probabilidade. O espaço de estados de um *qubit* pode ser representado geometricamente através da esfera de Bloch (NIELSEN; CHUANG, 2000), conforme a Figura 2.1, que além de demonstrar os estados puros do *qubit*, ainda apresenta o eixo complexo que denota suas amplitudes de probabilidade.

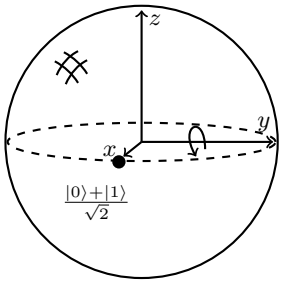


Figura 2.1 – Esfera de *Bloch* representando um estado em *superposição*.

Observando a infinidade de possíveis pontos da esfera de *Bloch*, pode-se imaginar o poder de representação de um único *qubit*, de modo a armazenar uma grande quantidade de informação. Entretanto, essa percepção é inválida. Como já mencionado, a partir da medição de um *qubit*, este entra em colapso, apresentando apenas um de seus estados puros, o que frusta a ideia do poder de armazenamento de um *qubit*. Por exemplo, se uma medição de um estado em superposição $|0\rangle + |1\rangle / \sqrt{2}$, retornar $|0\rangle$, então, o *qubit* deixa o estado de superposição e a partir daí passará a ter o valor $|0\rangle$ para o restante do circuito quântico (NIELSEN; CHUANG, 2000).

2.2.5 Operações Quânticas

Estados quânticos podem ser manipulados de duas maneiras: transformações unitárias e operações de medida. De forma análoga ao que ocorre na computação clássica, onde informações são transformadas através de circuitos e portas lógicas, em um computador quântico as operações se dão através de circuitos quânticos, por meio de portas quânticas, que manipulam e transformam a informação (NIELSEN; CHUANG, 2000).

2.2.5.1 Transformações Unitárias

Transformações unitárias são operações que se aplicam sobre os *qubits* de modo a alterar os valores dos mesmos. Essas operações são representadas por matrizes, e podem ser aplicadas sobre um ou mais *qubits*. Às operações que representam as transformações físicas do estado quântico são chamadas de *portas quânticas*.

A aplicação de uma porta sobre um estado quântico é, matematicamente, a multiplicação da matriz que representa esta transformação pelo vetor coluna que engloba o atual estado do sistema quântico. Este modo de compreender as transformações de estados quânticos é necessário para possibilitar o desenvolvimento de simuladores em computadores clássicos.

Portas aplicáveis a um Qubit

Na computação clássica utilizam-se circuitos lógicos que manipulam bits através da aplicação de portas lógicas, convertendo um ou mais *bits* de um valor para outro. A única operação clássica aplicável a apenas um *bit* é a porta *NOT*, que transforma 0 em 1 e vice-versa (NIELSEN; CHUANG, 2000).

A ideia das aplicações de portas para transformar informações é herdada na computação quântica. Pode-se citar como exemplo a porta *NOT quântica*, que transforma o *qubit* $|0\rangle$ em $|1\rangle$ e vice-versa. A Figura 2.2 apresenta o circuito quântico que aplica a transformação *NOT* a um *qubit*.

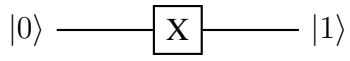


Figura 2.2 – Aplicação da porta X (not) sobre o *qubit* $|0\rangle$.

A representação matemática pode ser vista na equação 2.14.

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.14}$$

É importante notar que a aplicação da porta *NOT* não diz nada sobre o que acontece com as superposições dos estados $|0\rangle$ e $|1\rangle$. Ele simplesmente leva o estado $\alpha|0\rangle + \beta|1\rangle$ para $\alpha|1\rangle + \beta|0\rangle$. Do mesmo modo apresentado na equação anterior, a matriz correspondente a porta *NOT* quântica é apresentada na equação 2.15:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.15}$$

Diferentemente da computação clássica, que possui somente a porta *NOT* aplicável a apenas um *bit*, na computação quântica, existem diversas outras com funcionalidades distintas, sendo as mais importantes apresentadas a seguir.

A equação 2.16 apresenta a porta de *Hadamard*, utilizada para colocar um *qubit* em superposição.

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{ou} \quad \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.16}$$

Esta porta transforma $H\left|0\right\rangle$ em $1/\sqrt{2}(\left|0\right\rangle + \left|1\right\rangle)$ e $H\left|1\right\rangle$ em $1/\sqrt{2}(\left|0\right\rangle - \left|1\right\rangle)$, fazendo com que os *qubits* estejam em ambos os estados $\left|0\right\rangle$ e $\left|1\right\rangle$ ao mesmo tempo, com probabilidade de 50% para cada um. Caso aplique-se duas vezes esta porta sobre o mesmo *bit* quântico, este retorna ao seu estado original (NIELSEN; CHUANG, 2000).

A equação 2.17 apresenta a porta *Pauli-Z*, que deixa o $\left|0\right\rangle$ sem modificações e troca o sinal do $\left|1\right\rangle$ para -$\left|1\right\rangle$.

$$Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.17}$$

A equação 2.18 apresenta a porta *Pauli-I*, que é a função identidade, mantendo os *qubits* em seu estado original. Esta porta possui uma função essencial para a manutenção do estado quântico, e mais detalhes serão apresentados na sequência.

$$I \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.18}$$

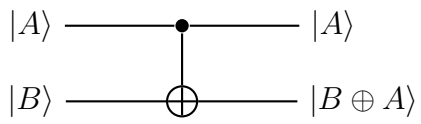
Uma premissa da computação quântica é que suas transformações sejam reversíveis, ou seja, deve ser possível retornar ao estado anterior da aplicação das portas quânticas.

Ainda existem outras operações que são passíveis de execução sobre apenas um *qubit* (MERMIN, 2007). Além das portas aplicáveis a um único *qubit*, existem outras que aplicam-se a dois ou mais. A computação quântica possui um conjunto de portas quânticas universais, que, somente a partir delas é possível criar qualquer outra operação envolvendo múltiplos *qubits* (NIELSEN; CHUANG, 2000).

Portas aplicáveis a mais de um Qubit

Na computação quântica exitem diversas portas que podem ser aplicadas sobre mais de um *qubit*. A porta *controlled-NOT* ou *CNOT*, serve como protótipo para construção de operações mais complexas envolvendo vários *qubits*. Qualquer operador unitário para múltiplos

qubits pode ser representado através da combinação da porta *CNOT* e das portas aplicáveis a apenas um *qubit*, apresentadas anteriormente. A representação em circuito desta porta e sua matriz unitária são apresentadas na Figura 2.3 (NIELSEN; CHUANG, 2000).



$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figura 2.3 – Circuito quântico e matriz unitária da porta *CNOT*.

Uma porta *controlada* age dependendo do valor do *qubit* de controle, ou seja, o processo é executado somente se este estiver com o estado em $|1\rangle$, caso contrário, o estado permanece idêntico ao original. É importante notar que os *qubits* podem estar em estado de superposição.

A ação da porta *CNOT* pode ser caracterizada pelas transformações operadas nos elementos da base computacional, conforme visto na equação 2.19.

$$\begin{aligned} |00\rangle &\rightarrow |00\rangle \\ |01\rangle &\rightarrow |01\rangle \\ |10\rangle &\rightarrow |11\rangle \\ |11\rangle &\rightarrow |10\rangle \end{aligned} \tag{2.19}$$

Esta ação na base computacional pode ser representada matematicamente, pela expressão 2.20.

$$|i, j\rangle \rightarrow |i, i \oplus j\rangle \tag{2.20}$$

Onde i,$j \in \{0, 1\}$ e $\oplus$ é a adição módulo 2.

Outra porta importante aplicável a múltiplos *qubits* é a porta *Toffoli*, que também é uma porta controlada, sendo aplicável a 3 ou mais *qubits*. Neste caso, existem dois ou mais *qubits* de controle. Sua ação computacional atuando sobre três *qubits* é apresentada na expressão 2.21.

$$|i, j, k\rangle \rightarrow |i, j, k \oplus ij\rangle \tag{2.21}$$

Onde i, j, $k \in \{0, 1\}$ e $\oplus$ é a adição módulo 2. A Figura 2.4 apresenta a representação desta porta em um circuito quântico.

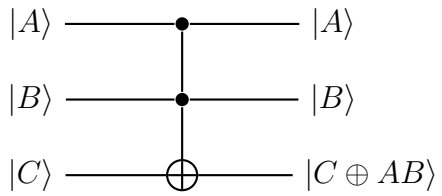


Figura 2.4 – Circuito quântico representando a porta *Toffoli*.

A porta *Toffoli* é utilizada para simplificar a construção de circuitos quânticos, porém, pode ser representada pela combinação de porta aplicáveis a um *qubit* e a *CNOT*.

2.2.5.2 Operação de Medida

Outro tipo de operação para manipulação de estados quânticos é a operação de medida, a qual, diferentemente das transformações unitárias, não é reversível. Medições são operações destrutivas, que transformam um estado quântico em um dos seus estados base, de acordo com suas amplitudes de probabilidade. Este processo é chamado de *colapso*.

A operação de medida é o único modo de obter informações de um estado quântico. Enquanto a medição não é realizada, o estado permanece como não observado e podem existir superposições de estados base. No momento que estes são observados, é obtido um estado clássico novamente. Sendo assim, a operação de medida tem como objetivo extrair informações clássicas das informações quânticas, com o custo de destruir o estado quântico (YANOFSKY; MANNUCCI, 2008).

É importante notar que as medições são probabilísticas, significando que podemos obter resultados distintos a partir da observação de um mesmo sistema quântico. Mais especificamente, pode-se executar uma medição que projeta o *qubit* $\alpha|0\rangle$ + $\beta|1\rangle$ nas bases $\{|0\rangle,|1\rangle\}$. Então, após a medida, o estado será $|0\rangle$ com probabilidade $|\alpha|^2$, ou $|1\rangle$ com probabilidade $|\beta|^2$. Além disso, depois da aplicação da medida, o *qubit* estará em um de seus estados base, e operações de medida subsequentes não alterarão o seu estado, até que novas operações quânticas sejam aplicadas (VIZZOTTO, 2006).

2.2.6 Características de Estados Quânticos

O modelo de computação quântica tem o objetivo de aplicar as leis que regem o funcionamento das partículas na mecânica quântica, que diferem das regras aplicadas na computação atual (SIMON, 1994). Vários algoritmos utilizando essas novas regras foram propostos, indicando fortemente que sua aplicação em um computador quântico pode realmente aumentar a velocidade de execução de diversos problemas atuais. A seguir serão apresentados algumas das principais características que proveem o poder da computação quântica.

2.2.6.1 Emaranhamento Quântico

Emaranhamento é um fenômeno fundamental na computação quântica. Resumidamente, um par de *qubits* em um estado quântico é dito *emaranhado* se ele não puder ser expresso através do produto tensorial de *qubits* individuais. É um elemento chave nos efeitos como teleportação quântica, algoritmos quânticos e correções de erros, sendo de grande utilidade na computação e informação quântica, apesar de suas características ainda não serem totalmente compreendidas pelos pesquisadores (NIELSEN; CHUANG, 2000).

Neste fenômeno da mecânica quântica, no qual os estados quânticos de dois ou mais objetos são descritos como referência entre eles, mesmo que os objetos individuais estejam separados no espaço, é possível preparar duas partículas em um estado quântico único, de modo que quando um deles é observado, o outro sempre será influenciado por esta medição, não importando a distância entre eles.

O emaranhamento quântico é a base para tecnologias emergentes, como computação quântica e criptografia quântica, também sendo usado em experimentos de teleportação quântica. Em 1935, Einstein, Podolski e Rosen formularam o *paradoxo EPR*, demonstrando que o emaranhamento faz da mecânica quântica uma teoria *não-local*. Este fenômeno também é conhecido como *efeito fantasmagórico a distância* (NIELSEN; CHUANG, 2000). Este efeito *não-local* é algo que não pode acontecer do ponto de vista da física clássica, sem uma comunicação instantânea entre as partículas (GRUSKA, 2000).

No ponto de vista computacional, o emaranhamento tem um papel central na teoria da informação quântica, sendo capaz de interconectar duas partes fisicamente distantes, não permitindo a sua cópia, sem possibilitar a interceptação da informação, além de auxiliar na melhoria de performance dos algoritmos quânticos. É uma das principais razões da computação

quântica não poder ser simulada eficientemente por computadores clássicos (GRUSKA, 2000).

2.2.6.2 Paralelismo Quântico

O paralelismo quântico é a característica fundamental dos algoritmos quânticos. Simplificando, o paralelismo quântico permite ao computador executar uma função $f(x)$ para muitos valores diferentes simultaneamente (NIELSEN; CHUANG, 2000).

Geralmente, a computação de uma função f é determinada por uma operação unitária que age sobre dois ou mais qubits, sendo que, na maioria dos casos, os primeiros representam os dados de entrada para a função, e o último sendo utilizado para armazenar a própria função e permitir a simultaneidade do processamento. É conveniente considerar os qubits do modo $|x, y\rangle$. Com uma sequência apropriada de transformações através das portas lógicas quânticas, é possível transformar este estado em $|x, y \oplus f(x)\rangle$, ou seja:

$$U_f \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle \tag{2.22}$$

Sendo assim, podemos observar que, se $y = 0$, então:

$$U_f \left|x\right\rangle \left|0\right\rangle = \left|x\right\rangle \left|0 \oplus f(x)\right\rangle = \left|x\right\rangle \left|f(x)\right\rangle \tag{2.23}$$

Na Figura 2.5 é apresentada a função unitária que trabalha com 2 *qubits* de entrada. Sendo o primeiro passado em um estado de superposição (NIELSEN; CHUANG, 2000).

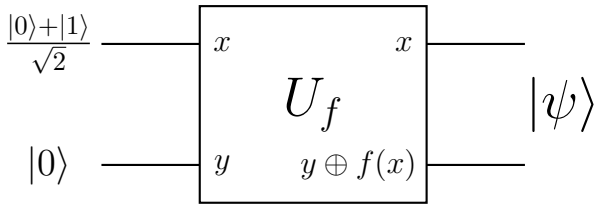


Figura 2.5 – Função unitária para avaliar $f(0)$ e $f(1)$ simultaneamente.

Note que, ao aplicar esta função unitária sobre os qubits $|0\rangle + |1\rangle / \sqrt{2}$ e $|0\rangle$, teremos como resultado a expressão 2.24:

$$\frac{|0, f(0)\rangle + |1, f(1)\rangle}{\sqrt{2}} \tag{2.24}$$

Ou seja, o circuito realiza a computação de todos os valores em uma única chamada, executando $f(0)$ e $f(1)$ simultaneamente. O mesmo ocorreria se fossem passados mais de

2 qubits para a função unitária. Esta característica incomum de calcular todos os valores da função f ao mesmo tempo é chamada de paralelismo quântico. Entretanto, após realizar o processamento das entradas, ainda é necessário realizar operações para que seja possível interpretar seu resultado (JÚNIOR; LIMA, 2006).

2.3 Algoritmos Quânticos

De forma similar aos algoritmos que executam em computadores clássicos, os algoritmos quânticos são descritos através de um conjunto finito de instruções ordenadas e não ambíguas, executando em um computador quântico. Os algoritmos quânticos fazem uso de diversas propriedades específicas do mundo quântico, como a superposição quântica, para transformar entradas clássicas, através de estados emaranhados, para saídas clássicas mais eficientes do que em algoritmos clássicos (GRUSKA, 2000).

É importante notar que qualquer problema que pode ser resolvido por um computador quântico, também pode ser resolvido por um computador clássico. O grande interesse nos computadores quânticos é sua capacidade de resolver alguns problemas mais rapidamente que os clássicos (YANOFSKY; MANNUCCI, 2008). Projetar algoritmos quânticos mais rápidos que os seus análogos clássicos tem sido foco de pesquisa e apresentam desafios intelectuais para os pesquisadores, devido as propriedades não triviais da mecânica quântica, que apresentam um novo paradigma para esta tarefa (GRUSKA, 2000).

2.3.1 Introdução

Muitos algoritmos clássicos foram desenvolvidos muito tempo antes de existirem máquinas nas quais eles deveriam executar. Algoritmos clássicos foram desenvolvidos milênios antes dos computadores, e similarmente, foram descritos vários algoritmos quânticos antes de qualquer computador quântico de larga escala. Estes algoritmos manipulam *qubits* para resolver problemas e, em geral, eles resolvem estas tarefas mais eficientemente do que em computadores clássicos (YANOFSKY; MANNUCCI, 2008).

Algoritmos quânticos são descritos através de funções disponíveis em processadores quânticos (ou simuladores) para resolver problemas computáveis. Em uma explanação mais técnica, o projeto de um algoritmo quântico pode ser visto como o processo de uma decomposição eficiente de transformações unitárias complexas em produtos de operações unitárias

elementares ou portas quânticas (GRUSKA, 2000).

Todo algoritmo quântico trabalha sobre o seguinte *framework* básico (YANOFSKY; MANNUCCI, 2008):

- O sistema inicia com *qubits* em um estado clássico particular.
- A partir daí, o sistema é colocado em uma superposição de estados.
- Seguido pela transformação deste estado através de várias operações unitárias.
- Para finalmente realizar a operação de medida nos *qubits*.

Este esquema pode apresentar variações, porém, para compreensão dos algoritmos seguintes, é importante compreender estes passos.

2.3.2 Algoritmo de Deutsch

O algoritmo de Deutsch é o exemplo mais simples que demonstra o poder do paralelismo quântico. Sua primeira versão foi apresentada por David Deutsch (DEUTSCH, 1985), que junto com o trabalho de Richard Feynman (FEYNMAN, 1982), lançaram todo o campo da computação quântica (MERMIN, 2007).

Este algoritmo tem o objetivo de determinar se uma função *booleana* é balanceada ou constante. Se $f(0) = f(1)$, a função é dita constante, caso contrário, é dita balanceada. Classicamente, para resolver este problema é necessário avaliar a função f duas vezes, sendo $f(0)$ e $f(1)$, para posteriormente comparar o seu resultado (VIZZOTTO, 2006). Na versão quântica, o algoritmo resolve o problema com apenas uma verificação, ou seja, metade do tempo de seu respectivo clássico (MERMIN, 2007), utilizando o princípio da superposição de dois estados básicos ao mesmo tempo, o que permitirá a avaliação das duas entradas em uma única verificação.

Para desenvolver a versão quântica para este algoritmo, primeiramente, é necessário construir uma versão quântica da função f, que é uma transformação unitária U_f que executa a mesma computação de f. Como a computação quântica deve ser reversível, é possível reconstruir a função f utilizando dois *qubits*, como demonstrado na expressão 2.25 (VIZZOTTO, 2006).

$$U_f \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle \tag{2.25}$$

O circuito quântico que utiliza a função U_f é apresentado na Figura 2.6.

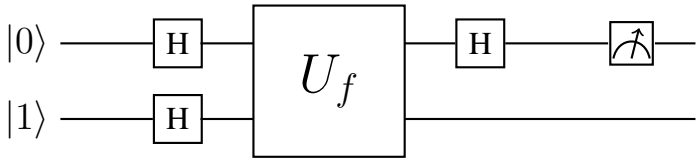


Figura 2.6 – Circuito quântico para o algoritmo de *Deutsch*.

O truque utilizado no algoritmo é aplicar U_f aos estados em superposição, conforme a expressão 2.26.

$$|+\rangle\,|-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \tag{2.26}$$

Os estados $|+\rangle$ e $|-\rangle$ representam a superposição de estados, iniciados por $|0\rangle$ e $|1\rangle$ respectivamente, conforme apresentado na equação 2.27.

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{e} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.27}$$

Estes estados podem ser obtidos através da aplicação de $H \otimes H$ em $|01\rangle$, conforme ocorre na parte inicial do algoritmo de Deutsch, visto na Figura 2.6.

A expressão 2.28 apresenta o resultado da aplicação da função U_f sobre entradas $|0\rangle$ e $|1\rangle$ no segundo *qubit*.

$$\begin{aligned} U_f\,|x\rangle\,|0\rangle &= |x\rangle\,|f(x)\rangle \\ U_f\,|x\rangle\,|1\rangle &= |x\rangle\,|1 \otimes f(x)\rangle \end{aligned} \tag{2.28}$$

Utilizando estas equações para calcular $U_f|x\rangle|-\rangle$, tem-se o resultado da equação 2.29.

$$\begin{aligned} U_f\,|x\rangle\,|-\rangle &= \tfrac{1}{2}\,|x\rangle\,(|0\rangle - |1\rangle) \quad \text{se } f(x) = 0 \\ U_f\,|x\rangle\,|-\rangle &= \tfrac{1}{2}\,|x\rangle\,(|1\rangle - |0\rangle) \quad \text{se } f(x) = 1 \end{aligned} \tag{2.29}$$

A expressão 2.30 representa a função quântica que determina o resultado da execução do algoritmo.

$$U_f\,|x\rangle\,|-\rangle = (-1)^{f(x)}\,|x\rangle\,|-\rangle \tag{2.30}$$

Sendo assim, a expressão 2.31 interpreta o resultado da aplicação da função quântica.

$$U_f\,|+\rangle\,|-\rangle = \begin{cases} |+\rangle\,|-\rangle & \text{se } f(0) = f(1) \\ |-\rangle\,|-\rangle & \text{se } f(0) \neq f(1) \end{cases} \tag{2.31}$$

E a informação sobre se a função é constante ou balanceada fica concentrada no primeiro *qubit*, podendo ser obtida através da última aplicação de *hadamard* sobre o primeiro *qubit*, e efetuando a sua medida, conforme visto em 2.32 (VIZZOTTO, 2006).

$$\begin{aligned} |0\rangle\,|-\rangle \text{ se } f(0) = f(1) \\ |1\rangle\,|-\rangle \text{ se } f(0) \neq f(1) \end{aligned} \tag{2.32}$$

2.3.3 Algoritmo de Grover

O algoritmo de Grover apresenta uma alternativa para buscas de elementos em bases de dados não estruturadas. Enquanto na computação clássica, no pior caso, seriam necessárias n consultas para encontrar o elemento desejado, para a computação quântica, Lov Grover (GROVER, 1996) propôs um algoritmo que resolve o problema em $\sqrt{n}$ consultas. Embora existam algoritmos quânticos que apresentam uma melhora mais significativa com relação a computação clássica, como o algoritmo de Deutsch-Jozsa, o algoritmo de Simon e o algoritmo de Shor, ainda assim, ele apresenta uma melhora considerável. Este algoritmo tem aplicações principalmente na teoria de bancos de dados.

A definição do algoritmo se dá inicialmente pela representação matemática do problema. Supondo que a busca do elemento ocorre sobre uma lista $\{0, 1, ..., N-1\}$, onde $N = 2^n$ para algum número natural n, e que a função que realiza o processamento tenha o formato da expressão 2.33.

$$f : \{0, 1, ..., N-1\} \longrightarrow \{0, 1\} \tag{2.33}$$

Esta função tem sua definição apresentada na expressão 2.34.

$$f(x) = \begin{cases} 1, & \text{se } x = x_0 \\ 0, & \text{se } x \neq x_0 \end{cases} \tag{2.34}$$

A função tem por objetivo encontrar x_0, assumindo que exista apenas um elemento $x \in \{0, 1, ..., N-1\}$, tal que $f(x) = 1$. O custo computacional para resolver este problema é medido pelo número de vezes que a função f é invocada. A função f, por sua vez, pode ser vista como um oráculo, capaz de informar se o dado elemento é ou não aquele procurado.

O operador Grover G necessita, para o cálculo de uma função de n *bits*, ter a possibilidade de manipular $n + 1$ entradas e saídas. As primeiras n entradas constituem as informações relacionadas aos elementos da lista onde será feita a busca, e a última possui um papel auxiliar.

O circuito quântico que apresenta o algoritmo de Grover pode ser visto na Figura 2.7 (NIELSEN; CHUANG, 2000). A representação G no circuito é um operador unitário que será descrito posteriormente.

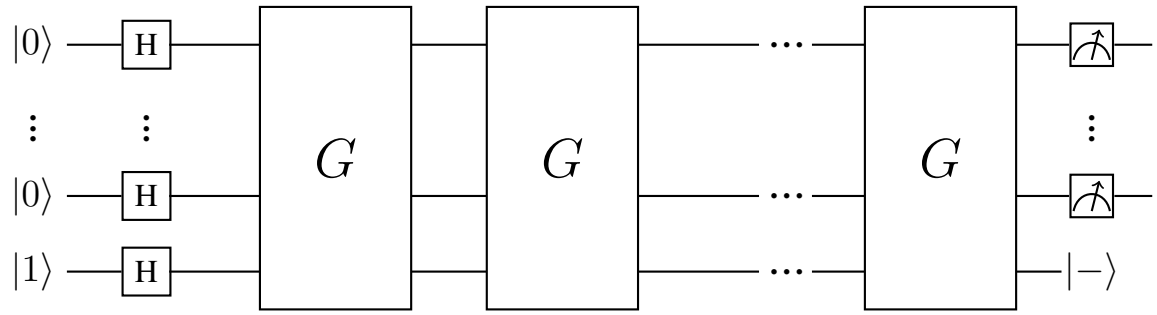


Figura 2.7 – Circuito quântico para o algoritmo de *Grover*.

Como podemos perceber, o circuito começa de forma semelhante ao algoritmo de Deutsch, colocando todos os *qubits* de entrada em superposição, através da execução das portas de *hadamard*. A partir desta execução, gera-se o estado apresentado na expressão 2.35.

$$|\psi_0\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{2.35}$$

Após isto, aplica-se o operador unitário G um certo número de vezes. Este operador consiste da aplicação de uma porta unitária U_f, responsável por realizar a consulta ao oráculo para determinar se o elemento atual é aquele sendo pesquisado, seguido por uma série de aplicações da porta de *hadamard* e transformações de fase nos primeiros n *qubits*. A Figura 2.8 apresenta o detalhamento do operador de Grover G (NIELSEN; CHUANG, 2000).

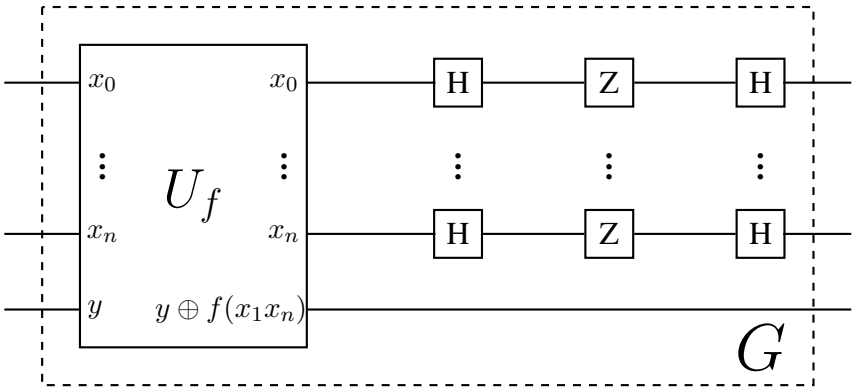


Figura 2.8 – Operador quântico de *Grover*.

Para apresentar uma representação quântica da função f, vista na equação 2.33, que é utilizada para identificar o elemento procurado, cria-se o operador linear U_f de modo a trans-

formar $|x\rangle$ em $|f(x)\rangle$, onde $|x\rangle$ é representado pelos n *qubits* superiores. A operação da função U_f é apresentada na expressão 2.36.

$$|x\rangle |y\rangle \xrightarrow{U_f} |x\rangle |y \oplus f(x)\rangle \tag{2.36}$$

O resultado do processamento da função U_f é apresentado na expressão 2.37.

$$U_f(|x\rangle |0\rangle) = \begin{cases} |x\rangle |1\rangle, & \text{se } x = x_0 \\ |x\rangle |0\rangle, & \text{se } x \neq x_0 \end{cases} \qquad U_f(|x\rangle |1\rangle) = \begin{cases} |x\rangle |0\rangle, & \text{se } x = x_0 \\ |x\rangle |1\rangle, & \text{se } x \neq x_0 \end{cases} \tag{2.37}$$

Como pode ser percebido, o operador U_f altera o estado do último *qubit* quando os primeiros representam o elemento procurado. Sendo assim, para identificar a informação sendo procurada x_0, basta aplicar a porta quântica U_f em cada estado associado aos elementos da lista e manter o segundo registrador no estado $|0\rangle$ ou $|1\rangle$. Quando este estado for alterado significa que o elemento foi encontrado. Porém, isso não proporciona ganho algum em relação ao algoritmo clássico, utilizando a função f. A melhoria de performance na versão quântica se dá pela capacidade de aplicar a operação unitária U_f sobre estados em superposição.

Quando a operação U_f é aplicada sobre uma superposição, ela não inverte o valor do *qubit* como demonstrado anteriormente, porém, inverte o sinal da amplitude de probabilidade do elemento procurado, de $\frac{1}{\sqrt{n}}$ para $-\frac{1}{\sqrt{n}}$. É importante notar, que neste caso, a avaliação da função f ocorreu simultaneamente sobre todos os elementos da lista e o elemento procurado foi identificado como sendo o único que teve a sua amplitude de probabilidade alterada. No entanto, não adiantaria realizar uma medição nos *qubits*, pois a informação da amplitude de probabilidade só está disponível em seu estado quântico (GRUSKA, 2000).

Com o elemento a ser buscado já identificado em seu estado quântico, as próximas aplicação de G tem o objetivo de aumentar a probabilidade de este elemento ser obtido após a operação de medida. É por este motivo que o algoritmo de *Grover* realiza um certo número de iterações, tentando aumentar a probabilidade de obter o resultado correto da operação.

2.4 Linguagens de Programação Quânticas

As leis da mecânica quântica nos apresentam alguns fenômenos, como a superposição e o emaranhamento, que permitem obter ganhos computacionais. Deste modo, geralmente, o processamento principal dos algoritmos quânticos é codificado de modo a explorar estas características especiais.

A física quântica apresenta alguns fenômenos, como superposição e emaranhamento, onde encontra-se o seu poder computacional, sendo utilizados no núcleo de algoritmos quânticos. Porém, estas características especiais nem sempre são intuitivas, o que dificulta a compreensão e concepção de novos algoritmos quânticos. Cada novo algoritmo utiliza-se de uma série única de transformações quânticas para atingir o seu objetivo, pois ainda não existe um princípio pelos quais os mesmos são desenvolvidos (SELINGER, 2004a).

Para entender melhor como a computação quântica funciona e como pensar formalmente sobre algoritmos quânticos, há um esforço dos pesquisadores na investigação de modelos semânticos e linguagens de programação para computação quântica. Estes trabalhos tem foco nas necessidades dos pesquisadores, pois ainda é necessário uma maneira de compreender a criação, análise, modelagem e simulação de algoritmos quânticos de alto nível.

Dois principais grupos de linguagens de programação tem sido desenvolvidas, as imperativas e as funcionais.

2.4.1 Linguagens Imperativas

De acordo com Selinger (SELINGER, 2004b), as primeiras propostas de linguagens quânticas seguiram o paradigma imperativo, sendo iniciado por Knill (KNILL, 1996), que criou uma série de convenções para escrever algoritmos quânticos em pseudo-código. Após este trabalho, linguagens imperativas mais completas foram definidas por Ömer (ÖMER, 1998), Sanders e Zuliani (SANDERS; ZULIANI, 1999) e Bettelli et al. (BETTELLI; SERAFINI; CALARCO, 2001), onde um programa é visto como uma sequência de operações que atualizam algum estado global.

Essas propostas são linguagens de programação completas ou extensões de linguagens imperativas já existentes na computação clássica, oferecendo uma série de características avançadas para manipular informação quântica. Como é típico em linguagens neste paradigma, a maioria não oferece uma descrição formal da semântica e também não permitem uma checagem completa de tipos em tempo de compilação.

2.4.2 Linguagens Funcionais

No campo de linguagens funcionais, programas não trabalham atualizando um estado global. Basicamente mapeiam entradas específicas para saídas, deste modo, possibilitando um melhor tratamento de erros em tempo de compilação, uma vez que, em geral, as linguagens

funcionais não permitem efeitos colaterais.

Neste formato de linguagens, Selinger foi um pioneiro na definição de linguagens funcionais, introduzindo as linguagens QFC e QPL (SELINGER, 2004a), que utilizam este paradigma e permitem a construção de algoritmos quânticos através de fluxogramas ou código-fonte.

Pode-se citar também o trabalho de Altenkirch e Grattage, que introduziram uma linguagem de programação funcional para computações quânticas puras (ALTENKIRCH; J, 2005). Van Tonder (TONDER, 2004) apresentou um cálculo-lambda para computações quânticas puras com um sistema de tipos baseados na lógica linear. Arrighi e Dowek (Arrighi; Dowek, 2006) definiram um cálculo lambda algébrico linear, que também pode ser visto como uma linguagem de programação quântica pura. Além destes, também existem uma série de trabalhos sobre o modelo categórico para computação quântica (ABRAMSKY, 2004), (COECKE; DUNCAN, 2008), (SELINGER, 2007).

O trabalho de Vizzotto (VIZZOTTO; ALTENKIRCH; SABRY, 2006) inspirou a abordagem deste projeto, através do uso de mônadas para modelar efeitos quânticos. Além de ter uma versão do cálculo lambda com efeitos e um framework tradicional para computação quântica, também foi proposto o uso de setas (*arrows*) (LINDLEY; WADLER; YALLOP, 2008) como uma linguagem de programação quântica para computações mistas (VIZZOTTO; DUBOIS; SABRY, 2009). Mais recentemente (VIZZOTTO; CALEGARO; PIVETA, 2013) foi apresentada uma versão do cálculo lambda com tipos simples, onde pode-se representar ambos os estados de computação quântica puros e impuros, que compreende uma camada de setas quânticas definidas sobre uma camada quântica monádica.

3 FEATHERWEIGHT JAVA

O Featherweight Java é uma proposta *core* mínima para a linguagem Java, proposto por Igarashi, Pierce e Wadler (IGARASHI; PIERCE; WADLER, 2001) para o estudo formal das definições do Java com classes parametrizadas. É dito um *core* mínimo no sentido de que são omitidos o máximo possível de características do Java, mantendo a forma de escrita de código e possibilitando a modelagem de seu sistema de tipos. Apesar disso, este fragmento é grande o suficiente para incluir vários programas, com o objetivo de prover uma maneira fácil de aplicar provas e estudar as consequências de extensões e variações em regras semânticas.

É possível utilizar o FJ como um ponto de partida para modelar linguagens que estendem o Java. Pelo fato dele ser compacto, é possível manter o foco em aspectos essenciais da extensão, motivando seu uso para a descrição da linguagem *FJQuantum*.

3.1 Características

O FJ apresenta características similares a linguagem Java, como o *Cálculo Lambda* tem em relação ao *Haskell*. Ela oferece as principais operações da linguagem, provendo classes, métodos, atributos, herança e *casts* dinâmicos com uma semântica muito próxima da linguagem original (IGARASHI; PIERCE; WADLER, 2001).

A ideia do Featherweight Java é propor uma visão puramente funcional da linguagem sem efeitos colaterais, onde operações sobre classes não afetam seus atributos internos em memória. Este projeto mínimo de linguagem provê originalmente apenas cinco maneiras para criação de termos, sendo:

- Criação de objetos.
- Invocação de métodos.
- Acesso a atributos.
- Conversão de tipos (*casts*).
- Variáveis.

Deste modo, muitos mecanismos da linguagem Java deixam de ser tratados, podendo ser citados: atribuição, interfaces, sobrecarga, ponteiros nulos, tipos base, declaração de métodos abstratos, controle de acesso e excessões. Características avançadas como concorrência,

inner classes e *reflection* também são omitidas (PIERCE, 2002). Porém, nas características que são modeladas nesta abordagem, é possível trabalhar com definições de classes mutuamente recursivas, sobrescrita de métodos, recursão de métodos com o uso de *this*, subtipos, etc.

Um programa escrito em Featherweight Java consiste de um conjunto de classes e um termo a ser avaliado. O seguinte trecho de código mostra algumas classes construídas utilizando esta proposta (IGARASHI; PIERCE; WADLER, 2001).

Código-fonte 3.1 – Exemplo de código em Featherweight Java

```
1  class A extends Object {
2      A() { super(); }
3  }
4  class B extends Object {
5      B() { super(); }
6  }
7  class Pair extends Object {
8      Object fst;
9      Object snd;
10     Pair(Object fst, Object snd) {
11         super();
12         this.fst=fst;
13         this.snd=snd;
14     }
15     Pair setfst(Object newfst) {
16         return new Pair(newfst, this.snd);
17     }
18 }
```

O exemplo acima é o mais comum, onde são definidas as classes *A*, *B* e *Pair*. Como pode-se perceber, uma superclasse é sempre incluída (mesmo sendo *Object*), sempre é escrito um construtor, mesmo que a classe não tenha atributos, e sempre é usado *this* para acessar atributos dentro de uma classe. Os construtores são sempre da mesma forma, com um parâmetro para cada atributo, com o mesmo nome, o construtor *super* é sempre invocado para inicializar os atributos da superclasse. O construtor é o único lugar onde aparecem os símbolos *super* e =.

Como mencionado, o FJ não suporta o comportamento imperativo, não sendo aplicáveis efeitos colaterais. Deste modo, é impossível escrever um método *set* para modificar o conteúdo de um atributo do objeto. No exemplo acima, pode-se notar uma forma de criar um comportamento similar, entretanto, é apenas retornado uma nova instância do objeto com o primeiro campo modificado. Assim, a classe *Pair* e qualquer outra nesta linguagem é imutável.

A abordagem proposta não oferece métodos estáticos e modificadores de acesso, portanto, o método *public static void main*, utilizado na linguagem Java, é representado pelo termo descrito após as definições de classes. Estas possíveis construções para termos, são as mesmas passíveis de uso no corpo de métodos. Abaixo é possível ver um exemplo de termo (PIERCE,

2002).

Código-fonte 3.2 – Exemplo de termo em Featherweight Java

```
1 new Pair(new A(), new B()).setfst(new B());
```

No contexto do termo e das classes previamente apresentadas, a expressão é avaliada, utilizando as regras de substituição apresentadas nas próximas seções, resultando em:

Código-fonte 3.3 – Exemplo de termo após a avaliação

```
1 new Pair(new B(), new B())
```

Este resultado é obtido, mantendo a abordagem funcional, por substituir *new B()* por *newfst* e *new Pair(new A(), new B())* por *this* no corpo do método *setfst*. Depois disso, *this.snd* é substituído por *new B()* no termo *new Pair(newfst, this.snd)*, apresentando como resultado da avaliação *new Pair(new B(),new B())*.

Na sequência serão apresentadas as definições sintáticas, de avaliação semântica e do sistema de tipos da linguagem. Essas definições podem ser encontradas sumarizadas no Apêndice A.

3.2 Sintaxe

Para iniciar a definição formal do Featherweight Java, na Figura 3.1 é apresentada a sintaxe abstrata da linguagem, contendo a gramática *Backus-Naur-Form* (*BNF*), onde são apresentadas as declarações de classes (*CL*), construtores (*K*), métodos (*M*) e expressões ou termos (*t*).

Para a sintaxe é assumida a existência da variável especial *this*, e ela nunca é usada fora do contexto de uma classe ou como nome de parâmetro de um método. A regra de avaliação semântica se encarrega de fazer as apropriadas substituições para este caso, e será apresentada posteriormente.

Como convenção adotou-se $\bar{\mathtt{f}}$ para designar uma lista de itens $\mathtt{f}_1, \ldots, \mathtt{f}_n$ (similarmente para $\bar{\mathtt{C}}$, $\bar{\mathtt{x}}$, $\bar{\mathtt{t}}$, etc.) em todo o projeto. Além disso, é assumido que sequências de declaração de campos, nomes de parâmetros, e declarações de métodos não contém duplicados.

A partir dessas definições, um programa escrito nesta linguagem pode ser visto como um par (*CT*,*t*), sendo uma tabela de classes (*CT*), e um termo (*t*). A tabela de classes é uma sequência contendo o nome da classe (*C*) e a declaração da mesma no formato sintático já apresentado (*CL*). Como percebido, toda classe tem uma superclasse, declarada com a palavra-chave *extends*. No caso especial da classe *Object*, que não tem superclasse, é tratado como um nome de classe reservado e este não aparece na tabela de classes. (PIERCE, 2002). Nesta

Figura 3.1 – Definições Sintáticas do Featherweight Java.

Definição de Tipos
```
T   ::= TFJ
TFJ ::= C
```
Declarações de classes
```
CL ::= class C extends C { C̄ f̄; K M̄ }
```
Declarações de construtores
```
K   ::= C(C̄ f̄) { super(f̄); this.f̄=f̄; }
```
Declarações de métodos
```
M   ::= C m(C̄ x̄) { return t; }
```
Termos
```
t   ::= x
      | t.f
      | t.m(t̄)
      | new C(t̄)
      | (C) t
```
Valores
```
v   ::= new C(v̄)
```

proposta, diferentemente da linguagem Java, os métodos de *Object* são ignorados.

3.3 Sistema de Tipos

A proposta original FJ (IGARASHI; PIERCE; WADLER, 2001) prevê a implementação de uma linguagem fortemente tipada, com um sistema de tipos bem definido. Suas regras de tipo são direcionadas à sintaxe, com uma regra para cada forma de expressão, exceto para casts, onde existem três regras. Muitas delas são simples adaptações das regras da linguagem Java. As definições de inferência verificam a corretude da tabela de classes e dos termos, ou seja, antes de executar a avaliação do código fonte, primeiro é verificado se a tabela de classes está bem definida e com sentido. Após isto, os termos a serem avaliados são checados, verificando se estão declarados de acordo com a tabela de classes, e se as definições de tipos estão corretas. Somente após isto o termo é avaliado.

As linguagens orientadas a objetos trabalham com o mecanismo de herança, que cria uma relação de *subtipo* entre as classes, conforme demonstrado na Figura 3.2. A primeira regra demonstra que cada classe é *subtipo* de si mesma. Na segunda regra, se a classe *C* é subtipo da classe *D*, que é subtipo de *E*, então, *C* também é subtipo de *E*. Na última regra é possível perceber a relação de herança, a qual é definida na sintaxe da linguagem, através da palavra chave *extends*.

Figura 3.2 – Regras de subtipo para as classes.

$$\frac{}{\text{C} <: \text{C}}$$

$$\frac{\text{C} <: \text{D} \quad \text{D} <: \text{E}}{\text{C} <: \text{E}}$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{...\}}{\text{C} <: \text{D}}$$

A seguir serão apresentadas as definições para a correta descrição de classes e termos.

3.3.1 Formação da Tabela de Classes

O primeiro processo realizado pelo sistema de tipos, é a verificação de sanidade da tabela de classes. Neste processo é verificado se cada classe contida na tabela de classes está bem declarada. De acordo com as regras apresentadas na Figura 3.3, para uma classe ser considerada bem formada, é necessário que o construtor aplique os primeiros parâmetros para *super* (em caso de herança), e os subsequentes para inicializar os atributos internos da classe. Além disso, todos os métodos da classe devem estar corretamente declarados.

Utiliza-se a função auxiliar *fields*, também definida na Figura 3.3, para obter todos os atributos da classe e possibilitar a sua checagem. Pode-se observar que a classe *Object* não apresenta nenhum atributo ($\bullet$), e as demais, resultam nos atributos da própria classe (C), mais os atributos de sua classe base (D).

Finalmente, para verificar se um método está corretamente declarado, é necessário que cada um de seus parâmetros sejam de tipos válidos na linguagem, bem como o resultado do processamento do termo descrito no corpo do método realmente seja avaliado para o tipo esperado na definição do retorno do método.

3.3.2 Tipagem dos Termos

Após a completa verificação da tabela de classes, inicia-se o processo de checagem dos tipos presentes no termo a ser avaliado. Neste processo, utiliza-se uma estrutura auxiliar, chamada de *contexto* Γ, que tem a responsabilidade de mapear as variáveis a seus tipos. Este contexto é preenchido dinamicamente de acordo com as definições presentes nos termos e na tabela de classes.

Figura 3.3 – Regras para checagem da tabela de classes.

Obtenção de atributos de uma classe

$$\textit{fields}(\text{Object}) = \bullet$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\bar{\text{C}}\ \bar{\text{f}};\ \text{K}\ \bar{\text{M}}\} \quad \textit{fields}(\text{D}) = \bar{\text{D}}\ \bar{\text{g}}}{\textit{fields}(\text{C}) = \bar{\text{D}}\ \bar{\text{g}}, \bar{\text{C}}\ \bar{\text{f}}}$$

Regras para classes bem definidas

$$\frac{\text{K} = \text{C}(\bar{\text{D}}\ \bar{\text{g}}, \bar{\text{C}}\ \bar{\text{f}})\ \{\ \text{super}(\bar{\text{g}});\ \text{this}.\bar{\text{f}} = \bar{\text{f}};\ \} \quad \textit{fields}(\text{D}) = \bar{\text{D}}\ \bar{\text{g}} \quad \bar{\text{M}}\ \text{OK em C}}{\text{class C extends D } \{\ \bar{\text{C}}\ \bar{\text{f}};\ \text{K}\ \bar{\text{M}}\ \}\ \ \text{OK}}$$

Regras para métodos bem definidos:

$$\frac{\bar{\text{x}}\text{: } \bar{\text{C}}, \text{this: C} \vdash \text{t}_0\text{: E}_0 \quad \text{E}_0 <: \text{C}_0 \quad CT(\text{C}) = \text{class C extends D } \{\ \ldots\ \} \quad \textit{override}(\text{m, D}, \bar{\text{C}} \to \text{C}_0)}{\text{C}_0\ \text{m}(\bar{\text{C}}\ \bar{\text{x}})\ \{\ \text{return t}_0;\ \}\ \text{OK em C}}$$

As regras e definições para tipagem dos termos e métodos são apresentadas na Figura 3.4. A primeira parte corresponde a obtenção de tipos de métodos, utilizada nas regras posteriores. Na sequência, inicia-se a apresentação da tipagem dos termos.

Para obter o tipo de um método de uma classe, também foi proposta uma função auxiliar, chamada *mtype*, que tem por objetivo obter, a partir da sintaxe, o tipo dos parâmetros esperados e o tipo de retorno do método. O primeiro passo apresentado nas regras verifica se o método está contido na classe alvo C, e caso não esteja, é executado a chamada *mtype* na classe base D. Após ser encontrada a definição sintática do método, o retorno é provido como uma lista de tipos dos parâmetros $\bar{B}$ mais o tipo do retorno do método B.

A primeira regra referente a tipagem dos termos, trata dos tipos de variáveis, verificando a existência de sua definição no contexto, ou seja, a variável *x* é do tipo definido no contexto Γ. O contexto Γ pode ser visto como o *escopo* do bloco sendo avaliado. As próximas regras apresentam respectivamente os tipos para acesso a atributos, invocação de métodos e criação de objetos.

Para obter o tipo de um termo que refere-se ao acesso a um atributo de uma classe, leva-se em consideração o tipo deste atributo presente na classe alvo. Primeiramente, obtêm-se o tipo do termo t_0, obtendo o nome da classe alvo, para posteriormente verificar se realmente existe um atributo nesta classe com o nome utilizado no termo, para assim retornar o seu tipo.

Figura 3.4 – Regras de tipos para termos e métodos.

Obtenção do tipo de um método

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad \mathrm{B}\ \mathrm{m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}}{mtype(\mathrm{m}, \mathrm{C}) = \bar{\mathrm{B}} \to \mathrm{B}}$$

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad m \text{ is not defined in } \bar{\mathrm{M}}}{mtype(\mathrm{m}, \mathrm{C}) = mtype(\mathrm{m}, \mathrm{D})}$$

Tipagem dos termos

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash \mathrm{t}_0 : \mathrm{C}_0 \quad fields(\mathrm{C}_0) = \bar{\mathrm{C}}\ \bar{\mathrm{f}}}{\Gamma \vdash \mathrm{t}_0.\mathrm{f}_i : \mathrm{C}_i}$$

$$\frac{mtype(\mathrm{m}, \mathrm{C}_0) = \bar{\mathrm{D}} \to \mathrm{C} \quad \Gamma \vdash \bar{\mathrm{t}} : \bar{\mathrm{C}} \quad \bar{\mathrm{C}} <: \bar{\mathrm{D}}}{\Gamma \vdash \mathrm{t}_0.\mathrm{m}(\bar{\mathrm{t}}) : \mathrm{C}}$$

$$\frac{fields(\mathrm{C}) = \bar{\mathrm{D}}\ \bar{\mathrm{f}} \quad \Gamma \vdash \bar{\mathrm{t}} : \bar{\mathrm{C}} \quad \bar{\mathrm{C}} <: \bar{\mathrm{D}}}{\Gamma \vdash \text{new } \mathrm{C}(\bar{\mathrm{t}}) : \mathrm{C}}$$

A obtenção do tipo de um termo onde é realizada a invocação de um método, realiza o processo de identificação da classe que contém este método, para posteriormente buscar o tipo dos parâmetros e o tipo de retorno do método, através da função auxiliar *mtype*, descrita anteriormente, checando se os parâmetros passados na invocação (parâmetros atuais) são de tipos correspondentes aos esperados (parâmetros formais). Caso estes itens estejam em conformidade, é retornado o tipo de retorno do método.

Já na regra de construção de objetos, são verificados todos os parâmetros passados para o construtor, verificando se realmente eles existem na definição da classe, através da função *fields*, e verificando se cada um dos atributos foi recebido de acordo com o seu tipo. Caso estejam definidos corretamente, é retornado o tipo (nome da classe) do objeto recém criado.

Além das regras de tipos já apresentadas, existem ainda três regras para *casts* (conversão de tipos), sendo respectivamente uma regra para *upcast*, onde o termo t_0 é uma subclasse do tipo alvo, uma para *downcast*, onde o tipo alvo é uma subclasse do termo, e uma regra, que é uma das inovações apresentadas no Featherweight Java, chamada pelos autores de *stupid casts*, onde os tipos a serem convertidos não são relacionados.

Figura 3.5 – Regras de tipos para *casts*.

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)\ t_0 : C}$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)\ t_0 : C}$$

$$\frac{\Gamma \vdash t_0 : D \quad C \not<: D \quad D \not< C}{\Gamma \vdash (C)\ t_0 : C}$$

É importante notar que é possível carregar a tabela de classes e usá-la para verificar os tipos de todas as classes antes de avaliar o código fonte do programa. Esta característica similar ao Java permite a manutenção da segurança da linguagem (*type safety*) (PIERCE, 2002).

3.4 Avaliação de Termos

Após realizar o processo de checagem dos tipos e consistência da tabela de classes, se faz necessário processar o termo que utiliza as classes previamente declaradas. Como citado anteriormente, o método estático *main* do Java, é substituído pelo termo declarado após as definições de classes no Featherweight Java.

Este é o último passo executado na interpretação do código-fonte, e portanto, nesta etapa o código fonte está sintaticamente correto, as classes estão bem definidas, e os tipos dos termos utilizados condizem com os previamente declarados em classes.

3.4.1 Regras de Avaliação

A proposta original do FJ prevê apenas três regras de avaliação (ou redução), sendo uma para acesso aos atributos de uma classe, uma para invocação de métodos e outra para tratamentos dos *casts*.

A primeira regra diz respeito ao acesso a um atributos de uma classes, de forma a apresentar o valor que está associado a ele. Como pode ser visto na Figura 3.6, primeiramente utiliza-se a função auxiliar *fields*, já apresentada na Figura 3.3, que retorna uma lista contendo todos os atributos declarados na classe informada ($\bar{C}\,\bar{f}$). Após isto, busca-se nesta lista a posição i do elemento f que está sendo solicitado. Por fim, o valor v repassado ao construtor respectivo a esta posição i é retornado.

Figura 3.6 – Regra de avaliação para acesso a um atributo.

$$\frac{\mathit{fields}(\mathrm{C}) = \bar{\mathrm{C}}\ \bar{\mathrm{f}}}{\text{new } C(\bar{v}).\mathrm{f}_i \rightarrow v_i}$$

A segunda regra trata da avaliação do acesso a métodos, buscando na tabela de classes o método respectivo a classe informada, retornando os parâmetros esperados e o termo associado. A primeira parte da figura 3.7 apresenta as definições auxiliares para a obtenção da definição do método m de uma classe C através da função *mbody*. As informações do método são obtidas através da verificação da classe informada ou de sua classe base.

Na sequencia, é apresentada a regra de avaliação de acesso ao método, que utiliza-se da função *mbody*, fazendo a substituição das ocorrências dos parâmetros recebidos no método ($\bar{\mathrm{u}}$) e das variáveis membro da classe (*this*) no termo t_0.

Figura 3.7 – Regras auxiliares e de avaliação do acesso a métodos.

Obtenção do corpo de um método

$$\frac{\begin{array}{c}\mathit{CT}(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \\ \mathrm{B\ m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}\end{array}}{\mathit{mbody}(\mathrm{m},\mathrm{C}) = (\bar{\mathrm{x}},\ \mathrm{t})}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \\ m \text{ não está definido em } \bar{\mathrm{M}}\end{array}}{\mathit{mbody}(\mathrm{m},\mathrm{C}) = \mathit{mbody}(\mathrm{m},\ \mathrm{D})}$$

Avaliação de acesso a um método

$$\frac{\mathit{mbody}(\mathrm{m},\ \mathrm{C}) = (\bar{\mathrm{x}},\ \mathrm{t}_0)}{\text{new } C(\bar{v}).\mathrm{m}(\bar{\mathrm{u}}) \rightarrow [\bar{\mathrm{x}} \mapsto \bar{\mathrm{u}},\ \mathrm{this} \mapsto \text{new C}(\bar{\mathrm{v}})]\mathrm{t}_0}$$

A última regra de avaliação trata dos *casts*, onde é verificado se a conversão de tipos é válida no contexto da tabela de classes. A figura 3.8 mostra que para a conversão de tipos ser válida, a classe a ser convertida deve ter uma relação de subtipo com a classe alvo.

Figura 3.8 – Regras de avaliação para *casts*.

$$\frac{\mathrm{C} <: \mathrm{D}}{(\mathrm{D})\ (\text{new C}(\bar{\mathrm{v}})) \rightarrow \text{new C}(\bar{\mathrm{v}})}$$

Além das definições auxiliares para obtenção de atributos e métodos das classes, e das regras para computação dos termos, na Figura 3.9 também são propostas as regras de con-

gruência para todas as regras de computação, que tratam da avaliação dos termos até que eles tornem-se um valor.

Figura 3.9 – Regras de congruência para avaliação dos termos.

Congruência para acesso a atributos

$$\frac{t_0 \to t'_0}{t_0.f \to t'_0.f}$$

Congruência para acesso a métodos:

$$\frac{t_0 \to t'_0}{t_0.m(\bar{t}) \to t'_0.m(\bar{t})}$$

$$\frac{t_i \to t'_i}{v_0.m(\bar{v}, t_i, \bar{t}) \to v_0.m(\bar{v}, t'_i, \bar{t})}$$

Congruência para construtores

$$\frac{t_i \to t'_i}{\text{new } C(\bar{v}, t_i, \bar{t}) \to \text{new } C(\bar{v}, t'_i, \bar{t})}$$

Congruência para casts

$$\frac{t_0 \to t'_0}{(C)\ t_0 \to (C)\ t'_0}$$

Na proposta original do FJ optou-se por trabalhar com uma relação de redução não determinística, similar a relação de redução *full beta-reduction* do *Cálculo Lambda*. Cada regra de congruência demonstra a ordem em que os termos são avaliados, e apresentam as definições semânticas que fornecem o comportamento similar a linguagem Java original.

3.5 Tipos Genéricos

Os mesmos autores da proposta original do FJ também definiram uma semântica para os tipos genéricos, similares a implementação que a linguagem Java possui (IGARASHI; PIERCE; WADLER, 2001). O trecho de código abaixo apresenta uma classe para tratar pares similar ao exemplo da seção anterior, agora reescrito utilizando tipos genéricos.

```
1  class Pair<X extends Object, Y extends Object> extends Object {
2      X fst;
3      Y snd;
4      Pair(X fst, Y snd) {
5          super(); this.fst=fst; this.snd=snd;
6      }
7      <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
8          return new Pair<Z,Y>(newfst, this.snd);
9      }
10 }
```

Neste exemplo, podemos perceber que tanto classes como métodos podem receber tipos genéricos. A classe *Pair* possui os parâmetros X e Y e o método *setfst* possui o parâmetro Z. Cada tipo genérico possui um limitador, onde no exemplo, X, Y e Z são limitados pelo tipo *Object*.

No contexto das definições acima, podemos escrever:

```
1  new Pair<A,B>(new A(), new B())
```

Neste caso, o atributo *fst* foi instanciado com o tipo A, e o atributo *snd* com o tipo B.

A Figura 3.10 mostra as construções adicionadas na sintaxe da linguagem neste projeto. Estas apresentam a definição de um novo tipo para a linguagem $\mathrm{T_G}$, representando a criação de uma classe C com tipos os genéricos $\bar{\mathrm{T}}$. Além disso, a definição sintática para `CL` permite a adição das informações para lidar com os tipos genéricos na definição da classe. Também foi adicionado um termo que permita a passagem dos tipos genéricos na instanciação destas classes. Como opção de projeto e para simplificar a implementação do interpretador, não foram adicionadas regras para tratamento de métodos genéricos, porém essas podem ser encontradas na proposta original.

Figura 3.10 – Definições sintáticas para os tipos genéricos.

Tipos

$\mathtt{T} \ ::= \ \mathtt{T_{FJ}} \mid \mathtt{T_G}$

$\mathtt{T_G} \ ::= \ \mathtt{C}\langle\bar{\mathtt{T}}\rangle$

Declarações de classes

$\mathtt{CL} \ ::= \ \mathtt{class\ C}\langle\bar{\mathtt{C}}\ \mathtt{extends}\ \bar{\mathtt{C}}\rangle\ \mathtt{extends\ C\ \{\ }\bar{\mathtt{C}}\ \bar{\mathtt{f}}\mathtt{;\ K\ }\bar{\mathtt{M}}\ \mathtt{\}}$

Sintaxe

$\mathtt{t} \ ::= \ \ldots \ \triangleright \text{ Outros termos da linguagem}$

$\quad \mid \mathtt{new\ T_G}(\bar{\mathtt{t}})$

Ao utilizar tipos genéricos na linguagem, são necessárias algumas alterações nas definições auxiliares do FJ, de modo a permitir a correta manipulação e obtenção de informações da tabela de classes, conforme pode ser visto na Figura 3.11.

Figura 3.11 – Definições auxiliares para os tipos genéricos.

Obtenção de atributos de uma classe

$$\frac{CT(\mathrm{C}) = \mathrm{class\ C}\langle \bar{\mathrm{X}} \lhd \bar{\mathrm{N}} \rangle \mathrm{\ extends\ D\ } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \qquad \mathit{fields}([\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \mathrm{D}) = \bar{\mathrm{U}}\ \bar{\mathrm{g}}}{\mathit{fields}(\mathrm{C}\langle \bar{\mathrm{T}} \rangle) = \bar{\mathrm{U}}\ \bar{\mathrm{g}},\ [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \bar{\mathrm{C}}\ \bar{\mathrm{f}}}$$

Obtenção do tipo dos métodos de uma classe

$$\frac{CT(\mathrm{C}) = \mathrm{class\ C}\langle \bar{\mathrm{X}} \lhd \bar{\mathrm{N}} \rangle \mathrm{\ extends\ D\ } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \qquad \mathrm{B\ m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \mathrm{return\ t;}\ \} \in \bar{\mathrm{M}}}{\mathit{mtype}(\mathrm{m}, \mathrm{C}\langle \bar{\mathrm{T}} \rangle) = [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ (\bar{\mathrm{B}} \to \mathrm{B})}$$

Obtenção do corpo dos métodos de uma classe

$$\frac{CT(\mathrm{C}) = \mathrm{class\ C}\langle \bar{\mathrm{X}} \lhd \bar{\mathrm{N}} \rangle \mathrm{\ extends\ D\ } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \qquad \mathrm{B\ m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \mathrm{return\ t;}\ \} \in \bar{\mathrm{M}}}{\mathit{mbody}(\mathrm{m},\mathrm{C}\langle \bar{\mathrm{T}} \rangle) = (\bar{\mathrm{x}},\ [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \mathrm{t})}$$

As alterações nas regras auxiliares originais demonstram que no detalhamento da obtenção dos atributos de uma classe, o tipo definido na instanciação da classe genérica $\bar{\mathrm{T}}$ deve ser substituído pelas ocorrências de $\bar{\mathrm{X}}$ nos atributos obtidos da classe. Para o caso da manipulação dos tipos de métodos, também são substituídas as ocorrências de $\bar{\mathrm{X}}$ por $\bar{\mathrm{T}}$ no retorno da função *mtype*. O mesmo ocorre na obtenção das informações do corpo do método, neste caso, substituindo as ocorrências de $\bar{\mathrm{X}}$ por $\bar{\mathrm{T}}$ no termo t.

4 A LINGUAGEM FJQUANTUM

Este capítulo descreve formalmente os recursos adicionados ao Featherweight Java, de modo a torná-lo uma linguagem capaz de manipular dados e operações quânticas. Esta linguagem utiliza-se do paradigma da orientação a objetos, seguindo as premissas da proposta original, projetada para integrar uma camada monádica que encapsule o processamento de informações quânticas. Os principais objetivos a serem alcançados pelo projeto da linguagem são:

- Definir uma semântica operacional para uma extensão ao FJ, que possibilite a utilização de mônadas, e consequentemente, permita implementar a mônada quântica.
- Oferecer uma alternativa aos programadores que já utilizam a linguagem Java, mantendo os conceitos do paradigma OO, facilitando a fluência dos futuros usuários da linguagem proposta.
- Possibilitar a definição de operadores quânticos na própria linguagem, através da utilização de *closures*, sem necessitar alterações na especificação sintática ou semântica.

As seções seguintes apresentam diversos recursos da linguagem proposta e uma série de exemplos ilustrando a sua utilização. No decorrer do capítulo serão definidas e explanadas sequencialmente as alterações na sintaxe e semântica da linguagem, fornecendo as construções necessárias para, ao final, apresentar a semântica monádica que encapsula as definições quânticas. O Apêndice A apresenta a formalização completa da linguagem desenvolvida neste trabalho.

4.1 Tipos Primitivos e Operações

Como já explanado anteriormente, o FJ não possui tipos primitivos, e apresenta apenas algumas regras para computação de termos. Para permitir a simulação da computação quântica na linguagem, é necessário que a mesma seja capaz de manipular alguns tipos primitivos, como *booleanos* e *números complexos*. Além disso, também há a necessidade de realizar algumas operações sobre estes tipos.

A Figura 4.1 apresenta as construções adicionadas na sintaxe da linguagem. Para tratar dos booleanos, foram adicionadas as palavras-chave que representam os estados *verdadeiro* (*true*) e *falso* (*false*) e também um termos para tratamento das condicionais (*if*). Os valores

booleanos também são modelados como um tipo primitivo da linguagem (PIERCE, 2002). Já para o caso dos números complexos, como uma maneira de simplificar a especificação, apenas adicionou-se algumas palavras chave, representando as constantes necessárias para trabalhar com a computação quântica, onde *ComplexZero* e *ComplexOne* representam os números 0 e 1, *ComplexHalf* e *ComplexMHalf*, representam o valor $1/\sqrt{2}$ e $-1/\sqrt{2}$. Os operadores matemáticos atuam sobre estes números complexos, tendo a possibilidade de realizar a *adição*, *subtração* e *multiplicação* sobre os mesmos. Além disso, como uma forma de melhorar a leitura e escrita de código, também foi adicionado o operador *let*, facilitando a manutenção de expressões complexas.

Figura 4.1 – Definições sintáticas para os tipos primitivos.

Tipos

```
T  ::= TFJ | TG | TP
TP ::= boolean | Complex
```

Sintaxe

```
t  ::= ...  ▷ Outros termos da linguagem
     | true
     | false
     | if ( t ) { t } else { t }
     | ComplexOne
     | ComplexZero
     | ComplexHalf
     | ComplexMHalf
     | t + t
     | t - t
     | t * t
     | let x = t in t
```

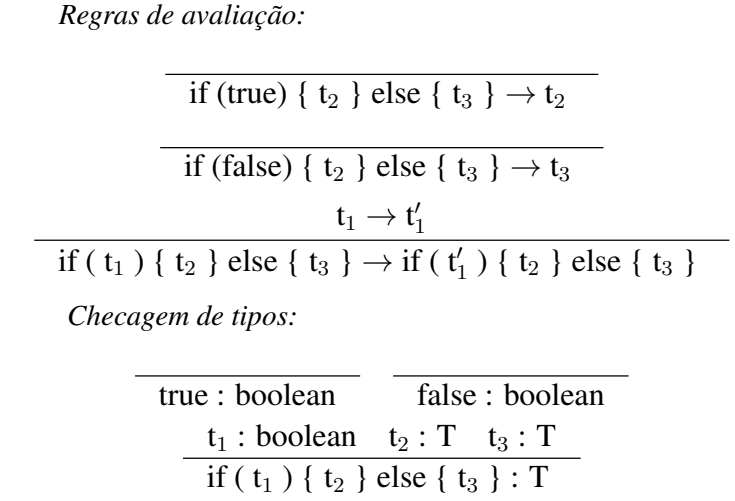
Valores

```
v  ::= ...  ▷ Outros valores da linguagem
     | true
     | false
     | ComplexOne
     | ComplexZero
     | ComplexHalf
     | ComplexMHalf
```

A Figura 4.2 apresenta as regras para avaliação e verificação de tipos para os tipos booleanos e expressões condicionais.

As duas primeiras regras de avaliação são axiomas sobre constantes do tipo *booleano*, avaliando cada uma das possibilidades de acordo com a definição no código-fonte. Se o pri-

Figura 4.2 – Regras de avaliação e checagem de tipos para booleanos e condicionais.



meiro termo for *true*, retorna-se t_2, caso contrário, retorna-se t_3. A terceira regra representa a congruência do termo, avaliando t_1 para t'_1, em um passo de redução, caso este não seja um valor.

Na verificação de tipos, caso sejam processadas as constantes *booleanas*, é retornado o tipo *boolean*. No caso de processamento de uma expressão condicional, é verificado se t_1 resulta em um termo booleano, e t_2 e t_3 são do mesmo tipo. Como o FJ é uma versão funcional de Java, uma condicional sempre deve retornar um valor, e as duas possibilidades devem ter o mesmo tipo de retorno.

Já a Figura 4.3 apresenta as regras para avaliação e verificação de tipos para os números complexos e operadores matemáticos.

As regras de redução para os operadores matemáticos demonstram a ordem de avaliação, começando da esquerda para a direita, até que os termos tornem-se valores. As regras de tipo consideram as constantes definidas na sintaxe, e os tipos esperados para as operações matemáticas.

A Figura 4.4 apresenta as regras de avaliação e de checagem de tipos para o operador *let*.

As regras de avaliação indicam o funcionamento da substituição do primeiro termo v_1 no termo t_2, caso este seja um valor. Se o primeiro termo não estiver na forma de um valor, este é avaliado primeiro. No caso da checagem de tipos, deve-se adicionar o tipo de t_1 no contexto Γ durante o processamento de t_2, retornando como o tipo da expressão, o tipo do termo t_2.

Figura 4.3 – Regras de avaliação e checagem de tipos para números complexos e operadores.

Regras de avaliação:

$$\frac{t_1 \to t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2} \qquad \frac{t_2 \to t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}$$

$$\frac{t_1 \to t_1'}{t_1 - t_2 \longrightarrow t_1' - t_2} \qquad \frac{t_2 \to t_2'}{v_1 - t_2 \longrightarrow v_1 - t_2'}$$

$$\frac{t_1 \to t_1'}{t_1 * t_2 \longrightarrow t_1' * t_2} \qquad \frac{t_2 \to t_2'}{v_1 * t_2 \longrightarrow v_1 * t_2'}$$

Checagem de tipos:

$$\frac{}{\Gamma \vdash \text{ComplexZero, ComplexOne} : \mathbb{C}} \qquad \frac{}{\Gamma \vdash \text{ComplexHalf, ComplexMHalf} : \mathbb{C}}$$

$$\frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 + t_2} \qquad \frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 - t_2} \qquad \frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 * t_2}$$

Figura 4.4 – Regras de avaliação e de tipos para o operador *let*.

Regras de avaliação

$$\text{let } x = v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]\, t_2$$

$$\frac{t_1 \to t_1'}{\text{let } x = t_1 \text{ in } t_2 \longrightarrow \text{let } x = t_1' \text{ in } t_2}$$

Checagem de tipos

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

4.2 Tuplas

A maioria das linguagens de programação fornecem maneiras de construir estruturas de dados compostas. A própria orientação a objetos trabalha diretamente com a criação de novos tipos de dados compostos, através de classes e objetos. Nesta seção serão apresentados os aspectos semânticos para adição de tuplas embutidas na linguagem (PIERCE, 2002).

Adicionar tuplas na linguagem envolve duas novas formas de termos, a primeira para a descrição das tuplas e a segunda para acesso aos seus elementos. Além disso, as tuplas também são adicionadas como um novo tipo da linguagem, este tipo sendo descrito como a combinação do tipo de cada um de seus elementos. Por exemplo, `{1,2,true}` é uma tupla com três elementos, contendo dois números e um *booleano*. O tipo desta tupla poderia ser escrito como

`{int,int,boolean}`. A Figura 4.5 apresenta as novas definições sintáticas.

Para possibilitar a formalização das tuplas, é necessária a criação de notações para descrever uniformemente estruturas de tamanho arbitrário. Sendo assim, é utilizada a notação $\{\mathtt{t}_i{}^{i\in 1..n}\}$ para uma tupla de n elementos, $\mathtt{t}_1$ até $\mathtt{t}_n$, e $\{\mathtt{T}_i{}^{i\in 1..n}\}$ para os seus tipos.

Figura 4.5 – Definições sintáticas para tuplas.

Tipos

$\mathtt{T} \;::=\; \mathtt{T_{FJ}} \mid \mathtt{T_G} \mid \mathtt{T_P} \mid \mathtt{T_T}$

$\mathtt{T_T} \;::=\; \{\mathtt{T}_i{}^{i\in 1..n}\}$

Sintaxe

$\mathtt{t} \;::=\; \ldots \quad \triangleright \text{ Outros termos da linguagem}$

$\mid \{\mathtt{t}_i{}^{i\in 1..n}\}$

$\mid \mathtt{t.i}$

Valores

$\mathtt{v} \;::=\; \ldots \quad \triangleright \text{ Outros valores da linguagem}$

$\mid \{\mathtt{v}_i{}^{i\in 1..n}\}$

A Figura 4.6 formaliza as regras de avaliação, indicando como as tuplas e projeções (acesso aos seus elementos) devem se comportar. A primeira regra refere-se à projeção de um elemento da tupla, sendo j o número correspondente à posição do elemento desejado, iniciando em 1. A segunda regra refere-se à avaliação do termo antes de tentar realizar a sua projeção. Na última, apresenta-se a estratégia de avaliação *call-by-value*, avaliando-se cada posição da tupla antes de apresentá-la como um valor, sendo processada da esquerda para a direita.

Figura 4.6 – Regras de avaliação para tuplas.

$$\frac{}{\{\mathtt{v}_i{}^{i\in 1..n}\}.\mathtt{j} \rightarrow \mathtt{v}_j}$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathtt{t}_1.\mathtt{i} \rightarrow \mathtt{t}'_1.\mathtt{i}}$$

$$\frac{\mathtt{t}_j \rightarrow \mathtt{t}'_j}{\{\mathtt{v}_i{}^{i\in 1..j-1}, \mathtt{t}_j, \mathtt{t}_k{}^{k\in j+1..n}\} \rightarrow \{\mathtt{v}_i{}^{i\in 1..j-1}, \mathtt{t}'_j, \mathtt{t}_k{}^{k\in j+1..n}\}}$$

A Figura 4.7 apresenta primeiramente a regra na qual utiliza-se o contexto Γ para obter o tipo de cada elemento, resultado no tipo da tupla. A segunda regra trabalha sobre o resultado da regra anterior, buscando o tipo do elemento associado ao índice solicitado.

Nesta seção foi apresentada uma abordagem funcional da utilização de tuplas embutidas na linguagem Java, porém a mesma funcionalidade poderia ser implementada através do

Figura 4.7 – Regras de tipo para tuplas.

$$\frac{\text{para cada } i \quad \Gamma \vdash \mathrm{t}_i : \mathrm{T}_i}{\Gamma \vdash \{\mathrm{t}_i{}^{i \in 1..n}\} : \{\mathrm{T}_i{}^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \{\mathrm{T}_i{}^{i \in 1..n}\}}{\Gamma \vdash \mathrm{t}_1.\mathrm{j} : \mathrm{T}_j}$$

uso de uma classe encapsulando as características das tuplas, de maneira similar a classe *Pair* apresentada em exemplos nas seções anteriores.

4.3 Closures

No ano de 2014, foi liberada a versão 8 da linguagem Java, apresentando, dentre outras funcionalidades, as *expressões lambda* ou *closures*. Este novo recurso da linguagem, herdada do Cálculo Lambda, permite o tratamento de funções como argumentos a métodos, possibilitando realizar procedimentos complexos com um número reduzido de linhas de código. Além disso, esta nova funcionalidade permite explorar o paralelismo em nível de operações sobre coleções de objetos através da API de *Streams*.

Com o objetivo de estender o FJ, de modo a permitir o uso de *closures*, Bellia e Occhiuto (BELLIA; OCCHIUTO, 2010) propuseram as regras sintáticas, semânticas e de tipos que definem o comportamento e garantem a segurança da implementação de *closures* na linguagem. Esta extensão modela todas as características essenciais do FJ envolvidas nas propriedades das *closures*. Seguindo a abordagem proposta pelos autores, basicamente, para trabalhar com *expressões lambda*, é preciso estender o sistema de tipos da linguagem, adicionando um tipo que represente uma função, permitindo a passagem dos mesmos como parâmetros, podendo ser utilizados em métodos ou em outras expressões, além de implementar as regras de avaliação dessas funções anônimas.

A abordagem utilizada neste projeto difere da real implementação de *closures* na linguagem Java, pois nela não existe o tipo *função*, e *closures* somente podem ser usadas através das chamadas *interfaces funcionais*. Para simplificar a construção, foi escolhida a modelagem de expressões *lambda* como um tipo, sem necessitar a criação do comportamento completo de interfaces do Java.

Para conseguir fornecer tal funcionalidade foram adicionadas novas regras sintáticas na

linguagem. A Figura 4.8 apresenta as regras para *expressões lambda*, que consistem de uma lista de parâmetros e um termo que atua como o corpo da função anônima. Além disso, a *closure* é definida como um tipo na linguagem (T_F), o qual é representado pelo mapeamento dos tipos dos parâmetros mais o tipo de retorno da função $(\bar{T}) \rightarrow T$.

Figura 4.8 – Definições sintáticas para *closures*.

Tipos

```
T   ::= T_FJ | T_G | T_P | T_T | T_F
T_F ::= (T̄) -> T
```

Termos

```
t ::= ...          ▷ Termos Featherweight Java
    | (T̄ x̄) -> t
    | t.invoke(t̄)
```

Valores

```
v ::= ...          ▷ Valores Featherweight Java
    | (T̄ x̄) -> t
```

Com a adição das *closures* aos termos do FJ através da abordagem proposta, é possível escrever programas com *expressões lambda* no termo principal, nos parâmetros de métodos ou de outras *closures*, no corpo de métodos, etc. A Figura 4.9 apresenta as regras de avaliação para este tipo de expressão.

Figura 4.9 – Regras de avaliação para *closures*.

Regras de redução

$$((\bar{\mathrm{T}}\ \bar{\mathrm{x}}) \rightarrow \mathrm{T})\ \ \mathrm{t.invoke}(\bar{\mathrm{d}}) \longrightarrow [\bar{\mathrm{d}} \mapsto \bar{\mathrm{x}}]\ \mathrm{t}$$

Regras de congruência

$$\frac{\mathrm{t} \longrightarrow \mathrm{t}'}{\mathrm{t.invoke}(\bar{\mathrm{t}}) \longrightarrow \mathrm{t}'.\mathrm{invoke}(\bar{\mathrm{t}})}$$

$$\frac{\mathrm{t} \longrightarrow \mathrm{t}'}{\mathrm{t.invoke}(...,\mathrm{t}_i,...) \longrightarrow \mathrm{t.invoke}(...,\mathrm{t}'_i,...)}$$

A primeira regra demonstra o processo de avaliação de uma função anônima, substituindo os valores passados como parâmetro atual $\bar{d}$ nos parâmetros formais $\bar{x}$ que aparecem no termo t que representa o corpo da função. Já nas regras de congruência, primeiro avalia-se o termo a esquerda da invocação da *closure*, e caso ele seja um valor, avalia-se cada um dos parâmetros, da esquerda para a direita, mantendo a estratégia *call-by-value* utilizada na linguagem Java.

Além das regras de avaliação, a Figura 4.10 apresenta as regras de tipos para esta nova funcionalidade da linguagem.

Figura 4.10 – Regras de tipos para *closures*.

Expressão Lambda *bem definida:*

$$\frac{\bar{T}\ \text{OK} \qquad \Gamma, \bar{x} : \bar{T} \vdash t : T}{\Gamma \vdash (\bar{T}\ \bar{x}) \rightarrow t : (\bar{T} \rightarrow T)}$$

Regras para checagem de invocação:

$$\frac{\Gamma \vdash t : (\bar{T} \rightarrow T) \qquad \Gamma \vdash \bar{t} : \bar{S} \qquad \bar{S} <: \bar{T}}{\Gamma \vdash t.\text{invoke}(\bar{t}) : T}$$

A regra que verifica se o *closure* está bem definido, primeiramente avalia os tipos de cada um dos parâmetros formais da função, verificando se estão definidos corretamente de acordo com os tipos primitivos e com as definições na tabela de classes. Estando em conformidade, verifica o tipo do termo da expressão *lambda*, resultando no tipo do *closure*.

Já na verificação de tipos na invocação, primeiramente é verificado se o termo da esquerda é do tipo *função*, posteriormente checando se cada um dos parâmetros passados na invocação são de subtipos daqueles esperados nos parâmetros formais previamente definidos. Se todas as verificações estiverem em conformidade, resulta no tipo de retorno da execução da função.

4.4 Mônadas e Mônada Quântica

Uma mônada é uma maneira de estruturar computações em termos de valores e sequências de transformações. A mônada determina como transformações combinadas formam uma nova computação e liberam o programador de ter que codificar a combinação manualmente cada vez que ela é necessária. Elas foram primeiramente aplicadas na ciência da computação por Moggi (MOGGI, 1988) quando apresentou uma semântica de computações vinda da teoria das categorias. Como um exemplo desta abordagem semântica de computação, mostrou-se uma maneira geral de estruturar vários efeitos computacionais no *cálculo lambda*, como computações usando efeitos colaterais, exceções, computações parciais e não-determinísticas, dentre outros casos.

4.4.1 Construção de Mônadas

A formulação usual de uma mônada para programação é conhecida como *Kleisli Triple* (BARR; WELLS, 2005), e possui os seguintes componentes:

- Um *construtor de tipo* que define, para cada tipo básico, como obter um correspondente tipo monádico. Se M é o nome da mônada e t é o tipo de dado, então $M\ t$ é o correspondente tipo monádico.

- Uma função *return* (também conhecida como *unit*), que transforma um valor de um tipo básico t em seu correspondente tipo monádico $(M\ t)$. Esta função tem um tipo polimórfico $(t \rightarrow M\ t)$.

- Uma operação *bind*, muitas vezes vista como o operador ">>=", do tipo polimórfico $((M\ t) \rightarrow (t \rightarrow M\ u) \rightarrow (M\ u))$. O primeiro argumento é o valor no tipo monádico $(M\ t)$, o segundo é uma função que transforma um valor de tipo básico t em outro tipo monádico $(M\ u)$, e o resultado é dado neste outro tipo monádico (Mu).

A função *bind* tem o objetivo de transformar os dados contidos no tipo monádico apresentado no primeiro argumento, através da função disposta no segundo, gerando e retornando um novo resultado em um novo tipo monádico (LIPOVACA, 2011).

Para construir uma mônada apropriadamente, as definições monádicas devem obedecer no mínimo três axiomas, conhecidos como as leis monádicas.

1. `(return x) »= f ⟶ f x`
2. `m »= return ⟶ m`
3. `(m »= f) »= g ⟶ (\x → f x »= g)`

A primeira lei monádica (*left-identity*) demonstra que utilizando um valor x com a função *return* e então alimentando o resultado na operação $>>=$ para a função f, isto é o mesmo que aplicar x em f. A segunda (*right-identity*) demonstra que se temos um valor monádico e usamos $>>=$ para alimentar a função *return*, o resultado é o valor monádico original. A última lei (*associativity*) diz que quando temos uma cadeia de aplicações de funções monádicas com $>>=$, não importa o modo de como estão aninhadas (LIPOVACA, 2011).

Obedecendo estas três leis, fica garantido que a semântica da *notação-do* usando a mônada será consistente. Construtores de tipo com operações *return* e *bind* que satisfaçam as leis monádicas podem ser considerados como *mônadas*.

4.4.2 Mônada Quântica

O modelo tradicional de computação quântica é baseado em espaços vetoriais, com vetores normalizados para modelar estados computacionais e transformações unitárias, que modelam as computações quânticas realizáveis fisicamente. Usando este modelo, pode-se enfatizar duas características principais da programação quântica (VIZZOTTO, 2006).

- Paralelismo quântico, devido ao fenômeno da superposição.
- Estado quântico global, onde toda operação realizada sobre os qubits deve ser aplicada a todo o estado quântico.

A semântica de qualquer linguagem de programação quântica deve levar estes itens em consideração.

A modelagem de bits quânticos pode ser pensada como um tipo de efeito computacional, devido à sua natureza não-determinística. Mais especificamente, *qubits* podem ser modelados como um tipo de mônada (MU; BIRD, 2001). A ideia desta mônada é construir os estados quânticos, representados matematicamente por um vetor de números complexos contendo as amplitudes de probabilidades dos qubits, com a possibilidade de transformação destes estados através de portas quânticas, representadas por matrizes unitárias, que são aplicadas através da operação *bind* (GRATTAGE et al., 2005).

O modelo apresentado a seguir (VIZZOTTO; ALTENKIRCH; SABRY, 2006) implementa uma mônada quântica usando a linguagem de programação funcional *Haskell*, mostrando como estruturar estados quânticos usando mônadas e na sequência um modo de aplicar as transformações unitárias para os vetores de estado através de operações monádicas. Esta abordagem foi utilizada para modelar a semântica dos efeitos quânticos proposta neste trabalho.

4.4.2.1 Vetores

Os *vetores* representam o estado quântico do sistema, sendo apresentados como uma transformação de valores clássicos observáveis em números complexos, os quais associam cada elemento base com uma amplitude de probabilidade.

Uma implementação genérica utilizando a linguagem Haskell é definida como (VIZZOTTO; ALTENKIRCH; SABRY, 2006):

```
1  type Vec a = a → Complex
2  return a1 a2 = if (a1 ≡ a2) then 1.0 else 0.0
3  bind va f = λ b → sum [(va a) ∗ (f a b) | a ∈ basis]
```

O *construtor de tipos* da mônada quântica, chamado *Vec* define que um tipo básico a é transformado em um número *complexo*. A função *return* constroi vetores quânticos triviais, isto é, apenas os estados base $|0\rangle$ e $|1\rangle$. A operação *bind* especifica como sequenciar computações (VIZZOTTO, 2006).

Além das operações básicas sobre espaços vetoriais, foram propostos dois construtores adicionais, *mzero* e *mplus*, os quais proveem uma computação vazia e uma operação para adicionar computações, conforme pode ser visto no código *Haskell* a seguir:

```
1  mzero :: Vec a
2  mzero = const 0
3  mplus :: Vec a → Vec a → Vec a
4  mplus v1 v2 a = v1 a + v2 a
```

Também foram propostas diversas operações sobre os vetores, como por exemplo, o produto escalar ($\$*$), produto tensorial ($\otimes$), produto interno ($\langle\cdot\rangle$) e produto externo ($\rangle\cdot\langle$) (GRATTAGE et al., 2005). É importante notar que os vetores são capazes de representar corretamente os estados quânticos, incluindo estados *emaranhados* e em *superposição*.

4.4.2.2 Operadores Lineares

Como um vetor é representado como uma função $v \in a \rightarrow Complex$, uma matriz é representada como uma função do tipo $m \in a \rightarrow b \rightarrow Complex$. Vetores denotam estados quânticos, e matrizes unitárias denotam operadores quânticos (GRATTAGE et al., 2005). Como um exemplo, podemos representar em *Haskell* a matriz para negação de um *qubit* (*Pauli-X*), através do código a seguir:

```
1  qnot :: a → Vec b
2  qnot i = if (i == false) then
3                 return true
4             else
5                 return false
```

A definição de uma operação linear especifica sua ação sobre cada elemento individual da base como uma matriz. Para aplicar a operação linear f sobre um vetor v, usamos a operação *bind*, a qual pode ser estendida de modo a atuar sobre vetores com dimensões maiores (VIZZOTTO, 2006).

4.4.3 Extensão Quântica

A extensão quântica monádica apresentada nesta seção utiliza os conceitos explanados na seção anterior e define a semântica formal para a linguagem *FJQuantum*, considerando trabalhos desenvolvidos anteriormente (CALEGARO; VIZZOTTO, 2013; VIZZOTTO; CALEGARO; PIVETA, 2013).

Como uma forma de complementar a linguagem baseada no FJ para fornecer uma definição semântica formal aos trabalhos que inspiraram esta abordagem, foram propostas as regras sintáticas, semânticas e de tipos, as quais serão detalhadas a seguir.

4.4.3.1 Definições Sintáticas

Para contemplar a definição de estados quânticos e as operações monádicas sobre estes estados, foi proposta a sintaxe apresentada na Figura 4.11. Como pode ser percebido, primeiro são definidas as constantes α e β que representam as *amplitudes de probabilidade* através de números complexos. Na sequência, é apresentada a definição do novo tipo adicionado à linguagem, chamado *Vec*, representando os vetores de estados quânticos. O tipo *Vec*<T_B> manipula um tipo genérico, restrito aos tipos base T_B, os quais são representados por *booleanos* ou *tuplas de booleanos*, que atuam como índice para acesso às amplitudes de probabilidade dos estados quânticos.

Para lidar com os tipos quânticos monádicos, foram adicionados seis novos termos. Dentre eles, estão dois construtores: o *mzero*, que representa uma computação vazia, e o *mreturn*, responsável pelo *envelopamento* do tipo base em um valor monádico. Estes dois construtores também podem ser vistos como valores na linguagem. Também foi definido o operador monádico »=, chamado *bind*, que especifica uma maneira de compor computações, isto é, como executar transformações sobre os vetores de estados quânticos. A sintaxe proposta é similar àquela utilizada na linguagem *Haskell* para mônadas, e demonstrada na seção anterior (VIZZOTTO; ALTENKIRCH; SABRY, 2006).

Adicionalmente, foi definido o operador *mplus*, o qual expressa a *adição* de vetores quânticos. Esta regra é necessária para possibilitar a representação de *superposições*. Além disso, é necessário uma funcionalidade para manipular as amplitudes de probabilidade sobre os vetores, a qual é provida pelo operador de produto escalar $\$\star$, tendo o objetivo de aplicar um número complexo a um vetor de estados.

Figura 4.11 – Definições sintáticas para a *mônada quântica*.

Amplitudes Prob. $\alpha, \beta \in \mathbb{C}$

Representação $\texttt{Vec<T}_\texttt{B}\texttt{>} = \texttt{T}_\texttt{B} \to \mathbb{C}$

Tipos

$\texttt{T} ::= \texttt{T}_\texttt{FJ} \mid \texttt{T}_\texttt{G} \mid \texttt{T}_\texttt{P} \mid \texttt{T}_\texttt{T} \mid \texttt{T}_\texttt{F} \mid \texttt{T}_\texttt{Q}$

$\texttt{T}_\texttt{Q} ::= \texttt{Vec<T}_\texttt{B}\texttt{>}$

$\texttt{T}_\texttt{B} ::= \texttt{boolean} \quad \triangleright \text{ Tipos Base}$

$\mid \{\texttt{T}_{\texttt{B}i}{}^{i \in 1..n}\}$

Termos

$\texttt{t} ::= \ldots \quad \triangleright \text{ Expressões FJ}$

$\mid \texttt{mzero}$

$\mid \texttt{mreturn t}$

$\mid \texttt{t}_1 \texttt{ >>= t}_2$

$\mid \texttt{t}_1 \texttt{ mplus t}_2$

$\mid \texttt{t}_1 \texttt{ \$* t}_2$

Valores

$\texttt{v} ::= \ldots \quad \triangleright \text{ Valores FJ}$

$\mid \texttt{mzero}$

$\mid \texttt{mreturn v}$

4.4.3.2 Sistema de Tipos

As definições das regras de tipo para lidar com efeitos quânticos levam em consideração o novo tipo criado *Vec*. Sua construção depende do tipo genérico base $\mathrm{T_B}$, que é utilizado para expressar os qubits dentro do estado quântico.

A Figura 4.12 apresenta as regras de tipo para os construtores monádicos e para a operação *bind*. As primeiras duas regras tratam dos construtores, dizendo que *mzero* é do tipo *Vec<T_B>* e no caso do construtor *mreturn*, para o tipo estar declarado corretamente, é necessário que o termo t seja do tipo base $\mathrm{T_B}$. O operador de composição de computações »= atua sobre dois termos, t_1 e t_2. De acordo com as regras, podemos perceber que t_1 deve ser do tipo *Vec<T_B>*, e t_2 deve ser do tipo função, que transforme um tipo base $\mathrm{T_B}$ em um *Vec<T_B>*, para a declaração ser considerada correta. Deste modo, o processamento do termo resultará em um *Vec<T_B>*.

A Figura 4.13 apresenta as regras de tipo para as operações que atuam sobre os vetores quânticos. As duas primeiras são extensões monádicas que representam a *adição* e *subtração*. Para estarem declaradas corretamente, é necessário que tanto t_1 quanto t_2 sejam vetores com o mesmo tipo base. A partir disto, temos *Vec<T_B>* como resultado do processamento de ambos

Figura 4.12 – Regras de tipo para *construtores monádicos* e operador *bind*.

Tipos Monádicos

$$\frac{}{\Gamma \vdash \text{mzero} : \text{Vec<}T_B\text{>}} \qquad \frac{\Gamma \vdash \text{t}: T_B}{\Gamma \vdash \text{mreturn t} : \text{Vec<}T_B\text{>}}$$

Operação Monádica

$$\frac{\Gamma \vdash t_1\text{: Vec<}T_B\text{>} \quad \Gamma \vdash t_2\text{: } (T_B \rightarrow \text{Vec<}T_B\text{>})}{\Gamma \vdash t_1 \text{ »= } t_2 : \text{Vec<}T_B\text{>}}$$

os operadores. Já na regra de tipos para produto escalar, pode ser percebido que o termo t_1 deve ser um número complexo, e o termo t_2 um vetor quântico. Como resultado do processamento deste operador, teremos também um *Vec<T_B>*.

Figura 4.13 – Regras de tipo para operações sobre vetores.

Monad Plus e Monad Minus

$$\frac{\Gamma \vdash t_1, t_2\text{: Vec<}T_B\text{>}}{\Gamma \vdash t_1 \text{ mplus } t_2 : \text{Vec<}T_B\text{>}}$$

Tipo Escalar

$$\frac{\Gamma \vdash t_1\text{: } \mathbb{C} \quad t_2\text{: Vec<}T_B\text{>}}{\Gamma \vdash t_1 \text{ \$* } t_2 : \text{Vec<}T_B\text{>}}$$

4.4.3.3 Regras de Avaliação

As regras de avaliação definem o comportamento semântico das novas construções, onde cada regra expressa a redução de um termo em outro, apresentando a relação $t \rightarrow t'$, que diz que "o termo t reduz para o termo t' em um passo de avaliação". São estas regras que realmente demonstram o processo monádico para executar computações quânticas.

A Figura 4.14 apresenta a única regra de redução associada aos construtores monádicos. Esta regra de congruência refere-se a avaliação do termo t_1, até que t_1 não possa mais ser reduzido. Os construtores *mzero* e *mreturn*, são valores na linguagem, e portanto, não podem ser reduzidos.

Já na Figura 4.15, são apresentadas as regras de redução para termos aplicados ao operador *bind*, que representa a composição de computações quânticas. As primeiras duas regras,

Figura 4.14 – Regra de congruência para o construtor monádico.

Construtor monádico

$$\frac{t_1 \rightarrow t_1'}{\text{mreturn } t_1 \rightarrow \text{mreturn } t_1'}$$

representam a congruência dos termos t_1 e t_2, até que sejam avaliados para valores, demonstrando a ordem de avaliação, da esquerda para a direita.

Na sequência da Figura 4.15 estão definidas as regras de avaliação para o operador de composição. A primeira regra atua com o construtor *mreturn*, realizando a invocação da *closure* disposta no termo t_2, utilizando como parâmetro o valor v_1 passado para o construtor *mreturn*. A regra de avaliação do operador »= aplicada a termos envolvendo o operador *mplus*, definem a transformação do termo da esquerda aplicado ao termo da direta, na aplicação de t_1 e t_2 na função t_3 separadamente, unindo-as novamente através do operador *mplus*. A última regra especifica o comportamento da aplicação de um termo associado a um produto escalar em uma função. Neste caso, aplicando o termo t_1 em t_2, para posteriormente aplicar o produto escalar.

Figura 4.15 – Regras de avaliação para o operador *bind*.

Regras de congruência

$$\frac{t_1 \rightarrow t_1'}{t_1 \gg\!= t_2 \longrightarrow t_1' \gg\!= t_2}$$

$$\frac{t_2 \rightarrow t_2'}{v_1 \gg\!= t_2 \longrightarrow v_1 \gg\!= t_2'}$$

Regras de avaliação

$$(\text{mreturn } v_1) \gg\!= v_2 \longrightarrow v_2.\text{invoke}(v_1)$$

$$(v_1 \text{ mplus } v_2) \gg\!= v_3 \longrightarrow (v_1 \gg\!= v_3) \text{ mplus } (v_2 \gg\!= v_3)$$

$$(\alpha \ \$* \ v_1) \gg\!= v_2 \longrightarrow \alpha \ \$* \ (v_1 \gg\!= v_2)$$

A Figura 4.16 apresenta as regras de avaliação para o operador monádico *mplus*, que representa a soma de vetores de estados quânticos. As duas primeiras regras referem-se à congruência dos termos, demonstrando a ordem de avaliação da esquerda para a direita. Já nas regras de avaliação, pode ser percebida a aplicação deste operador com o construtor *mzero*, que tem como resultado da avaliação o próprio valor. É importante ressaltar que o operador *mplus*

apresenta as propriedades comutativa e associativa. Para lidar com as propriedades da adição de vetores, é utilizada a lógica de reescrita de termos, que é aplicada repetidas vezes para realizar a simplificação matemática (BELKHIR; GIORGETTI, 2011).

Figura 4.16 – Regras de avaliação para o operador *mplus*.

Regras de Congruência

$$\frac{t_1 \rightarrow t_1'}{t_1 \text{ mplus } t_2 \longrightarrow t_1' \text{ mplus } t_2}$$

$$\frac{t_2 \rightarrow t_2'}{v_1 \text{ mplus } t_2 \longrightarrow v_1 \text{ mplus } t_2'}$$

Regras de avaliação

$$\text{mzero mplus } v_1 \longrightarrow v_1$$

$$v_1 \text{ mplus mzero} \longrightarrow v_1$$

Por fim, a Figura 4.17 define as regras para a aplicação do operador de produto escalar, demonstrando uma maneira de definir as amplitudes de probabilidade para os vetores quânticos. As regras de congruência são similares às já apresentadas para o operadores *mplus*. Já nas regras de avaliação, é realizada a multiplicação das amplitudes de probabilidade associadas tanto ao termo da esquerda, quanto ao da direita, gerando um novo termo.

Figura 4.17 – Regras de avaliação para o *produto escalar*.

Regras de congruência

$$\frac{t_1 \rightarrow t_1'}{t_1 \text{ \$* } t_2 \longrightarrow t_1' \text{ \$* } t_2}$$

$$\frac{t_2 \rightarrow t_2'}{v_1 \text{ \$* } t_2 \longrightarrow v_1 \text{ \$* } t_2'}$$

Regras de avaliação

$$\text{ComplexZero \$* t} \longrightarrow \text{mzero}$$

$$\alpha \text{ \$* } (t_1 \text{ mplus } t_2) \longrightarrow \alpha \text{ \$* } t_1 \text{ mplus } \alpha \text{ \$* } t_2$$

$$\alpha_1 \text{ \$* t mplus } \alpha_2 \text{ \$* t} \longrightarrow (\alpha_1 + \alpha_2) \text{ \$* t}$$

É importante notar, que as regras de congruência para os construtores monádicos definem a ordem de avaliação dos termos.

4.5 Exemplos

Esta seção apresenta alguns exemplos para ilustrar o uso da linguagem proposta e do interpretador. Será demonstrada a criação de uma classe encapsulando estados quânticos e outra com algumas das operações quânticas universais, para posteriormente demonstrar o seu uso na construção e modelagem de algoritmos quânticos.

O exemplo a seguir é usado para encapsular um vetor de estados quânticos em uma classe genérica, tendo um método para *criar* o estado quântico e outro para aplicar *transformações* sobre este estado.

Código-fonte 4.1 – Encapsulamento de estados quânticos em uma classe.

```
1  class QState<T> extends Object {
2      T base;
3      QState(T base) {
4          super();
5          this.base = base;
6      }
7      Vec<T> create() {
8          return mreturn base;
9      }
10     Vec<T> transform((T -> Vec<T>) fun) {
11         return this.create() >>= fun;
12     }
13 }
```

O exemplo seguinte apresenta uma classe que implementa uma série de portas quânticas universais reversíveis. O método *not* é a versão quântica do operador *not* clássico, aplicado a um único *qubit*. O método *hadamard* representa um operador que tem o objetivo de transformar um *qubit* de um estado básico em uma *superposição* de estados. O método *controlledNot* representa um *not* condicional.

Código-fonte 4.2 – Encapsulamento de operações quânticas em uma classe.

```
1  class QOp extends Object {
2      // Construtor e outros metodos
3      (boolean -> Vec<boolean>) not() {
4          return (boolean i) -> if (i == false) {
5                                    mreturn true
6                                }
7                                else {
8                                    mreturn false
9                                };
10     }
11     (boolean -> Vec<boolean>) hadamard() {
```

```
12          return (boolean b) >
13              if (b == false) {
14                  (ComplexHalf $* mreturn false) mplus
15                  (ComplexHalf $* mreturn true)
16              } else {
17                  (ComplexHalf $* mreturn false) mplus
18                  (ComplexMHalf $* mreturn true)
19              };
20      }
21      ({boolean,boolean} > Vec<{boolean,boolean}>)
22        controlledNot() {
23          return ({boolean,boolean} b) >
24              if (b.1 == true) {
25                  if (b.2 == true) {
26                      mreturn {true, false}
27                  } else {
28                      mreturn {true, true}
29                  }
30              } else {
31                  mreturn {b.1,b.2}
32              };
33      }
34  }
```

Dando ênfase à criação de um estado em *superposição*, no método chamado *hadamard*, podemos notar a utilização dos operadores *mplus* e *produto escalar*, explicitando o motivo da criação destes operadores na linguagem proposta. Já no método *controlledNot* podemos visualizar o funcionamento da linguagem ao tratar um estado com mais de um *qubit*. É importante notar o modo como foram criados os operadores, retornando uma expressão lambda, que fornece uma maneira de trabalhar com o operador de composição »= de maneira semelhante ao funcionamento nas linguagens funcionais.

Como mencionado em capítulos anteriores, as operações quânticas devem ser reversíveis, isto é, deve ser possível reconstruir o estado anterior a partir do novo estado após a aplicação de transformações quânticas. No exemplo a seguir pode ser visualizada a aplicação da porta quântica *hadamard* duas vezes sobre um estado quântico, demonstrando o correto comportamento da semântica, agindo como a aplicação de uma função identidade.

Código-fonte 4.3 – Demonstração da reversibilidade dos operadores quânticos.

```
1   class QExec extends Object {
2     // Construtor e outros metodos
3
4     Vec<boolean> hadamardTwice(boolean ini) {
5         return let qop = new QOp() in
6             ((mreturn ini) >>= qop.hadamard())
7                            >>= qop.hadamard();
8     }
9   }
10
11  new QExec().hadamardTwice(false);
```

Após a execução do código acima é apresentado como resultado:

Código-fonte 4.4 – Resultado de duas aplicações da porta de *hadamard*.

```
1  mreturn false
```

O último exemplo apresenta um código com operações complexas, com o objetivo de executar transformações sobre um estado inicial com vários *qubits* usando as classes definidas anteriormente.

Código-fonte 4.5 – Exemplo de operações quânticas compostas.

```
1  class QExec extends Object {
2    // Construtor e outros metodos
3
4    Vec<{boolean,boolean,boolean}> composedOperation() {
5      return let qop = new QOp() in
6        ({boolean,boolean,boolean} state) >
7        ((qop.hadamard()).invoke(state.3)) >>=
8        (boolean b) >
9        ((qop.controlledNot()).invoke({state.1,state.2})) >>=
10       ({boolean,boolean} tm) >
11       ((qop.hadamard()).invoke(b)) >>=
12       (boolean ba) > mreturn {tm.1,tm.2,ba};
13   }
14   Vec<{boolean,boolean,boolean}>
15     exec({boolean,boolean,boolean} ini) {
16       return new QState<{boolean,boolean,boolean}>(ini)
17             .transform(this.composedOperation());
18   }
19 }
20
21 new QExec().exec({true,true,true});
```

Neste exemplo foi possível visualizar uma maneira de aplicar transformações parciais no estado quântico, e também como compor operações através do operador *bind*. O método *composedOperation* atua sobre um estado quântico de três *qubits* e aplica na sequência o operador *hadarmard* ao terceiro *qubit*, o operador *controlledNot* ao primeiro e segundo, novamente aplicando *hadarmard* sobre o primeiro *qubit*, para finalmente retornar o processamento do algoritmo. O método *exec* somente demonstra um ponto de entrada para o processamento na classe.

Os exemplos apresentados mostram como expressar algoritmos quânticos de forma concisa na linguagem proposta usando o paradigma orientado a objetos, demonstrando como a nova semântica monádica se encaixa com a proposta original do Featherweight Java.

5 UM INTERPRETADOR PARA FJQUANTUM

A partir das definições apresentadas nos capítulos anteriores, foi desenvolvido um interpretador, levando em consideração todas as características da proposta Featherweight Java, com as extensões já mencionadas anteriormente. O objetivo principal é viabilizar a aplicação do conceito funcional de *mônadas* de modo a possibilitar o uso das mônadas quânticas como meio para simulação do computador quântico em uma linguagem orientada a objetos, facilitando a implementação e o encapsulamento de estados e portas quânticas.

Para implementar o interpretador, foi utilizada a linguagem de programação *Haskell*, que é uma avançada linguagem de programação puramente funcional, baseada no cálculo lambda, polimorficamente e estaticamente tipada, com avaliação *lazy* e funcionalidade de *pattern matching*. Muitas das funcionalidades fornecidas pela linguagem auxiliam no processo de manipulação e interpretação das árvores de sintaxe abstratas (AST).

Na sequência serão apresentadas as etapas de implementação do interpretador.

5.1 Analisador Léxico

O analisador léxico é responsável pelo processo de converter o código-fonte da linguagem em uma lista de *tokens*. A Tabela 5.1 apresenta todas as palavras reservadas na linguagem, e seu token correspondente.

Tabela 5.1 – Palavras reservadas da linguagem.

Palavra-Chave	Token
class	TokenKWClass
extends	TokenKWExtends
super	TokenKWSuper
this	TokenKWThis
new	TokenKWNew
return	TokenKWReturn
Object	TokenKWObject
if	TokenKWIf
else	TokenKWElse
let	TokenKWLet
in	TokenKWIn
invoke	TokenKWInvoke

Palavra-Chave	Token
boolean	TokenKWBoolean
true	TokenKWTrue
false	TokenKWFalse
Complex	TokenKWComplex
ComplexOne	TokenKWComplexOne
ComplexZero	TokenKWComplexZero
ComplexHalf	TokenKWComplexHalf
ComplexMHalf	TokenKWComplexMHalf
mzero	TokenKWMZero
mreturn	TokenKWMReturn
mplus	TokenKWMPlus

Já a Tabela 5.2 apresenta os caracteres especiais aceitos pela linguagem.

Além das palavras-chave e dos caracteres especiais, os usuários podem utilizar números

Tabela 5.2 – Caracteres especiais da linguagem.

Caracter Especial	Token	Caracter Especial	Token
‘{’	TokenLBrace	‘>’	TokenGT
‘}’	TokenRBrace	“->”	TokenArrow
‘(’	TokenLParen	‘*’	TokenTimes
‘)’	TokenRParen	‘+’	TokenPlus
‘,’	TokenComma	‘-’	TokenMinus
‘;’	TokenSemi	“==”	TokenEqual
‘.’	TokenDot	“>>=”	TokenBind
‘=’	TokenAssign	“$*”	TokenScalar
‘<’	TokenLT		

na linguagem, os quais representam as posições de acessos em tuplas, e nomes, que podem representar classes, atributos, variáveis, métodos, etc.

Tabela 5.3 – Nomes ou números definidos pelo usuário.

Declarações	Token
name	TokenName $$
number	TokenNumber $$

O funcionamento do analisador léxico se dá pela leitura de cada caracter escrito no código fonte do programa, o qual é repassado para a função *lexer*, responsável por gerar uma lista de *tokens* na saída do processo. No código fonte abaixo, podemos visualizar o funcionamento desta função.

Código-fonte 5.1 – Trecho de código-fonte do analisador léxico.

```
1  lexer :: String > [Token]
2  lexer [] = []
3  lexer ('{':cs) = TokenLBrace : lexer cs
4  lexer ('}':cs) = TokenRBrace : lexer cs
5  lexer ('(':cs) = TokenLParen : lexer cs
6  lexer (')':cs) = TokenRParen : lexer cs
7  lexer (',':cs) = TokenComma : lexer cs
8  lexer (';':cs) = TokenSemi : lexer cs
9  lexer ('.':cs) = TokenDot : lexer cs
10 lexer ('+':cs) = TokenPlus : lexer cs
11 lexer (c:cs)   | isSpace c = lexer cs
12                | isAlpha c = lexStr (c:cs)
13                | isDigit c = lexDigit (c:cs)
14                | isToken c = lexSymbol (c:cs)
```

Como pode ser visto no código fonte acima, além da conversão direta de alguns caracteres especiais da linguagem, foram utilizadas algumas funções auxiliares para tratar das especificidades do código fonte, incluindo leitura de *strings*, *digitos* e *símbolos*. Este processo garante que somente foram utilizados caracteres permitidos na linguagem.

5.2 Analisador Sintático

A análise sintática (*parsing*) é o processo que analisa uma sequência de *tokens* de entrada, para verificar a sua estrutura gramatical, de acordo com uma gramática formal, em geral expressa através da notação *BNF*.

A linguagem *Haskell* possui um sistema gerador de analisadores sintáticos, chamado *Happy*, similar às ferramentas *yacc* para a linguagem C e *AntLR*, o qual foi utilizado para facilitar a construção do *parser* da linguagem. Esta ferramenta trabalha com um arquivo que contém a especificação de uma gramática *BNF*, produzindo como saída um módulo *Haskell* contendo o *parser* para a gramática apresentada.

Para cada unidade de compilação, o analisador sintático manipula a lista de tokens gerada pelo analisador léxico, transformando-a em uma *AST*. Nesta fase é verificado se o código está escrito corretamente (sintaticamente correto), ou seja, as construções informadas no código-fonte respeitam as regras gramaticais definidas. Para o caso do FJ com a extensão quântica, um programa é visto basicamente como uma lista de classes, seguidas de um termo para processamento. A definição da gramática para um programa escrito na linguagem proposta pode ser vista abaixo:

Código-fonte 5.2 – Trecho da gramática *BNF* que armazena o código-fonte do programa sendo interpretado.

```
1  Program     : ClassList Term ';'              { Program $1 $2 }
2
3     Construtor de tipos utilizado para a geracao das ASTs.
4  data Program = Program [(String,ClassDef)] Term
5                  deriving (Show, Eq)
```

Uma lista de classes é representada por uma ou mais classes e a partir do processo de análise sintática é transformada em uma estrutura chamada de *tabela de classes*, que é utilizada em todos os processos na análise semântica. A tabela de classes é definida por uma tupla em *Haskell*, que tem na sua primeira posição uma *string*, representando o nome da classe criada, e uma estrutura *ClassDef*, a qual contém todas as definições internas de uma classe, como atributos, construtores e métodos. A descrição da gramática fornecida para o *Happy* pode ser vista a seguir:

Código-fonte 5.3 – Trecho da gramática *BNF* para as classes da linguagem.

```
1     Lista de classes
2  ClassList : ClassDef                                  { [$1] }
3            | ClassList ClassDef                        { $2 : $1 }
4
5  ClassDef : class ClassName GenericDefList extends ClassName '{'
```

```
6                AttrList ConstrDef MethodList
7              '}' { ($2,ClassDef $2 $3 $5 $7 $8 $9) }
```

Basicamente, o módulo gerado *Happy* processa a lista de *tokens* recebida do analisador léxico e a transforma utilizando um construtor de tipo previamente definido, gerando a árvore de sintaxe abstrata para o código sendo processado. O código abaixo apresenta os construtores de tipo para classes, construtores e métodos.

Código-fonte 5.4 – Construtores da *AST* utilizados na linguagem.

```
1  data ClassDef = ClassDef String [(String,String)] String
2                     [(Type,String)] ConstrDef [MethodDef]
3                  deriving (Show, Eq)
4
5  data ConstrDef = ConstrDef String [(Type,String)]
6                      [String] [(String,String)]
7                   deriving (Show, Eq)
8
9  data MethodDef = MethodDef Type String [(Type,String)] Term
10                  deriving (Show, Eq)
```

Além do processamento da tabela de classes, o analisador sintático verifica a construção do termo, que representa o ponto de partida da execução do programa fornecido para a linguagem proposta. Este termo pode ser descrito por diferentes construções, conforme pode ser visto no código abaixo:

Código-fonte 5.5 – Trecho da gramática *BNF* para os termos da linguagem.

```
1  Term : BooleanLiteral                    { BooleanLiteral $1 }
2       | ComplexLiteral                    { ComplexLiteral $1 }
3       | name                              { Var $1 }
4       | this '.' name                     { ThisAccessAttr $3 }
5       | this '.' name '(' TermList ')'    { ThisAccessMeth $3 $5 }
6       | Term '.' name                     { AttrAccess $1 $3 }
7       | Term '.' name '(' TermList ')'    { MethodAccess $1 $3 $5 }
8     Outros possiveis termos ...
9       | mzero                             { MonadZero }
10      | mreturn Term                      { MonadReturn $2 }
11      | '(' Term ')' ">>=" Term           { MonadBind $2 $5 }
12      | '(' Term ')' mplus '(' Term ')'   { MonadPlus $2 $6 }
13      | Term "$*" Term                    { ScalarProduct $1 $3 }
```

Cada uma das construções da gramática *BNF* apresentadas precisa ter um construtor de tipo associado (*AST*), sendo a descrição completa de ambos está disponível no Apêndice B.

5.3 Analisador Semântico

A etapa de análise semântica do interpretador é aquela que dá significado e realiza as verificações de tipos das construções de alto nível da linguagem. Nesta etapa, o código está sin-

taticamente correto, ou seja, respeitando as regras de construção do código-fonte e a(s) AST(s) estão disponíveis para processamento.

Um programa descrito utilizando o paradigma OO é, em geral, composto por um conjunto de classes, com um ou mais pontos de entrada para o processamento das mesmas. Após a análise sintática, é disponibilizada para interpretação a *tabela de classes*, contendo todas aquelas codificadas no programa, e um único termo, que representa o início do processamento. O termo usa as classes para desenvolver a lógica necessária do programa e, deste modo, são necessários mecanismos para a manipulação da *tabela de classes*.

5.3.1 Funções Auxiliares

As regras do FJ original preveem diversas definições auxiliares para manipulação das regras no paradigma OO, bem como para percorrer a *tabela de classes*. Dentre as funções auxiliares, podemos citar a verificação de *subtipos* entre os tipos disponíveis na linguagem, obtenção de atributos, nome e tipo de retorno dos métodos, e obtenção do termo disposto dentro de um método disponível em determinada classe.

A implementação para verificação de subtipos reflete as regras dispostas na Figura 3.2, verificando se uma classe é subtipo de outra, de acordo com as definições na *tabela de classes*. Esta função auxilia no processo de verificação de tipos e interpretação de termos que envolvem o mecanismos de herança. O código pode ser visto abaixo:

Código-fonte 5.6 – Código-fonte para a função *isSubtype*.

```
1  isSubtype :: String > String > FJClassTable
2                > FJClassTable > Bool
3  isSubtype d b (h:t) ct
4      | d == b = True
5      | otherwise =
6          if (fst(h) == d) then
7            if (b == getBaseClass(snd(h))) then
8              True
9            else
10             isSubtype (getBaseClass(snd(h))) b ct ct
11         else
12           isSubtype d b t ct
```

Outra função auxiliar relevante para a implementação do interpretador refere-se à obtenção dos atributos de uma classe, sendo retornada uma lista contendo todos os identificadores com seus respectivos tipos, implementando as primeiras regras das definições da Figura 3.3, conforme pode ser visto no código abaixo.

Código-fonte 5.7 – Código-fonte para a função *fieldLookup*.

```
1  fieldLookup :: String > [Type] > FJClassTable
2                 > FJClassTable > [(Type, String)]
3  fieldLookup _ _ _ [] = []
4  fieldLookup "Object" _ _ _ = []
5  fieldLookup c g (h:t) all =
6      if (c == fst(h)) then
7        let base = getBaseClass(snd(h))
8          in (getFieldList (snd(h)) g) ++
9            fieldLookup base g all all
10     else
11       fieldLookup c g t all
```

As funções que tratam da obtenção dos métodos de uma classe auxiliam no processo de interpretação da invocação de métodos e também da avaliação dos tipos. Para essas situações, foram modeladas no interpretador as funções *methodTypeLookup* e *methodBodyLookup*, implementando as regras apresentadas anteriormente na Figura 3.4 e na Figura 3.7 respectivamente.

5.3.2 Verificação de Tipos

A etapa de verificação de tipos tem a responsabilidade de verificar a coerência na utilização dos tipos em um programa. É nesta etapa que são verificados os atributos, métodos e a construção de classes e termos, garantindo que, em caso de sucesso, o programa executará sem erros de tipo (*type-safe*), prevenindo que operações esperadas para um certo tipo de valores sejam executadas com outros tipos não suportados. Na implementação do presente interpretador, esta é a penúltima etapa, ocorrendo imediatamente antes da avaliação dos termos, garantindo que as classes estão bem formadas e os termos só utilizam tipos válidos no programa.

Para garantir que as classes estejam bem formadas, é necessário avaliar cada classe contida na tabela de classes, conforme as regras de tipo já apresentadas nos capítulos anteriores. Este processo é realizado pela função *checkClassTable*, apresentada abaixo:

Código-fonte 5.8 – Código-fonte para a função *checkClassTable*.

```
1  checkClassTable :: FJClassTable > FJClassTable > Bool
2  checkClassTable [] _ = True
3  checkClassTable (h:t) ct = if (checkClass (snd(h)) ct [])
4                             then
5                               checkClassTable t cts
6                             else
7                               False
```

Cada classe, para estar corretamente definida, deve verificar seus atributos, seu construtor e cada um dos seus métodos. Os tipos dos atributos devem obrigatoriamente ser de algum tipo disponível na linguagem ou estarem implementados no código-fonte sendo avaliado como uma classe. Estes mesmos atributos devem ser inicializados no construtor. Já com relação aos

métodos, deve-se garantir que cada um deles utilize apenas tipos válidos, seja como retorno, em seus parâmetros, ou dentro do termo a ser avaliado pelo método. O código abaixo apresenta a função responsável por checar os tipos em uma classe:

Código-fonte 5.9 – Código-fonte para a função *checkClass*.

```
1  checkClass :: ClassDef > FJClassTable > TypeContext > Bool
2  checkClass (ClassDef n p attr cons meth) ct ctx =
3      if ((checkAttributes attr ct) && (checkConstructor n cons ct)
4         && (checkMethods n meth ct ctx)) then
5           True
6      else
7           False
```

Podemos visualizar que cada definição sintática da classe tem suas próprias funções de verificação de tipos, sendo: *checkAttributes* para verificar a coerência dos atributos, *checkConstructor* para verificar se o construtor está bem formado, e *checkMethods*, com a responsabilidade de verificar cada um dos métodos descritos na classe.

O ponto de início da execução de um programa escrito para FJ é o termo disposto ao final do código-fonte, após as definições de classes. É neste termo que é executado o processo de criação de objetos, acesso a atributos, invocação de métodos, etc. Na análise dos tipos, deve-se verificar cada uma das possíveis construções de termos, garantindo que os tipos sejam utilizados adequadamente. O trecho de código abaixo apresenta parte da implementação desta funcionalidade.

Código-fonte 5.10 – Código-fonte para a função *checkTerm*.

```
1  checkTerm :: Term > FJClassTable > TypeContext > Bool
2  checkTerm (AttrAccess t f) ct ctx = checkTerm t ct ctx
3  checkTerm (MethodAccess t m tl) ct ctx =
4      let tp = typeof ctx t ct
5          ex = fst(methodTypeLookup m (getNameFromType(tp)) ct)
6        in (checkTerm t ct ctx) && (checkParamList ex tl ct ctx)
7  checkTerm (CreateObject obj tl) ct ctx =
8      let fl = fieldLookup obj ct ct
9        in (classTableContains (TypeClass obj) ct) &&
10          (checkConstrParamList fl tl ct ctx)
11 // Outras checagens de termos
12 checkTerm (ClosureDef pc t) ct ctx =
13     (checkClosureTerm t pc ct) && (checkClosureParamDefs pc ct)
14 checkTerm (InvokeClosure t tl) ct ctx =
15     let tt = typeof ctx t ct
16       in checkClosureInv tt tl ct ctx
17 checkTerm (MonadReturn t) ct ctx =
18     checkMonadReturn (typeof ctx t ct)
19 checkTerm (MonadBind t1 t2) ct ctx =
20     let tp1 = typeof ctx t1 ct
21         tp2 = typeof ctx t2 ct
22       in checkMonadBind tp1 tp2
23 checkTerm (MonadPlus t1 t2) ct ctx =
24     let tp1 = typeof ctx t1 ct
```

```
25          tp2 = typeof ctx t2 ct
26        in checkMonadicOperator tp1 tp2
27  checkTerm (ScalarProduct c t) ct ctx =
28      let tc = typeof ctx c ct
29          tt = typeof ctx t ct
30        in checkScalarProduct tc tt
```

A verificação dos termos em geral utiliza uma estrutura recursiva, onde termos parciais são verificados através da mesma função *checkTerm*. Em muitos casos, existem vários termos embutidos em um único termo inicial, e esta função é responsável pela verificação de cada um deles, garantindo a correta utilização dos tipos na linguagem.

Explanando especificamente os termos que envolvem conceitos de computação quântica, para a verificação do termo *MonadReturn*, foi criada uma outra função (*checkMonadReturn*), com a responsabilidade de implementar a verificação de tipos para o construtor *mreturn*. Esta verificação deve garantir que somente sejam utilizados os tipos definidos nas regras apresentadas no capítulo anterior, onde somente são aceitos *booleanos* ou *tuplas de booleanos* para o construtor quântico monádico. A função que descreve esta funcionalidade é apresentada abaixo:

Código-fonte 5.11 – Código-fonte para a função *checkMonadReturn*.

```
1  checkMonadReturn :: Type -> Bool
2  checkMonadReturn TypeBool = True
3  checkMonadReturn (TypeTuple []) = True
4  checkMonadReturn (TypeTuple (h:t)) =
5      if (checkMonadReturn h) then
6          checkMonadReturn (TypeTuple t)
7      else
8          False
9  checkMonadReturn _ = False
```

Na regra de verificação para o construtor monádico *bind* também foi desenvolvida uma função específica para tratar os tipos para este operador. Esta função tem por objetivo garantir que o termo sendo processado à esquerda seja do tipo *Vec* (*TypeQuantum*) e o termo da direita seja representado por uma *closure*. Além disso, o tipo base do vetor de estados quânticos deve ser do mesmo tipo do parâmetro recebido pela *closure*. Caso as definições estejam de acordo com a especificação, a função retorna *True*. O código fonte abaixo implementa esta funcionalidade.

Código-fonte 5.12 – Código-fonte para a função *checkMonadBind*.

```
1  checkMonadBind :: Type -> Type -> Bool
2  checkMonadBind (TypeQuantum bt)
3                 (TypeClosure (TypeQuantum _) pt) =
4      if ([bt] == pt) then
5          True
6      else
7          False
8  checkMonadBind _ _ = False
```

Para os outros operadores monádicos também foram desenvolvidas funções auxiliares de modo a realizar a verificação de tipos. Para verificar o operador *mplus*, foi desenvolvida a função *checkMonadicOperator*, que tem pro objetivo validar se ambos os termos utilizados são do tipo quântico *Vec*. No caso da verificação para o produto escalar, existe a função *checkScalarProduct*, que verifica se o termo à esquerda avalia para um número complexo e se o termo da direita avalia para um vetor quântico.

Como uma forma de manter as definições dos tipos dos termos em seu escopo, foi definida a função *typeof*, que retorna o tipo de cada termo, e utiliza-se do contexto Γ para obter os tipos dos termos em seu escopo específico. O contexto Γ na implementação é representado por uma lista de tuplas, contendo cada termo com o seu respectivo tipo. No início do programa, o contexto é vazio, sendo posteriormente preenchido pelas variáveis repassadas como parâmetros para funções ou métodos. O trecho de código abaixo, apresenta parte da implementação da função *typeof*:

Código-fonte 5.13 – Trecho de código-fonte para a função *typeof*.

```
1  typeof :: TypeContext > Term > FJClassTable > Type
2     Tipos primitivos
3  typeof ctx (BooleanLiteral _) _ = TypeBool
4  typeof ctx (ComplexLiteral _) _ = TypeComplex
5     Tipos FJ
6  typeof ctx (Var v) _ = getTypeFromContext ctx (Var v)
7  typeof ctx (AttrAccess t f) ct =
8      let tc = typeof ctx t ct
9          fl = fieldLookup (getNameFromType(tc)) ct ct
10       in getAttrAccessType fl f
11 typeof ctx (MethodAccess t m pm) ct =
12     let tc = typeof ctx t ct
13         mt = methodTypeLookup m (getNameFromType(tc)) ct
14         pcs = getParamsTypeFromContext ctx pm ct
15       in getMethodAccessType pcs mt ct
16 typeof ctx (CreateObject c pc) ct =
17     let pm' = getParamsTypeFromContext ctx pc ct
18         fl = fieldLookup c ct ct
19       in getConstructorType c pm' fl ct
20    Outros possiveis termos
21    Tipos para closures
22 typeof ctx (ClosureDef pc t) ct =
23     let rt = typeof ctx' t ct
24       in TypeClosure rt (getParamsType(pc))
25     where
26       ctx' = addKeyPairToContext ctx pc
27 typeof ctx (InvokeClosure t pc) ct =
28     let tt = typeof ctx t ct
29         in getClosureReturnType tt
30    Tipos monadicos
31 typeof ctx (MonadReturn t) ct =
32     let bt = typeof ctx t ct
33       in TypeQuantum bt
```

```
34  typeof ctx (MonadBind t1 t2) ct =
35      let tp = typeof ctx t2 ct
36        in getClosureReturnType tp
```

Podemos perceber na função *typeof* a avaliação recursiva da obtenção de tipos. Caso sejam avaliados tipos primitivos, como *booleanos*, *complexos*, *closures*, o tipo é retornado diretamente. No caso de avaliar uma *variável*, o contexto Γ é consultado. O caso especial que trata da verificação de tipos do construtor monádico *mreturn* faz uma chamada recursiva à função *typeof*, com objetivo de obter o tipo do termo sendo processado. Em posse deste tipo, é retornado o construtor *TypeQuantum*, com o tipo do termo à direta, que deve ser obrigatoriamente um *booleano* ou uma tupla de *booleanos*. Para os demais casos, os termos são avaliados recursivamente, até que seja obtido o tipo de retorno da avaliação da expressão sendo processada.

Caso a tabela de classes e o termo principal do programa sejam válidos para o sistema de tipos ao final da verificação, a etapa é finalizada, e as *ASTs* são repassadas para a etapa de avaliação dos termos. Caso seja encontrado algum erro na verificação de tipos, o programa é abortado imediatamente, e na maioria dos casos apresenta uma mensagem descritiva da ocorrência do erro.

5.3.3 Avaliação de Termos

A avaliação dos termos representa a etapa final do processamento do interpretador. Como o FJ é uma visão funcional de Java, sem apresentar efeitos colaterais, o funcionamento da avaliação dos termos é similar ao processamento de linguagens funcionais. Como utilizamos *Haskell* para desenvolver o interpretador, a implementação das funções que realizam a redução dos termos fica muito próxima às regras semânticas definidas nos capítulos anteriores.

As regras definidas na semântica da linguagem definem funções parciais que, quando aplicadas a um termo, que ainda não é um valor, resultam em um novo termo, que representa o próximo passo de avaliação para o termo inicial. Quando aplicadas a um valor, resultam neste mesmo valor, indicando que o processamento do interpretador está finalizado.

Para realizar a interpretação de um programa escrito na linguagem proposta, existe a função *eval*, responsável por avaliar recursivamente o termo principal até que ele seja transformado em um valor. Para realizar a avaliação de um termo, foi desenvolvida a função *evalTerm*, com a responsabilidade de realizar um passo de avaliação. O trecho de código abaixo, apresenta parte da função *evalTerm*.

Código-fonte 5.14 – Trecho de código-fonte para a função *evalTerm*.

```
1  evalTerm :: Term > FJClassTable > Term
2     Termos FJ
3  evalTerm (AttrAccess t f) ct = evalAttrAccess t f ct
4  evalTerm (MethodAccess t m pm) ct = evalMethodAccess t m pm ct ct
5  evalTerm (CreateObject c g pc) ct = let pc' = evalTermList pc ct
6                                      in CreateObject c g pc'
7     Termos para closures
8  evalTerm (InvokeClosure t ap) ct = evalClosure t ap ct
9     Avaliacao de outros termos triviais ...
10    Termos monadicos
11 evalTerm (MonadReturn t) ct = evalMonadReturn t ct
12 evalTerm (MonadBind t1 t2) ct = evalMonadBind t1 t2 ct
13 evalTerm (MonadPlus t1 t2) ct = evalMonadPlus t1 t2 ct
14 evalTerm (ScalarProduct t1 t2) ct = evalScalarProduct t1 t2 ct
```

Neste trecho de código, podemos notar a utilização da funcionalidade de *pattern matching* oferecida pela linguagem *Haskell*, facilitando o processamento da árvore de sintaxe abstrata fornecida pelas etapas prévias do interpretador. O processamento do acesso a atributos e métodos é realizado pelas funções *evalAttrAccess* e *evalMethodAccess* respectivamente. Já no caso da avaliação da criação de objetos, o único passo realizado é avaliar os parâmetros repassados ao construtor da classe (congruência), pois como já visualizado na Figura 3.1, a construção de um objeto pode ser vista como um valor final para a linguagem, quando todos os seus atributos estiverem reduzidos a valores.

Já nas funcionalidades adicionadas ao FJ neste trabalho, como *closures* e termos monádicos, também foram criadas funções com o objetivo específico de processamento, como *evalClosure*, *evalMonadReturn*, *evalMonadBind*, etc., de modo a manter o código organizado e passível de manutenção. Alguns termos triviais foram suprimidos do trecho de código acima, mas suas especificações estão explícitas nas regras semânticas apresentadas nos capítulos anteriores.

A função *evalAttrAccess* é responsável por *reduzir* um termo que realiza o acesso a um atributo de uma classe. Primeiramente é verificado se o termo sendo processado é um valor no formato de criação de objeto (*CreateObject*). Em caso positivo, é utilizada a função auxiliar *fieldLookup*, para posteriormente buscar o atributo na posição respectiva àquela repassada ao construtor, conforme já demonstrado na Figura 3.6. Caso contrário, é aplicada a regra de congruência, vista na Figura 3.9, sobre o termo sendo processado.

Código-fonte 5.15 – Código-fonte para a função *evalAttrAccess*.

```
1  evalAttrAccess :: Term > String > FJClassTable > Term
2  evalAttrAccess (CreateObject obj g p) f ct =
3      let attrs = fieldLookup obj g ct ct
4        in getAttrValue f attrs p
```

```
5  evalAttrAccess t f ct = let t' = evalTerm t ct
6                          in AttrAccess t' f
```

O objetivo da função *evalMethodAccess* é processar uma invocação de um método, que deve, dentre outras coisas, avaliar os parâmetros atuais passados para o método até que tornem-se valores, deixando explícita a estratégia de avaliação *call-by-value*, substituir os parâmetros formais pelos parâmetros atuais dentro do corpo do método, substituir as ocorrências de acesso a atributos internos da classe através da palavra-chave especial *this*, para finalmente retornar o termo presente dentro deste método. Caso o termo que esteja invocando um método não esteja na forma da construção de um objeto, é também aplicada a regra de congruência no mesmo. Podemos visualizar um trecho de código que implementa parte da operação descrita na Figura 3.7.

Código-fonte 5.16 – Código-fonte para a função *evalMethodAccess*.

```
1  evalMethodAccess :: Term > String > [Term] > FJClassTable >
2                      FJClassTable > Term
3  evalMethodAccess _ _ _ [] _ =
4      error "Classe nao encontrada na tabela de classes."
5  evalMethodAccess (CreateObject obj g pc) m pm (h:t) ct =
6      if (fst(h) == obj) then
7        let bodydef = methodBodyLookup m (snd(h))
8                      body = replaceThis (snd(bodydef))
9                                         (CreateObject obj g pc)
10                     params = fst(bodydef)
11                     pm' = evalTermList pm ct     E Invk Arg
12         in subsParams body params pm'
13     else
14       evalMethodAccess (CreateObject obj g pc) m pm t ct
15 evalMethodAccess t m pm ct _ = let t' = evalTerm t ct
16                               in MethodAccess t' m pm
```

As funções *replaceThis* e *subsParams* são cruciais para o correto funcionamento do interpretador, e são pontos chave para a implementação da avaliação no paradigma OO. Estas funções tem implementações similares em termos de sua codificação, porém, a primeira substitui a palavra chave *this* por uma nova construção da própria classe que está sendo processada, deixando explícito que esta semântica apresenta uma versão funcional da linguagem Java. A segunda realiza a substituição dos parâmetros formais pelos valores recebidos na invocação de um método. Ambas as funções devem considerar todos os termos possíveis na linguagem para realizar a substituição. Abaixo é apresentado apenas um trecho da função *subsParams*.

Código-fonte 5.17 – Código-fonte para a função *subsParams*.

```
1  subsParams :: Term > [String] > [Term] > Term
2  subsParams t [] _ = t
3  subsParams (Var v) (h1:t1) (h2:t2) =
4      if (h1 == v) then
```

```
5           h2
6       else
7           subsParams (Var v) t1 t2
8   subsParams (AttrAccess t f) fp ap =
9       let t' = subsParams t fp ap
10        in AttrAccess t' f
11  subsParams (MethodAccess t m pm) fp ap =
12      let t' = subsParams t fp ap
13          pm' = subsParamsList pm fp ap
14        in MethodAccess t' m pm'
15     Substituicoes aplicadas a outros termos triviais
16  subsParams (MonadReturn t) fp ap =
17      let t' = subsParams t fp ap
18        in MonadReturn t'
19  subsParams (MonadBind t1 t2) fp ap =
20      let t1' = subsParams t1 fp ap
21          t2' = subsParams t2 fp ap
22        in MonadBind t1' t2'
```

A função *evalClosure* implementa a interpretação para uma funcionalidade adicionada à proposta FJ original, permitindo a utilização de *expressões lambda* nesta linguagem. Nas construções sintáticas, foram acrescentadas as possibilidades de definição de *closures* e a possibilidade de invocação das mesmas. Como podemos visualizar na Figura 4.8, a definição de uma closure pode ser vista como um valor, e portanto não possui regras de avaliação. Já no caso da invocação, a implementação baseia-se nas regras dispostas na Figura 4.9, primeiramente avaliando os parâmetros repassados para a função, para posteriormente aplicar a substituição destes parâmetros no corpo da mesma. Caso o termo que esteja realizando a invocação da função não esteja na forma de uma *closure*, é aplicada a regra de congruência. O código abaixo apresenta a implementação destas regras.

Código-fonte 5.18 – Código-fonte para a função *evalClosure*.

```
1  evalClosure :: Term > [Term] > FJClassTable > Term
2  evalClosure (ClosureDef pf t) tl ct =
3      let params = getParamsName pf
4          tl' = evalTermList tl ct
5        in subsParams t params tl'
6  evalClosure t pc ct = let t' = evalTerm t ct
7                          in InvokeClosure t' pc
```

A função *evalMonadReturn* trata apenas da regra de congruência para o termo que representa o construtor quântico monádico que não estão na forma de um tipo base, uma vez que o construtor *mreturn* pode ser visto como um valor nesta linguagem. O código fonte desta função pode ser visto abaixo.

Código-fonte 5.19 – Código-fonte para a função *evalMonadReturn*.

```
1  evalMonadReturn :: Term > FJClassTable > Term
2  evalMonadReturn t ct = let t' = evalTerm t ct
3                          in MonadReturn t'
```

Já a função *evalMonadBind* implementa o operador de composição de computações quânticas, e considera as regras apresentadas na Figura 4.15. Neste operador, ambos os termos devem estar na forma de um valor, sendo o termo da esquerda como um valor do tipo *Vec*, e o da direita como um valor do tipo função, conforme já explanado na seção anterior. Caso não estejam nesta forma, são aplicadas as regras de congruência. O código que implementa estas regras pode ser visto abaixo:

Código-fonte 5.20 – Código-fonte para a função *evalMonadBind*.

```
1  evalMonadBind :: Term > Term > FJClassTable > Term
2  evalMonadBind (MonadReturn (BooleanLiteral b)) t2 ct =
3          InvokeClosure t2 [(BooleanLiteral b)]
4  evalMonadBind (MonadReturn (Tuple t1)) t2 ct =
5          InvokeClosure t2 [(Tuple t1)]
6  evalMonadBind (MonadPlus t1 t2) t3 ct =
7      let t1' = evalMonadBind t1 t3 ct
8          t2' = evalMonadBind t2 t3 ct
9        in MonadPlus t1' t2'
10 evalMonadBind (ScalarProduct (ComplexLiteral n) t1) t2 ct =
11     let t' = evalMonadBind t1 t2 ct
12       in ScalarProduct (ComplexLiteral n) t'
13 evalMonadBind t1 t2 ct = let t1' = evalTerm t1 ct
14                             t2' = evalTerm t2 ct
15                         in MonadBind t1' t2'
```

Como pode ser percebido no código acima, a interpretação do operador *bind* considera os termos *mreturn*, *mplus* e o operador de produto escalar. No primeiro caso (*MonadReturn*), a *closure* que representa a operação quântica sendo realizada é invocada diretamente, recebendo como parâmetro os tipos base contidos dentro do tipo monádico. Para o caso do operador *mplus*, são aplicadas as associações, conforme as definições semânticas, sendo cada termo processado separadamente pela *closure*. No último caso (*ScalarProduct*), o termo à direita é processado pelo operador quântico, para posteriormente aplicar o produtor escalar.

Neste capítulo foram apresentadas as principais características do interpretador da linguagem orientada à objetos baseada no Featherweight Java, capaz de manipular informações de computação quântica em uma abordagem monádica, apresentando passo a passo os principais aspectos da sua implementação.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este projeto apresentou uma extensão da linguagem Featherweight Java, com funcionalidades para manipular conceitos da mecânica quântica, como *qubits* e *portas quânticas*, adaptando uma abordagem monádica, que vem das linguages funcionais, para um contexto orientado a objetos, provendo regras que formalizam o trabalho de Calegaro e Vizzotto (CALEGARO; VIZZOTTO, 2013). Como contribuição desta dissertação podemos elencar o desenvolvimento de uma semântica monádica para trabalhar com programas ou algoritmos considerando as características da computação quântica através do paradigma OO. Além disso, foi demonstrada a implementação de um interpretador para o FJ com a extensão proposta, como uma maneira de verificar as definições da semântica operacional e do sistema de tipos.

Atualmente existem diversas pesquisas no campo de simulação de computação quântica com o objetivo de cobrir os aspectos envolvidos nesta abordagem, utilizando-se de aplicativos com cunho matemático, apresentando uma maneira conveniente para estudar efeitos quânticos. Entretanto, este trabalho tem um objetivo diferente, onde espera-se facilitar o aprendizado dos conceitos de computação quântica por parte dos programadores provenientes da computação clássica. A ideia de fornecer uma linguagem similar às já usadas na programação clássica, como o Java, devem encurtar a curva de aprendizado para este tipo de codificação, permitindo o foco direto nos aspectos distintos entre a abordagem clássica e quântica.

Além do fato de utilizarmos uma extensão para uma linguagem amplamente aceita pela indústria de software, também pode-se esperar uma melhor compreensão dos conceitos de mecânica quântica em geral através do uso da linguagem proposta, uma vez que é possível realizar a implementação de algoritmos quânticos expressos através de uma sintaxe e semântica bem definida e previamente conhecida por boa parte dos programadores. É óbvio que a utilização dos conceitos de computação quântica na programação apresenta diferenças significativas da computação atual, porém a possibilidade de encapsulamento de informações e rotinas quânticas em classes e bibliotecas pode facilitar a criação de algoritmos complexos, por reaproveitar código, e consequentemente o conhecimento já desenvolvido em outros algoritmos.

Com relação aos objetivos deste trabalho, pode-se afirmar que, de fato, é possível realizar a utilização de conceitos de linguagens funcionais, como *mônadas*, num contexto OO, e a semântica proposta foi embutida e implementada no FJ com sucesso, mantendo as características de segurança em se tratando dos tipos da linguagem Java original. Como ferramenta para

a implementação do interpretador, a linguagem *Haskell* mostrou-se adequada, pois as definições expressas através da semântica operacional são facilmente traduzidas para uma linguagem funcional. Infelizmente não é possível simular a computação quântica com estados envolvendo muitos *qubits* em computador clássico, e deste modo, apenas exemplos simples foram executados pelo interpretador proposto.

6.1 Trabalhos Futuros

- **Adequações sintáticas:** Para o processo de criação de estados quânticos na linguagem utilizou-se uma sintaxe monádica, muito aceita no paradigma funcional, porém pouco conhecida por parte dos programadores que utilizam orientação a objetos. Seria interessante possibilitar a construção dos estados mais diretamente através da utilização da notação de *Dirac* ou alguma variação que simplificasse a visualização direta da informação quântica contida no código-fonte.

- **Melhoria na escrita de código:** A semântica apresenta um operador de composição de computações, chamado *bind* que, a medida que são adicionadas operações, acaba-se perdendo a legibilidade do bloco. Uma maneira melhorar a forma de escrita do código é através da *do-notation* já utilizada nas linguagens funcionais, que deixam o código com um aspecto imperativo.

- **Adição da operação de medida:** Esta operação é crucial para o desenvolvimento de algoritmos quânticos, e é uma operação não determinística e portanto não reversível. A linguagem atual possibilita a escrita e pensamento sobre algoritmos quânticos, porém, para uma simulação completa é necessária esta operação.

- **Desenvolvimento de provas:** As definições semânticas apresentadas nos possibilitaram a implementação do interpretador e a realização de diversos testes que demonstram que *aparentemente* a semântica está correta. Como uma forma de garantir a corretude das regras, se faz necessária a realização de provas de cada uma delas, tanto das regras de avaliação quanto do sistema de tipos.

REFERÊNCIAS

ABRAMSKY, S. High-level methods for quantum computation and information. In: LOGIC IN COMPUTER SCIENCE, 2004. PROCEEDINGS OF THE 19TH ANNUAL IEEE SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.410–414.

ALTENKIRCH, T.; J, G. A Functional Quantum Programming Language. In: ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 20. **Proceedings...** [S.l.: s.n.], 2005.

Arrighi, P.; Dowek, G. Linear-algebraic Lambda-calculus: higher-order, encodings and confluence. **eprint arXiv:quant-ph/0612199**, [S.l.], Dec. 2006.

BARR, M.; WELLS, C. **Toposes, Triples and Theories**. 2005.

BELKHIR, W.; GIORGETTI, A. Lazy AC-Pattern Matching for Rewriting. In: INTERNATIONAL WORKSHOP ON REDUCTION STRATEGIES IN REWRITING AND PROGRAMMING, WRS 2011, NOVI SAD, SERBIA, 29 MAY 2011., 10. **Proceedings...** [S.l.: s.n.], 2011. p.37–51.

BELLIA, M.; OCCHIUTO, M. Java: proving type safety for Java simple closures. **CSP2010**, [S.l.], p.61–72, 2010.

BETTELLI, S.; SERAFINI, L.; CALARCO, T. Toward an architecture for quantum programming. **CoRR**, [S.l.], v.cs.PL/0103009, 2001.

CALEGARO, B. C.; VIZZOTTO, J. K. Quantum Monad Using Java Closures. In: THEORETICAL COMPUTER SCIENCE (WEIT), 2013 2ND WORKSHOP-SCHOOL ON. **Anais...** [S.l.: s.n.], 2013. p.34–39.

COECKE, B.; DUNCAN, R. Interacting Quantum Observables. In: AUTOMATA, LANGUAGES AND PROGRAMMING. **Anais...** Springer, 2008. p.298–310. (Lecture Notes in Computer Science, v.5126).

DEUTSCH, D. Quantum theory, the Church-Turing principle and the universal quantum computer. , [S.l.], v.400, p.97–117, 1985.

DIRAC, P. A. M. A new notation for quantum mechanics. **Mathematical Proceedings of the Cambridge Philosophical Society**, [S.l.], v.35, p.416–418, 7 1939.

FEYNMAN, R. Simulating physics with computers. **International Journal of Theoretical Physics**, [S.l.], v.21, p.467–488, 1982.

GRATTAGE, J. J. et al. A functional quantum programming language. In: IN: PROCEEDINGS OF THE 20TH ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE. **Anais...** [S.l.: s.n.], 2005. p.249–258.

GROVER, L. K. A fast quantum mechanical algorithm for database search. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 28. **Proceedings...** [S.l.: s.n.], 1996. p.212–219.

GRUSKA, J. **Quantum Computing**. [S.l.]: Mcgraw Hill Book, 2000.

IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight Java: a minimal core calculus for java and gj. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, [S.l.], v.23, n.3, p.396–450, 2001.

JÚNIOR, B. L.; LIMA, A. F. de. Circuitos Quânticos. In: 28., . **Anais...** [S.l.: s.n.], 2006. (WECIQ 2006 - Mini-Curso).

KNILL, E. **Conventions for Quantum Pseudocode**. 1996.

LINDLEY, S.; WADLER, P.; YALLOP, J. The Arrow Calculus (Functional Pearl). In: INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING. **Anais...** [S.l.: s.n.], 2008.

LIPOVACA, M. **Learn You a Haskell for Great Good!**: a beginner's guide. 1st.ed. San Francisco, CA, USA: No Starch Press, 2011.

MERMIN, N. D. **Quantum Computer Science**: an introduction. New York, USA: Cambridge University Press, 2007.

MOGGI, E. Computational Lambda-Calculus and Monads. In: **Anais...** IEEE Computer Society Press, 1988. p.14–23.

MU, S.-C.; BIRD, R. Functional Quantum Programming. In: ASIAN WORKSHOP ON PROGRAMMING LANGUAGES AND SYSTEMS, KAIST, Dajeaon, Korea. **Anais...** [S.l.: s.n.], 2001.

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

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

SANDERS, J. W.; ZULIANI, P. Quantum Programming. In: IN MATHEMATICS OF PROGRAM CONSTRUCTION. **Anais...** Springer-Verlag, 1999. p.80–99.

SELINGER, P. Towards a Quantum Programming Language. **Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages**, [S.l.], v.14, p.527–586, 2004.

SELINGER, P. A Brief Survey of Quantum Programming Languages. In: KAMEYAMA, Y.; STUCKEY, P. (Ed.). **Functional and Logic Programming**. [S.l.]: Springer Berlin Heidelberg, 2004. p.1–6. (Lecture Notes in Computer Science, v.2998).

SELINGER, P. Dagger Compact Closed Categories and Completely Positive Maps: (extended abstract). **Electronic Notes in Theoretical Computer Science**, [S.l.], v.170, n.0, p.139 – 163, 2007. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005).

SHOR, P. Progress in Quantum Algorithms. In: EVERITT, H. (Ed.). **Experimental Aspects of Quantum Computing**. [S.l.]: Springer US, 2005. p.5–13.

SIMON, D. R. On the Power of Quantum Computation. **SIAM Journal on Computing**, [S.l.], v.26, p.116–123, 1994.

T. D. LADD F. JELEZKO, R. L. Y. N. C. M.; O'BRIEN, J. L. Quantum Computers. **Nature Physics**, [S.l.], v.464/online, 2010.

TONDER, A. van. A Lambda calculus for quantum computation. **SIAM J.Comput.**, [S.l.], v.33, p.1109–1135, 2004.

VIZZOTTO, J. K. **Structuring General and Complete Quantum Computations in Haskell**: the arrows approach. 2006. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

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.; CALEGARO, B. C.; PIVETA, E. K. A Double Effect λ-calculus for Quantum Computation. In: DU BOIS, A.; TRINDER, P. (Ed.). **Programming Languages**. [S.l.]: Springer Berlin Heidelberg, 2013. p.61–74. (Lecture Notes in Computer Science, v.8129).

VIZZOTTO, J. K.; DUBOIS, A. R.; SABRY, A. Reasoning About General Quantum Programs over Mixed States. In: FORMAL METHODS: FOUNDATIONS AND APPLICATIONS. **Anais...** Springer, 2009. p.321–335. (Lecture Notes in Computer Science, v.5902).

YANOFSKY, N. S.; MANNUCCI, M. A. **Quantum Computing for Computer Scientists**. [S.l.]: Cambridge University Press, 2008.

YAO, A. C.-C. Quantum circuit complexity. In: FOUNDATIONS OF COMPUTER SCIENCE, 1993. PROCEEDINGS., 34TH ANNUAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 1993. p.352–361.

ÖMER, B. **A procedural formalism for quantum computing**. [S.l.: s.n.], 1998.

APÊNDICES

APÊNDICE A – Formalização da Linguagem Proposta

A.1 Sintaxe

Definição de Tipos

$T ::= T_{FJ} \mid T_G \mid T_P \mid T_T \mid T_F \mid T_Q$

$T_{FJ} ::= C$

$T_G ::= C\langle \bar{T} \rangle$

$T_P ::= \texttt{boolean} \mid \texttt{Complex}$

$T_T ::= \{T_i^{i \in 1..n}\}$

$T_F ::= (\bar{T}) \texttt{ -> } T$

$T_Q ::= \texttt{Vec}\langle T_B \rangle$

$T_B ::= \texttt{boolean} \quad \triangleright \text{ Tipos Base}$

$\quad \mid \{T_{Bi}^{i \in 1..n}\}$

Declarações de classes e classes genéricas

$CL ::= \texttt{class } C \texttt{ extends } C \texttt{ \{ } \bar{C}\ \bar{f}\texttt{; } K\ \bar{M} \texttt{ \}}$

$CL \mid \texttt{class } C\langle \bar{C} \texttt{ extends } \bar{C} \rangle \texttt{ extends } C \texttt{ \{ } \bar{C}\ \bar{f}\texttt{; } K\ \bar{M} \texttt{ \}}$

Declarações de construtores

$K ::= C(\bar{C}\ \bar{f})\ \{\ \texttt{super}(\bar{f})\texttt{; this.}\bar{f}=\bar{f}\texttt{; }\}$

Declarações de métodos

$M ::= C\ \texttt{m}(\bar{C}\ \bar{x})\ \{\ \texttt{return t; }\}$

Termos

$t ::= x$

$\mid t.f$

$\mid t.m(\bar{t})$

$\mid \texttt{new } C(\bar{t})$

$\mid \texttt{new } T_G(\bar{t})$

$\mid \texttt{(C) t}$

$\mid \texttt{true}$

$\mid \texttt{false}$

$\mid \texttt{if (t) \{ t \} else \{ t \}}$

$\mid \texttt{ComplexOne}$

$\mid \texttt{ComplexZero}$

$\mid \texttt{ComplexHalf}$

$\mid t + t$

$\mid t - t$

$\mid t \star t$

$\mid \texttt{let x = t in t}$

$\mid \{t_i^{i \in 1..n}\}$

$\mid t.i$

Termos (Cont.)

$\mid (\bar{T}\ \bar{x})\ \texttt{->}\ t$

$\mid t.\texttt{invoke}(\bar{t})$

$\mid \texttt{mzero}$

$\mid \texttt{mreturn } t$

$\mid t_1 \gg= t_2$

$\mid t_1\ \texttt{mplus}\ t_2$

$\mid t_1\ \$\star\ t_2$

Valores

$v ::= \texttt{new } C(\bar{v})$

$\mid \texttt{new } T_G(\bar{t})$

$\mid \texttt{true}$

$\mid \texttt{false}$

$\mid \texttt{ComplexOne}$

$\mid \texttt{ComplexZero}$

$\mid \texttt{ComplexHalf}$

$\mid \{v_i^{i \in 1..n}\}$

$\mid (\bar{T}\ \bar{x})\ \texttt{->}\ t$

$\mid \texttt{mzero}$

$\mid \texttt{mreturn } v$

A.2 Definições Auxiliares

Definições de subtipo

$$\frac{}{\mathrm{C} <: \mathrm{C}}$$

$$\frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{D} <: \mathrm{E}}{\mathrm{C} <: \mathrm{E}}$$

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{...\}}{\mathrm{C} <: \mathrm{D}}$$

Obtenção de atributos de uma classe

$$fields(\mathrm{Object}) = \bullet$$

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad fields(\mathrm{D}) = \bar{\mathrm{D}}\ \bar{\mathrm{g}}}{fields(\mathrm{C}) = \bar{\mathrm{D}}\ \bar{\mathrm{g}}, \bar{\mathrm{C}}\ \bar{\mathrm{f}}}$$

$$\frac{CT(\mathrm{C}) = \text{class C}\langle\bar{\mathrm{X}} \lhd \bar{\mathrm{N}}\rangle \text{ extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad fields([\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \mathrm{D}) = \bar{\mathrm{U}}\ \bar{\mathrm{g}}}{fields(\mathrm{C}\langle\bar{\mathrm{T}}\rangle) = \bar{\mathrm{U}}\ \bar{\mathrm{g}}, [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \bar{\mathrm{C}}\ \bar{\mathrm{f}}}$$

Obtenção do tipo de um método

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad \mathrm{B}\ \mathrm{m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}}{mtype(\mathrm{m}, \mathrm{C}) = \bar{\mathrm{B}} \to \mathrm{B}}$$

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad m \text{ is not defined in } \bar{\mathrm{M}}}{mtype(\mathrm{m}, \mathrm{C}) = mtype(\mathrm{m}, \mathrm{D})}$$

$$\frac{CT(\mathrm{C}) = \text{class C}\langle\bar{\mathrm{X}} \lhd \bar{\mathrm{N}}\rangle \text{ extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad \mathrm{B}\ \mathrm{m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}}{mtype(\mathrm{m}, \mathrm{C}\langle\bar{\mathrm{T}}\rangle) = [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ (\bar{\mathrm{B}} \to \mathrm{B})}$$

Obtenção do corpo de um método

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad \mathrm{B}\ \mathrm{m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}}{mbody(\mathrm{m},\mathrm{C}) = (\bar{\mathrm{x}}, \mathrm{t})}$$

$$\frac{CT(\mathrm{C}) = \text{class C extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad m \text{ não está definido em } \bar{\mathrm{M}}}{mbody(\mathrm{m},\mathrm{C}) = mbody(\mathrm{m}, \mathrm{D})}$$

$$\frac{CT(\mathrm{C}) = \text{class C}\langle\bar{\mathrm{X}} \lhd \bar{\mathrm{N}}\rangle \text{ extends D } \{\bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\} \quad \mathrm{B}\ \mathrm{m}(\bar{\mathrm{B}}\ \bar{\mathrm{x}})\ \{\ \text{return t;}\ \} \in \bar{\mathrm{M}}}{mbody(\mathrm{m},\mathrm{C}\langle\bar{\mathrm{T}}\rangle) = (\bar{\mathrm{x}}, [\bar{\mathrm{T}}/\bar{\mathrm{X}}]\ \mathrm{t})}$$

A.3 Sistema de Tipos

A.3.1 Formação da Tabela de Classes

Regras para classes bem definidas

$$\frac{\begin{array}{c}K = C(\bar{D}\ \bar{g}, \bar{C}\ \bar{f})\ \{\ super(\bar{g});\ this.\bar{f} = \bar{f};\ \} \\ \mathit{fields}(D) = \bar{D}\ \bar{g} \quad \bar{M}\ OK\ em\ C\end{array}}{class\ C\ extends\ D\ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\ \}\ \ OK}$$

Regras para métodos bem definidos:

$$\frac{\begin{array}{c}\bar{x}\text{: }\bar{C},\ this\text{: }C \vdash t_0\text{: }E_0 \quad E_0 <: C_0 \\ \mathit{CT}(C) = class\ C\ extends\ D\ \{\ ...\ \} \\ \mathit{override}(m, D, \bar{C} \to C_0)\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ return\ t_0;\ \}\ OK\ em\ C}$$

A.3.2 Tipagem dos Termos

Regras de tipo para os termos originais FJ

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad (\textit{Variáveis})$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad \mathit{fields}(C_0) = \bar{C}\ \bar{f}}{\Gamma \vdash t_0.f_i : C_i} \qquad (\textit{Acesso a Atributos})$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, C_0) = \bar{D} \to C \\ \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}\end{array}}{\Gamma \vdash t_0.m(\bar{t}) : C} \qquad (\textit{Invocação de Métodos})$$

$$\frac{\begin{array}{c}\mathit{fields}(C) = \bar{D}\ \bar{f} \\ \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}\end{array}}{\Gamma \vdash new\ C(\bar{t}) : C} \qquad (\textit{Criação de Objetos})$$

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)\ t_0 : C} \qquad (\textit{Upcast})$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)\ t_0 : C} \qquad (\textit{Downcast})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : D \quad C \not<: D \quad D \not< C \\ \textit{stupid warning}\end{array}}{\Gamma \vdash (C)\ t_0 : C} \qquad (\textit{Stupid Cast})$$

Regras de tipo para termos booleanos e condicionais

$$\frac{}{\text{true} : \text{boolean}} \qquad \frac{}{\text{false} : \text{boolean}} \qquad (\textit{Booleanos})$$

$$\frac{t_1 : \text{boolean} \quad t_2 : T \quad t_3 : T}{\text{if } (\ t_1\ )\ \{\ t_2\ \} \text{ else } \{\ t_3\ \} : T} \qquad (\textit{Condicionais})$$

Regras de tipo para números complexos e operações matemáticas

$$\frac{}{\Gamma \vdash \text{ComplexZero} : \mathbb{C}} \qquad \frac{}{\Gamma \vdash \text{ComplexOne} : \mathbb{C}} \qquad \frac{}{\Gamma \vdash \text{ComplexHalf} : \mathbb{C}}$$

$$\frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 + t_2} \qquad \frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 - t_2} \qquad \frac{\Gamma \vdash t_1, t_2 : \mathbb{C}}{\Gamma \vdash t_1 * t_2} \qquad (\textit{Operadores})$$

Regras de tipo para o termo let

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \qquad (\textit{Let})$$

Regras de tipo para tuplas

$$\frac{\text{para cada } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i^{\,i \in 1..n}\} : \{T_i^{\,i \in 1..n}\}} \qquad (\textit{Tupla})$$

$$\frac{\Gamma \vdash t_1 : \{T_i^{\,i \in 1..n}\}}{\Gamma \vdash t_1.j : T_j} \qquad (\textit{Projeção})$$

Regras de tipo para closures

$$\frac{\bar{T} \text{ OK} \qquad \Gamma, \bar{x} : \bar{T} \vdash t : T}{\Gamma \vdash (\bar{T}\ \bar{x}) \to t : (\bar{T} \to T)} \qquad (\textit{Closure bem definida})$$

$$\frac{\Gamma \vdash t : (\bar{T} \to T) \qquad \Gamma \vdash \bar{t} : \bar{S} \qquad \bar{S} <: \bar{T}}{\Gamma \vdash t.\text{invoke}(\bar{t}) : T} \qquad (\textit{Invocação da closure})$$

Regras de tipo para os construtores monádicos

$$\frac{}{\Gamma \vdash \text{mzero} : \text{Vec<}T_B\text{>}} \qquad (\textit{Monad zero})$$

$$\frac{\Gamma \vdash t : T_B}{\Gamma \vdash \text{mreturn } t : \text{Vec<}T_B\text{>}} \qquad (\textit{Monad return})$$

Regras de tipo para o operador bind

$$\frac{\Gamma \vdash t_1 : \text{Vec<}T_B\text{>} \qquad \Gamma \vdash t_2 : (T_B \to \text{Vec<}T_B\text{>})}{\Gamma \vdash t_1 \gg= t_2 : \text{Vec<}T_B\text{>}} \qquad (\textit{Operador bind})$$

Regras de tipo para os operadores monádicos

$$\frac{\Gamma \vdash t_1, t_2\text{: Vec<}T_B\text{>}}{\Gamma \vdash t_1 \text{ mplus } t_2 : \text{Vec<}T_B\text{>}} \quad (\textit{Monad Plus})$$

$$\frac{\Gamma \vdash t_1\text{: } \mathbb{C} \quad t_2\text{: Vec<}T_B\text{>}}{\Gamma \vdash t_1 \text{ \$* } t_2 : \text{Vec<}T_B\text{>}} \quad (\textit{Produto Escalar})$$

A.4 Avaliação Semântica

A.4.1 Regras de Redução

Regras de avaliação para os termos originais FJ

$$\frac{\textit{fields}(C) = \bar{C}\ \bar{f}}{\text{new } C(\bar{v}).f_i \rightarrow v_i} \quad (\textit{Acesso a Atributos})$$

$$\frac{\textit{mbody}(m, C) = (\bar{x}, t_0)}{\text{new } C(\bar{v}).m(\bar{u}) \rightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})]t_0} \quad (\textit{Invocação de Métodos})$$

$$\frac{C <: D}{(D)\ (\text{new } C(\bar{v})) \rightarrow \text{new } C(\bar{v})} \quad (\textit{Casts})$$

Regras de avaliação para termos condicionais

$$\frac{}{\text{if (true) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \rightarrow t_2} \quad (\textit{Condição True})$$

$$\frac{}{\text{if (false) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \rightarrow t_3} \quad (\textit{Condição False})$$

Regras de avaliação para o operador let

$$\frac{}{\text{let } x = v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]\ t_2} \quad (\textit{Operador Let})$$

Regras de avaliação para tuplas

$$\frac{}{\{v_i{}^{i \in 1..n}\}.j \rightarrow v_j} \quad (\textit{Projeção})$$

Regras de avaliação para closures

$$\frac{}{((\bar{T}\ \bar{x}) \rightarrow T)\ \ t.\text{invoke}(\bar{d}) \longrightarrow [\bar{d} \mapsto \bar{x}]\ t} \quad (\textit{Invocação de closures})$$

Regras de avaliação para o operador bind

$$\frac{}{(\text{mreturn } v_1) \gg\!= t_2 \longrightarrow t_2.\text{invoke}(v_1)} \quad (\textit{Aplicação construtor})$$

$$\frac{}{(t_1 \text{ mplus } t_2) \gg\!= t_3 \longrightarrow (t_1 \gg\!= t_3) \text{ mplus } (t_2 \gg\!= t_3)} \quad (\textit{Aplicação mplus})$$

$$\frac{}{(\alpha \text{ \$* } t_1) \gg\!= t_2 \longrightarrow \alpha \text{ \$* } (t_1 \gg\!= t_2)} \quad (\textit{Aplicação produto escalar})$$

Regras de avaliação para os operadores monádicos

$$\frac{}{\text{mzero mplus } v_1 \longrightarrow v_1} \quad (\textit{Monad Zero Plus})$$

$$\frac{}{v_1 \text{ mplus mzero} \longrightarrow v_1} \quad (\textit{Monad Plus Zero})$$

$$\frac{}{\text{ComplexZero \$* t} \longrightarrow \text{mzero}} \quad (\textit{Produto escalar zero})$$

$$\frac{}{\alpha \text{ \$* } (t_1 \text{ mplus } t_2) \longrightarrow \alpha \text{ \$* } t_1 \text{ mplus } \alpha \text{ \$* } t_2} \quad (\textit{Produto escalar mplus dist})$$

$$\frac{}{\alpha_1 \text{ \$* t mplus } \alpha_2 \text{ \$* t} \longrightarrow (\alpha_1 + \alpha_2) \text{ \$* t}} \quad (\textit{Produto escalar mplus})$$

A.4.2 Regras de Congruência

Regras de congruência para os termos originais FJ

$$\frac{t_0 \to t_0'}{t_0.f \to t_0'.f} \quad (\textit{Acesso a atributos})$$

$$\frac{t_0 \to t_0'}{t_0.m(\bar{t}) \to t_0'.m(\bar{t})} \quad (\textit{Invocação de métodos})$$

$$\frac{t_i \to t_i'}{v_0.m(\bar{v}, t_i, \bar{t}) \to v_0.m(\bar{v}, t_i', \bar{t})} \quad (\textit{Parâmetros dos métodos})$$

$$\frac{t_i \to t_i'}{\text{new } C(\bar{v}, t_i, \bar{t}) \to \text{new } C(\bar{v}, t_i', \bar{t})} \quad (\textit{Criação de objetos})$$

$$\frac{t_0 \to t_0'}{(C)\ t_0 \to (C)\ t_0'} \quad (\textit{Casts})$$

Regras de congruência para termos condicionais

$$\frac{t_1 \to t_1'}{\text{if (} t_1 \text{) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \to \text{if (} t_1' \text{) \{ } t_2 \text{ \} else \{ } t_3 \text{ \}}} \quad (\textit{Condicionais})$$

Regras de congruência para operadores matemáticos

$$\frac{t_1 \to t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2} \qquad \frac{t_2 \to t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'} \quad (\textit{Operador soma})$$

$$\frac{t_1 \to t_1'}{t_1 - t_2 \longrightarrow t_1' - t_2} \qquad \frac{t_2 \to t_2'}{v_1 - t_2 \longrightarrow v_1 - t_2'} \quad (\textit{Operador subtração})$$

$$\frac{t_1 \to t_1'}{t_1 * t_2 \longrightarrow t_1' * t_2} \qquad \frac{t_2 \to t_2'}{v_1 * t_2 \longrightarrow v_1 * t_2'} \quad (\textit{Operador multiplicação})$$

Regras de congruência para o operador let

$$\frac{t_1 \to t_1'}{\text{let x = } t_1 \text{ in } t_2 \longrightarrow \text{let x = } t_1' \text{ in } t_2} \quad (\textit{Operador Let})$$

Regras de congruência para tuplas

$$\frac{t_1 \to t_1'}{t_1.i \to t_1'.i} \quad (\textit{Projeção})$$

$$\frac{t_j \to t_j'}{\{{v_i}^{i \in 1..j-1}, t_j, {t_k}^{k \in j+1..n}\} \to \{{v_i}^{i \in 1..j-1}, t_j', {t_k}^{k \in j+1..n}\}} \quad (\textit{Tupla})$$

Regras de congruência para closures

$$\frac{t \longrightarrow t'}{t.invoke(\overline{t}) \longrightarrow t'.invoke(\overline{t})} \quad (\textit{Invocação de closures})$$

$$\frac{t \longrightarrow t'}{t.invoke(...,t_i,...) \longrightarrow t.invoke(...,t'_i,...)} \quad (\textit{Parâmetros das closures})$$

Regras de congruência para o construtor monádico

$$\frac{t_1 \rightarrow t'_1}{mreturn\ t_1 \rightarrow mreturn\ t'_1} \quad (\textit{Construtor monádico})$$

Regras de congruência para o operador bind

$$\frac{t_1 \rightarrow t'_1}{t_1 \gg\!= t_2 \longrightarrow t'_1 \gg\!= t_2} \quad (\textit{Bind esquerda})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \gg\!= t_2 \longrightarrow v_1 \gg\!= t'_2} \quad (\textit{Bind direita})$$

Regras de congruência para o operador mplus

$$\frac{t_1 \rightarrow t'_1}{t_1\ mplus\ t_2 \longrightarrow t'_1\ mplus\ t_2} \quad (\textit{MPlus esquerda})$$

$$\frac{t_2 \rightarrow t'_2}{v_1\ mplus\ t_2 \longrightarrow v_1\ mplus\ t'_2} \quad (\textit{MPlus direita})$$

Regras de congruência para o produto escalar

$$\frac{t_1 \rightarrow t'_1}{t_1\ \$*\ t_2 \longrightarrow t'_1\ \$*\ t_2} \quad (\textit{Produto escalar esquerda})$$

$$\frac{t_2 \rightarrow t'_2}{v_1\ \$*\ t_2 \longrightarrow v_1\ \$*\ t'_2} \quad (\textit{Produto escalar direita})$$

APÊNDICE B – Gramática BNF e Construtores ASTs

B.1 Gramática BNF para a Linguagem Proposta

```
1  -- Codigos do programa
2  Program: ClassList Term ';'                                  { Program $1 $2 }
3
4  -- Lista de classes
5  ClassList: ClassDef                                          { [$1] }
6           | ClassList ClassDef                                { $2 : $1 }
7
8  -- Definicao de classe
9  ClassDef: class ClassName GenericDefList extends ClassName
10          '{' AttrList ConstrDef MethodList '}'
11          { ($2,ClassDef $2 $3 $5 $7 $8 $9) }
12
13 -- Definicao de construtor
14 ConstrDef: ClassName '(' ParamList ')'
15            '{' super '(' ArgList ')' ';' AttrAssignList '}'
16            { ConstrDef $1 $3 $8 $11 }
17
18 -- Lista de implementacao de metodos
19 MethodList: {- empty -}                                     { [] }
20           | MethodList MethodDef                           { $2 : $1 }
21
22 -- Definicao de metodos
23 MethodDef: TypeDef name '(' ParamList ')'
24            '{' return Term ';' '}'      { MethodDef $1 $2 $4 $8 }
25
26 -- Lista de declaracao de parametros para construtores e funcoes
27 -- Parametros Formais
28 ParamList: {- empty -}                                      { [] }
29            | IdentStmt                                     { [$1] }
30            | ParamList ',' IdentStmt                       { $3 : $1 }
31
32 -- Lista de argumentos utilizados para passar informacoes nas
33 -- chamadas de funcoes
34 ArgList: {- empty -}                                        { [] }
35          | name                                            { [$1] }
36          | ArgList ',' name                                { $3 : $1 }
37
38 -- Lista de atributos de classe (declaracao)
39 AttrList: {- empty -}                                      { [] }
40         | AttrList IdentStmt ';'                          { $2 : $1 }
41
42 -- Lista de atribuicao interna de atributos (utilizada no construtor)
43 AttrAssignList: {- empty -}                               { [] }
44               | AttrAssign                                { [$1] }
45               | AttrAssignList AttrAssign                 { $2 : $1 }
46
47 -- Atribuicao interna de atributos
48 AttrAssign: this '.' name '=' name ';'                     { ($3,$5) }
49
50 -- Definicao de atributos ou parametros
51 IdentStmt: TypeDef name                                    { ($1,$2) }
```

```
52
53  -- Definicao de tipos
54  TypeDef: boolean                                    { TypeBool }
55         | Complex                                    { TypeComplex }
56         | ClassName GenericList                      { TypeClass $1 $2 }
57         | '(' TypeList "->" TypeDef ')'              { TypeClosure $4 $2 }
58         | '{' TypeList '}'                           { TypeTuple $2 }
59         | Vec '<' TypeDef '>'                        { TypeQuantum $3 }
60
61  -- Lista de tipos utilizados para as Closures
62  TypeList: {- empty -}                               { [] }
63                 | TypeDef                                    { [$1] }
64                 | TypeList ',' TypeDef                       { $3 : $1 }
65
66  -- Definicao de tipos genericos
67  GenericDefList: {- empty -}                         { [] }
68                | '<' ExtendsList '>'                 { $2 }
69
70  ExtendsList: ClassName extends ClassName            { [($1,$3)] }
71             | ExtendsList ','
72               ClassName extends ClassName            { ($3,$5) : $1 }
73
74  GenericList: {- empty -}                            { [] }
75             | '<' TypeList '>'                       { $2 }
76
77  -- Nomes de classes
78  ClassName: Object                                   { "Object" }
79           | name                                     { $1 }
80
81  -- Lista de termos
82  TermList: {- empty -}                               { [] }
83          | Term                                      { [$1] }
84          | TermList ',' Term                         { $3 : $1 }
85
86  -- Termos (utilizados no corpo dos metodos)
87  Term : BooleanLiteral                              { BooleanLiteral $1 }
88       | ComplexLiteral                              { ComplexLiteral $1 }
89       | name                                           { Var $1 }
90       | this '.' name                               { ThisAccessAttr $3 }
91       | this '.' name '(' TermList ')'              { ThisAccessMeth $3 $5 }
92       | Term '.' name                               { AttrAccess $1 $3 }
93       | Term '.' name '(' TermList ')'              { MethodAccess $1 $3 $5 }
94       | new ClassName GenericList '(' TermList ')' { CreateObject $2 $3 $5 }
95       | '(' ClassName ')' Term                      { Cast $2 $4 }
96       | if '(' Term ')' '{'
97             Term
98         '}' else '{'
99             Term '}'                                { If $3 $6 $10 }
100      | let name '=' Term in Term                   { Let $2 $4 $6 }
101      | '(' ParamList ')' "->" Term                 { ClosureDef $2 $5 }
102      | '(' Term ')' '.' invoke '(' TermList ')'    { InvokeClosure $2 $7 }
103      | '{' TermList '}'                            { Tuple $2 }
104      | Term '.' number                             { TupleAccess $1 $3 }
105      | Term '*' Term                               { ComplexTimes $1 $3 }
106      | Term '+' Term                               { ComplexPlus $1 $3 }
107      | Term '-' Term                               { ComplexMinus $1 $3 }
108      | Term "==" Term                              { RelEquals $1 $3 }
109      | mzero                                       { MonadZero }
```

```
110       | mreturn Term                                  { MonadReturn $2 }
111       | '(' Term ')' ">>=" Term                       { MonadBind $2 $5 }
112       | '(' Term ')' mplus '(' Term ')'               { MonadPlus $2 $6 }
113       | Term "$*" Term                                { ScalarProduct $1 $3 }
114
115   -- Booleanos
116   BooleanLiteral: true                                { BLTrue }
117                 | false                               { BLFalse }
118
119   -- Numeros Complexos
120
121   ComplexLiteral: ComplexZero                         { ComplexNumber 0.0 }
122                 | ComplexOne                          { ComplexNumber 1.0 }
123                 | ComplexHalf                         { ComplexHalf }
```

B.2 Construtores das ASTs

```
1   data Program = Program [(String,ClassDef)] Term
2                deriving (Show, Eq)
3
4   data ClassDef = ClassDef String [(String,String)] String [(Type,String)]
5                   ConstrDef [MethodDef]
6                 deriving (Show, Eq)
7
8   data ConstrDef = ConstrDef String [(Type,String)] [String] [(String,String)]
9                  deriving (Show, Eq)
10
11  data MethodDef = MethodDef Type String [(Type,String)] Term
12                 deriving (Show, Eq)
13
14  data Term = EmptyTerm
15            | BooleanLiteral BooleanLiteral
16            | ComplexLiteral ComplexLiteral
17            | Var String
18            | ThisAccessAttr String
19            | ThisAccessMeth String [Term]
20            | AttrAccess Term String
21            | MethodAccess Term String [Term]
22            | CreateObject String [Type] [Term]
23            | Cast String Term
24            | If Term Term Term
25            | Let String Term Term
26            | ClosureDef [(Type,String)] Term
27            | InvokeClosure Term [Term]
28            | Tuple [Term]
29            | TupleAccess Term Int
30            | ComplexTimes Term Term
31            | ComplexPlus Term Term
32            | ComplexMinus Term Term
33            | RelEquals Term Term
34            | MonadZero
35            | MonadReturn Term
36            | MonadBind Term Term
37            | MonadPlus Term Term
38            | ScalarProduct Term Term
39           deriving (Show, Eq)
40
```

```
41  data BooleanLiteral = BLTrue
42                      | BLFalse
43                      deriving (Show, Eq)
44
45  data ComplexLiteral = ComplexHalf
46                      | ComplexNumber (Complex Double)
47                      deriving (Show, Eq)
48
49  data Type = TypeBool
50            | TypeComplex
51            | TypeClass String [Type]
52            | TypeClosure Type [Type]
53            | TypeTuple [Type]
54            | TypeQuantum Type
55           deriving (Show, Eq)
```