

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Eric Tomás Zancanaro

**UM INTERPRETADOR E *TYPECHECKER* PARA UMA LINGUAGEM
REVERSÍVEL COM *PATTERN-MATCHING* SIMÉTRICO E CONTROLE
QUÂNTICO**

Santa Maria, RS
2018

Eric Tomás Zancanaro

**UM INTERPRETADOR E *TYPECHECKER* PARA UMA LINGUAGEM REVERSÍVEL
COM *PATTERN-MATCHING* SIMÉTRICO E CONTROLE QUÂNTICO**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, 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: Prof.^a Juliana Kaizer Vizzotto

Santa Maria, RS
2018

Zancanaro, Eric Tomás

Um Interpretador e Typechecker para uma Linguagem Reversível com Pattern-matching Simétrico e Controle Quântico / Eric Tomás Zancanaro.- 2018.

91 p.; 30 cm

Orientadora: Juliana Kaizer Vizzotto

Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação, RS, 2018

1. Linguagens de Programação 2. Sistema de Tipos 3. Computação Reversível 4. Computação Quântica I. Vizzotto, Juliana Kaizer II. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

©2018

Todos os direitos autorais reservados a Eric Tomás Zancanaro. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.

Endereço: BR 287, n. 7000

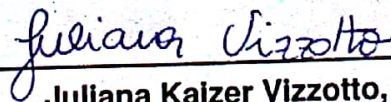
End. Eletr.: ezancanaro@inf.ufsm.br

Eric Tomás Zancanaro

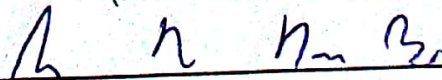
**UM INTERPRETADOR E TYPECHECKER PARA UMA LINGUAGEM REVERSÍVEL
COM PATTERN-MATCHING SIMÉTRICO E CONTROLE QUÂNTICO**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, 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.**

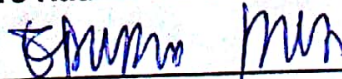
Aprovado em 13 de julho de 2018:



Juliana Kaizer Vizzotto, Dr^a. (UFSM)
(Presidenta/Orientadora)



André Rauber Du Bois, Dr. (UFPEL)



Eduardo Kessler Piveta, Dr. (UFSM)

*Every problem that is interesting is also
soluble.*

(David Deutsch)

RESUMO

UM INTERPRETADOR E *TYPECHECKER* PARA UMA LINGUAGEM REVERSÍVEL COM *PATTERN-MATCHING* SIMÉTRICO E CONTROLE QUÂNTICO

AUTOR: Eric Tomás Zancanaro
ORIENTADORA: Juliana Kaizer Vizzotto

Com o objetivo de superar os limites teóricos impostos pelo custo energético do modelo de computação clássico, novos paradigmas de computação foram formalizados. Enquanto a computação reversível busca conciliar a reversibilidade dos modelos físicos com a irreversibilidade computacional, a computação quântica visa a obtenção de desempenho através da manipulação de propriedades microscópicas da matéria. O desenvolvimento teórico é acompanhado da formalização de linguagens que codifiquem essas propriedades de forma independente da implementação física. Essa independência permite que programas nesses paradigmas sejam simulados através da construção de interpretadores. Esta dissertação apresenta a implementação de um interpretador e *typechecker* para uma linguagem de programação reversível com controle quântico. O projeto implementado na linguagem Haskell possibilita a verificação de tipos e a simulação ineficiente de sistemas quânticos, sendo capaz de executar algoritmos na direção usual e inversa. O texto apresenta também a adaptação e execução de algoritmos que comprovam a codificação do controle quântico na semântica da linguagem funcional utilizada como base para este trabalho.

Palavras-chave: Interpretador. Linguagens Funcionais. Computação Reversível. Computação Quântica. Controle Quântico.

ABSTRACT

AN INTERPRETER AND TYPECHECKER FOR A REVERSIBLE LANGUAGE WITH SYMMETRICAL PATTERN-MATCHING AND QUANTUM CONTROL

AUTHOR: Eric Tomás Zancanaro

ADVISOR: Juliana Kaizer Vizzotto

Aiming to overcome the theoretical limits imposed by the energetic cost of the classic computational model, new computing paradigms have been formalized. While reversible computing aims to reconcile the reversibility of the physical models with the irreversible way of modelling computations, quantum computing looks to harness the microscopical properties of matter to increase performance. The theoretical development is coupled with the formalization of programming languages that embed those properties independently of their physical implementation. This decoupling allows the simulation of programs belonging to these paradigms by the creation of interpreters. This dissertation presents the implementation of an interpreter and typechecker for a reversible language with quantum control. The project was implemented in the Haskell functional language, allowing for the typechecking and inefficient simulation of quantum systems, being capable of both forwards and backwards execution. The text also presents the adaptation and execution of algorithms that showcase the embedding of quantum control in the functional language chosen as basis of this work.

Keywords: Interpreter. Functional Languages. Reversible Computing. Quantum Computing. Quantum Control.

LISTA DE FIGURAS

Figura 2.1 – Gramática do Cálculo Lambda.	13
Figura 2.2 – Uma árvore de sintaxe abstrata	14
Figura 2.3 – Semântica operacional do cálculo lambda.	16
Figura 2.4 – Avaliação através da semântica operacional.	16
Figura 2.5 – Utilização de variáveis em linguagens de programação.	20
Figura 2.6 – Definição da função de Fibonacci através de <i>pattern-matching</i>	21
Figura 2.7 – Definição da função recursiva map.	22
Figura 2.8 – Exemplos de regras de tipos.	24
Figura 2.9 – Extensão e teste do conteúdo do contexto de tipos.	25
Figura 2.10 – Exemplos de operadores isomórficos.	29
Figura 2.11 – Derivação simplificada do passo 3 do algoritmo de Deutsch-Jozsa.	34
Figura 3.1 – Gramática de uma linguagem reversível com controle quântico.	38
Figura 3.2 – Regras de tipos para valores.	40
Figura 3.3 – Regras de tipos para termos.	41
Figura 3.4 – Regras de <i>pattern-matching</i>	42
Figura 3.5 – Contextos de avaliação.	43
Figura 3.6 – Semântica Operacional	43
Figura 3.7 – Propriedades algébricas das combinações lineares de termos.	44
Figura 3.8 – Regra de tipos para cláusulas com combinações lineares.	45
Figura 3.9 – Representação de matrizes unitárias através de isos.	45
Figura 3.10 – Definições de cláusulas com e sem combinações lineares.	45
Figura 3.11 – Regra de avaliação do operador de ponto fixo.	47
Figura 4.1 – Declaração de funções em Haskell.	49
Figura 4.2 – Exemplo de composição de funções.	50
Figura 4.3 – Uso do operador \$.	50
Figura 4.4 – Exemplo de uso da mônada State.	51
Figura 4.5 – Definição dos Tipos Abstratos da linguagem.	52
Figura 4.6 – Função para tratamento de erros.	55
Figura 4.7 – Teste de correspondência de tipos.	55
Figura 4.8 – Extração das amplitudes de uma combinação linear.	56
Figura 4.9 – Verificação de tipos para variáveis de valores.	56
Figura 4.10 – Implementação da verificação de tipos em termos.	57
Figura 4.11 – Diferenciação entre tipos de soma e o tipo recursivo de listas.	57
Figura 4.12 – Verificação do tipo para a aplicação de isomorfismos.	58
Figura 4.13 – Verificação de tipos das cláusulas de isomorfismos.	59
Figura 4.14 – Verificação da unitariedade das transformações lineares.	60
Figura 4.15 – Fluxo da avaliação de um termo $\Omega f t$	60
Figura 4.16 – Implementação do contexto de avaliação.	60
Figura 4.17 – Redução da aplicação de um valor a um isomorfismo.	61
Figura 4.18 – Processo para a avaliação de combinações lineares aplicadas a isomorfismos.	62
Figura 4.19 – Implementação do <i>pattern-matching</i> em um conjunto de cláusulas e um valor.	62
Figura 4.20 – Fragmento de código da substituição em isomorfismos.	63
Figura 4.21 – Função para expandir a recursão em um isomorfismo finito.	63

Figura 4.22 – Inversão simplificada das cláusulas de um isomorfismo.	64
Figura 4.23 – Implementação da inversão de cláusulas considerando transformações lineares.	66
Figura 4.24 – Iso codificando o algoritmo de Deutsch.	67
Figura 4.25 – Iso generalizando um Oráculo reversível.	67
Figura 4.26 – Iso codificando uma operação em 2 qubits, aplicando a transformada de Hadamard no primeiro e a função identidade no segundo.	67
Figura 4.27 – Iso generalizando um Oráculo reversível.	68
Figura 4.28 – Inversão do isomorfismo representando o algoritmo de Deutsch.	70
Figura 4.29 – Iso recursiva atuando em uma lista de qubits.	70
Figura 4.30 – Isomorfismo codificando o algoritmo de Deutsch-Jozsa.	71
Figura 4.31 – Isomorfismo codificando um passo único de um <i>quantum-walk</i>	71
Figura 4.32 – Demonstração de um passo de avaliação em uma recursão via ponto fixo.	73
Figura 4.33 – Inversão do isomorfismo X2.	76
Figura 4.34 – Transformação linear para um <i>quantum-walk</i>	77
Figura 4.35 – <i>Quantum walk</i> bidirecionalmente recursivo.	77
Figura 4.36 – <i>Quantum-walk</i> recursivo descrito por Ying.	77
Figura A.1 – Isomorfismos implementando Hadamard e o algoritmo de Deutsch.	86
Figura A.2 – Implementação do isomorfismo <i>next</i> para inteiros de 5 bits.	87
Figura A.3 – Definição do <i>quantum-walk</i> recursivo de Ying.	88
Figura B.1 – Ilustração da implementação atual das propriedades distributivas.	90
Figura B.2 – Ilustração de uma avaliação alternativa para as propriedades distributivas.	91

LISTA DE TABELAS

Tabela 2.1 – Avaliação <i>Call-by-name</i>	15
Tabela 2.2 – Avaliação <i>Call-by-value</i>	15
Tabela 2.3 – Contextos de Avaliação	17
Tabela 2.4 – Avaliação do operador de ponto fixo.....	18
Tabela 2.5 – Avaliação do operador de ponto fixo 2.....	19
Tabela 2.6 – Tabela verdade de nand com 2 bits.....	26
Tabela 2.7 – Tabela verdade de $nand^R$	28

SUMÁRIO

1	INTRODUÇÃO	9
2	CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL	12
2.1	SINTAXE E SEMÂNTICA	12
2.2	SISTEMAS DE TIPOS	22
2.3	LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL REVERSÍVEL	25
2.4	LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL QUÂNTICA	30
3	UMA LINGUAGEM FUNCIONAL REVERSÍVEL COM CONTROLE QUÂNTICO	37
3.1	PATTERN-MATCHING SIMÉTRICO	37
3.2	SISTEMA DE TIPOS	40
3.3	SEMÂNTICA DA LINGUAGEM	42
3.4	COMBINAÇÕES LINEARES PARA O CONTROLE QUÂNTICO	43
3.5	RECURSIVIDADE VIA <i>PONTO FIXO</i>	46
4	IMPLEMENTAÇÃO DO INTERPRETADOR E <i>TYPECHECKER</i>	48
4.1	HASKELL	48
4.2	TIPOS ABSTRATOS DE DADOS	52
4.3	FUNÇÕES UTILITÁRIAS	54
4.4	IMPLEMENTAÇÃO DO <i>TYPECHECKER</i>	55
4.5	AVALIAÇÃO	59
4.6	EXEMPLOS DE CONTROLE QUÂNTICO	65
4.6.1	Exemplo 1 - Algoritmo de Deutsch	66
4.6.2	Exemplo 2 - <i>Quantum-walks</i> recursivos	70
5	DISCUSSÃO FINAL	80
5.1	TRABALHOS RELACIONADOS	80
5.2	TRABALHOS FUTUROS	81
	REFERÊNCIAS BIBLIOGRÁFICAS	83
	APÊNDICE A – IMPLEMENTAÇÃO DOS EXEMPLOS VIA SINTAXE ABSTRATA	86
	APÊNDICE B – PROPRIEDADES DISTRIBUTIVAS EM EXPRESSÕES LET	89

1 INTRODUÇÃO

O modelo clássico de computação abstrata diverge das propriedades físicas dos sistemas que o implementam. Enquanto o primeiro é construído como um processo intrinsecamente irreversível, os modelos microscópicos da física apresentam interações estritamente reversíveis (TOFFOLI, 1980). Essa divergência implica em um custo mínimo de energia para computações realizadas em um sistema irreversível (LANDAUER, 1961), impondo limitações teóricas para o avanço do desempenho computacional baseadas na eficiência energética dos computadores modernos (FRANK, 2005). Novos paradigmas como a computação reversível e a computação quântica buscam reconciliar o modelo de computação abstrata com as propriedades físicas, buscando evitar tais limitações.

A computação reversível é um paradigma capaz de trazer benefícios de eficiência energética, explorados pela relação de entropia entre o ambiente e a informação codificada nos circuitos (SHANNON, 2001). Na computação clássica, operações irreversíveis incorrem no descarte de informação, fazendo com que a entropia armazenada seja transferida para o ambiente através da dissipação de energia na forma de calor (LANDAUER, 1961). Um modelo computacional construído sem a presença de operações irreversíveis, por sua vez, evita o descarte de informação, reduzindo o custo energético associado ao sistema como um todo (FRANK, 2005).

Um segundo paradigma de computação desenvolvido na busca de tirar proveito de propriedades microscópicas da matéria é o paradigma da computação quântica (AARONSON, 2013). O desenvolvimento da computação quântica canaliza propriedades de paralelismo e interferência quântica na especificação de algoritmos eficientes para um conjunto limitado de problemas altamente complexos, tais como o algoritmo de busca de Grover (GROVER, 1997) e o algoritmo de fatoração de Shor (SHOR, 1999).

Do lado teórico, o desenvolvimento de novos modelos de computação é acompanhado pela criação de representações abstratas para os mesmos. Linguagens de computação formalizam modelos teóricos de computação, abstraindo detalhes de implementação em representações mais próximas da linguagem natural, facilitando o raciocínio humano e simplificando a criação de provas para as propriedades do modelo (HUDAK, 1989). Por sua natureza abstrata, linguagens de programação são independentes do modelo físico que as implementa (SEBESTA, 2009), fazendo com que a criação de sistemas capazes de simular o comportamento desses novos paradigmas permita o desenvolvimento e teste empírico de novos algoritmos, sustentados pelo modelo formal.

Através da exploração de alguns conceitos básicos de linguagens de programação, seguido da exposição inicial dos paradigmas da computação quântica e reversível, este

trabalho descreve a criação de um interpretador para uma linguagem de programação reversível capaz de codificar controle de fluxo quântico. O interpretador é complementado pela especificação de um *typechecker* que garante a consistência dos programas especificados na linguagem escolhida como base para a implementação.

O objetivo central deste trabalho é ilustrar a semântica de uma linguagem de programação reversível com controle quântico de forma prática, fornecendo um ambiente para a especificação e simulação do comportamento de programas reversíveis e quânticos. Este objetivo deve ser alcançado através da construção de mecanismos de verificação de tipos e de avaliação. Através dessa implementação, objetiva-se especificar exemplos executáveis, visando compreender e demonstrar empiricamente a codificação de fenômenos como o paralelismo e a interferência quântica. Além disso, a implementação é utilizada para demonstrar empiricamente a expressão do controle quântico, em especial da recursão quântica. Para tal, exemplos de recursão quântica apresentados em um dos livros mais atuais sobre o assunto ((YING, 2016)), são adaptados para o ambiente desenvolvido, expondo a relação análoga do comportamento semântico descrito nos algoritmos de Ying, com o comportamento dado pela linguagem escolhida como base desta implementação.

Este trabalho não trata das preocupações em gerar uma interpretação eficiente para a simulação de estados quânticos, escolhendo por focar na codificação das propriedades semânticas de uma linguagem com controle quântico. A execução dos algoritmos aqui descrita é ineficiente tanto em termos de tempo computacional, quanto em termos de uso de memória. O custo de performance é devido, em sua maior parte, a implementação numérica escolhida para representar as amplitudes de termos e combinações lineares. Sugestões para melhorar a eficiência do interpretador são incluídas no capítulo final deste trabalho.

O interpretador é construído utilizando a linguagem Haskell. Um fator determinante na escolha da linguagem é a proximidade da sintaxe funcional com as descrições formais dos sistemas, simplificando a especificação de funções que espelhem as regras semânticas. Similarmente, a inexistência de efeitos colaterais em funções simplifica a verificação do comportamento do sistema e permite a rápida prototipação de funcionalidades da linguagem. A avaliação é realizada por um modelo sequencial, invocando as regras de avaliação recursivamente para alcançar uma forma normal dos valores. Essa escolha de projeto faz com que o modelo atual não suporte uma execução passo a passo da avaliação de programas reversíveis, possibilitando apenas a avaliação total de um termo inicial a um valor.

É de interesse que a implementação fornecida possibilite o desenvolvimento de algoritmos quânticos e reversíveis de forma empírica, contribuindo para o crescimento constante desses paradigmas. Espera-se que a exposição dos paradigmas, acompanhada pela tradução dos formalismos teóricos de uma linguagem em um programa funcional sirva para atizar o interesse na área, motivando a criação de mais ferramentas desse tipo.

Na sequência, descreve-se a estrutura adotada para o texto da presente dissertação. O Capítulo 2 apresenta noções básicas sobre linguagens de programação funcionais, iniciando por uma exploração da base teórica desse paradigma, representada pelo cálculo lambda, finalizando com uma breve introdução dos paradigmas reversível e quântico, acompanhada pela apresentação de linguagens de programação que os adotam.

No Capítulo 3, a linguagem de programação adotada como base deste trabalho é introduzida através da exposição das regras de tipos, análise da semântica da linguagem e a codificação da reversibilidade e do controle quântico.

O Capítulo 4 trata da implementação do interpretador, apresentando os módulos gerados e ilustrando as adaptações necessárias para a implementação do projeto. Esse capítulo é finalizado com a apresentação de exemplos de algoritmos com controle quântico, cuja manifestação na semântica da linguagem é ilustrada via a descrição dos passos de avaliação.

Uma análise dos resultados do projeto, relacionando-o com trabalhos similares na área é realizada na conclusão. São também apresentadas sugestões de trabalhos futuros e de melhorias para a implementação demonstrada.

2 CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL

Linguagens de programação definidas de acordo com o paradigma funcional permitem a especificação de programas através da definição e composição de funções. A computação nesse paradigma é realizada através da aplicação de argumentos de entrada a um conjunto de funções compostas (HUGHES, 1989).

Este capítulo introduz conceitos e notações relacionados ao paradigma da programação funcional, objetivando contextualizar o projeto da linguagem base utilizada para a implementação de um interpretador, resultado do presente trabalho. Visa também justificar a importância da inclusão de um sistema de tipos e, por consequência, da construção de um *typechecker*.

Essencialmente, o capítulo apresenta uma discussão sobre os mecanismos utilizados na especificação de linguagens funcionais. Ao longo do capítulo serão apresentadas noções sobre os fundamentos de sintaxe e semântica funcional sob a luz do formalismo do cálculo lambda. Finalmente, uma seção será dedicada à exposição dos sistemas de tipos, sua importância para o projeto de linguagens e sua base teórica.

2.1 SINTAXE E SEMÂNTICA

Linguagens funcionais tem sua origem no modelo formal de computação estabelecido pelo cálculo lambda de Alonzo Church. A proposta de Church gerou uma linguagem de programação simplificada na qual todas as computações são reduzidas às operações de definição e aplicação de funções (PIERCE, 2002). Explorar a sintaxe e semântica do cálculo Lambda é uma boa maneira de introduzir conceitos gerais empregados na construção de linguagens funcionais.

Em sua forma mais simples, o modelo é composto por variáveis, abstrações e aplicações de funções. A sintaxe da linguagem de programação é representada formalmente através da gramática da Figura 2.1. Na Figura, t é uma meta variável que apresenta a sintaxe dos termos na forma normal de Backus (BNF), podendo ser substituída por um dos 3 termos definidos após o símbolo ::= (PIERCE, 2002). A construção $\lambda x.t$ é utilizada para definir uma função anônima, com um único argumento de entrada representado pela variável x e corpo (valor de retorno) definido pelo termo t . O termo $(\lambda x.x) t$ representa a aplicação de um termo qualquer como parâmetro da função definida pela abstração λ .

A compreensão de programas mais complexos nessa linguagem é facilitada com a eliminação de possíveis ambiguidades sintáticas que confundam a estrutura dos termos. Por exemplo, o significado do termo $\lambda x.\lambda y.x$ e y é condicionado às convenções sintáticas a

Figura 2.1 – Gramática do Cálculo Lambda.

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

Fonte: (PIERCE, 2002)

seguir, evidenciadas pelo uso de parênteses (THOMPSON, 1991):

- A aplicação tem precedência sobre a abstração: $\lambda x.\lambda y.(x y x)$
- A aplicação é associativa a esquerda: $\lambda x.\lambda y.((x y) x)$
- Uma abstração é estendida à direita: $\lambda x.(\lambda y.((x y) x))$

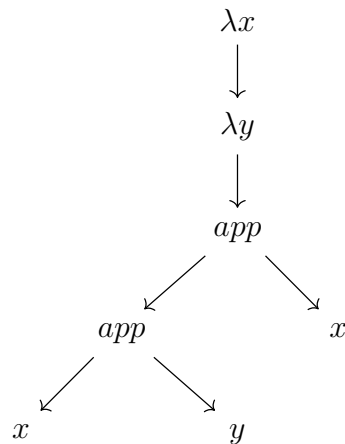
Quando se trabalha com linguagens de programação é importante enfatizar a distinção entre a chamada *sintaxe concreta* da linguagem, empregada na escrita de programas por parte de um usuário final, e a *sintaxe abstrata*, empregada na construção de ferramentas de especificação e análise semântica (PIERCE, 2002). Para o primeiro caso, elementos sintáticos como o uso de parênteses podem ser incluídos para eliminar ambiguidades de significado nos programas. Pelas convenções sintáticas, o termo $\lambda x.\lambda y.x y x$ representa uma abstração lambda com corpo definido pela aplicação $(x y z)$. Através da inclusão de parênteses, o termo $(\lambda x.\lambda y.x) y x$ representa a aplicação das variáveis x e y como argumento da abstração lambda cujo corpo é definido simplesmente por $\lambda y.x$.

No caso da sintaxe abstrata, uma representação simplificada é utilizada para tornar o significado mais aparente. Estruturas comumente utilizadas para tal propósito são as árvores de sintaxe abstrata, referidas pela sigla AST advinda da nomenclatura em inglês. A árvore de sintaxe abstrata do termo $\lambda x.\lambda y.x y x$ é apresentada na Figura 2.2. Em uma AST, termos são apresentados como nós, enquanto setas designam as relações entre os termos. A leitura é sempre realizada de cima para baixo, da esquerda para a direita, explicitando a precedência das aplicações: $(x y) x$.

Uma implementação da semântica da linguagem trabalha diretamente com a segunda representação, utilizando mecanismos de análise sintática para realizar a tradução entre as representações. Esses mecanismos são equipados com as convenções sintáticas, aplicando regras de precedência e associatividade na construção de uma AST (PIERCE, 2002).

Na forma $(\lambda x.\lambda y.x) y x$, o termo representa uma abstração aplicada a dois parâmetros: $(y x)$. Dado que uma abstração é sempre vinculada a uma única variável representando a entrada, é necessária uma equivalência. A especificação de funções com múltiplos argumentos é possibilitada através do mecanismo de Currying, detalhado em Curry et al. (1958). O processo de *Currying* dita que a aplicação de múltiplos argumentos a uma função é equivalente a aplicação sequencial das entradas, cada qual

Figura 2.2 – Uma árvore de sintaxe abstrata



Fonte: Adaptado de Pierce (2002).

sendo feita ao resultado da aplicação anterior. Desse modo, a aplicação descrita pelo termo acima é equivalente a aplicação denotada pelo termo $((\lambda x.\lambda y.x) y) x$. A aplicação desse mecanismo é um fator característico de linguagens funcionais (HUDAK, 1989).

O componente de uma linguagem de programação que define como uma computação será realizada é denominado semântica. No cálculo Lambda, a base semântica é a operação de substituição (PIERCE, 2002). Ao aplicar um termo t a uma abstração λ , substituímos as variáveis que denotam os parâmetros em por t no corpo da função, removendo a abstração no processo. Um exemplo simples é a aplicação $(\lambda x.x)t$, onde a aplicação da substituição resulta simplesmente em t . A abstração $\lambda x.x$ representa a função identidade.

Formalmente, a substituição é escrita $t_1[t_2/x]$, significando o termo onde as ocorrências livres da variável x são substituídas por t_1 no termo t_2 (PIERCE, 2002). Se uma variável x está presente no corpo t de uma abstração $\lambda x.t$, diz-se que a ocorrência dessa variável é ligada a abstração em t . Uma ocorrência livre de uma variável é um ponto no termo onde a variável não está ligada a nenhuma abstração. Essa relação de ocorrência é por vezes retratada como o escopo da variável, dizendo que uma variável pertence ao escopo da abstração a qual ela é ligada. Termos do tipo $\lambda x.y$ não possuem variáveis livres e representam funções com resultado constante. Aplicar qualquer termo t' a essa abstração resulta na própria variável y .

O processo de substituição em aplicações descrito acima é uma descrição textual da regra de β -redução (adaptada de Thompson (1991)):

Definição 1. β -redução: para todo x, t, t_1 , a aplicação pode ser reescrita através da substituição: $(\lambda x.t)t_1 \rightarrow t_1[t/x]$

A reescrita do termo através da substituição é chamada de redução e é equivalente a um passo de avaliação na linguagem. Um termo no qual a aplicação da β -redução é possível é dito redutível e chamado de *redex* (PIERCE, 2002). Quando um termo existe

em uma forma irreduzível, diz-se que o termo está em sua forma normal, ou que o termo é um valor.

Considerando a avaliação do termo $(\lambda a.\lambda b.a)c((\lambda d.e)d)$, a regra da β -redução indica a existência de dois *redexes*: $((\lambda a.\lambda b.a)c)$ e $((\lambda d.e)d)$. O processo de escolha do termo a ser reduzido no passo atual de redução é chamado de estratégia de avaliação ¹. A estratégia *call-by-name* determina que a redução sempre seja feita com o redex mais externo e mais a esquerda (PIERCE, 2002). Além disso, não é permitida a redução de termos no corpo de abstrações. Seguindo essa estratégia, os passos de redução do termo são destacados na Tabela 2.1, onde os termos redutíveis em cada passo aparecem sublinhados.

Tabela 2.1 – Avaliação *Call-by-name*.

Termo	Substituição
<u>$(\lambda a.\lambda b.a)c((\lambda d.e)d)$</u>	$c[a/\lambda b.a]$
<u>$\lambda b.c ((\lambda d.e)d)$</u>	$b[((\lambda d.e)d)/c]$
<u>c</u>	

Utilizando essa estratégia, os argumentos de uma função são passados a mesma em sua forma original, sem avaliação prévia. A avaliação dos argumentos só é realizada quando necessário ou, como no caso do exemplo, não é realizada quando os argumentos não são relevantes para o corpo da função. Linguagens como Algol-60 e Haskell utilizam variações otimizadas dessa estratégia de avaliação (PIERCE, 2002).

Uma segunda estratégia é a chamada *call-by-value*, ilustrada na Tabela 2.2. A redução continua sendo feita no termo mas externo e mais a esquerda, com a restrição de que um redex só é avaliado quando o seu lado direito é um valor. Utilizando a estratégia *call-by-value*, os argumentos sempre serão avaliados antes de serem aplicados a função, mesmo que sejam descartados pela avaliação final.

Tabela 2.2 – Avaliação *Call-by-value*.

Termo	Substituição
<u>$(\lambda a.\lambda b.a)c((\lambda d.e)d)$</u>	$c[a/\lambda b.a]$
<u>$\lambda b.c ((\lambda d.e)d)$</u>	$d[d/e]$
<u>$(\lambda b.c) e$</u>	$e[b/c]$
<u>c</u>	c

As regras de avaliação estabelecidas por uma estratégia podem ser representadas através de um formalismo denominado semântica operacional, representada através

¹O termo *estratégia de avaliação* costuma ser empregado para descrever sistemas nos quais a noção de valor é utilizada. Para casos mais gerais, o termo *estratégia de redução* pode ser mais adequado (PIERCE, 2002).

de um conjunto de regras de indução estrutural. Através desse mecanismo, um sistema de avaliação pode ser especificado formalmente e submetido a provas sobre seu funcionamento. Considerando a estratégia *call-by-value*, a semântica operacional do cálculo Lambda pode ser descrita pelas regras da Figura 2.3, onde t é uma meta-variável para termos do cálculo lambda, e v representa os valores da linguagem. As regras de indução são compostas por premissas, apresentadas acima da linha horizontal, e conclusões, abaixo da linha. Para verificar a veracidade da conclusão, é necessário verificar a veracidade de todas as premissas. Regras nas quais a linha horizontal é omitida são denominadas axiomas e representam verdades absolutas. A notação $t_1 \rightarrow t'$ representa a relação de avaliação dos termos t_1 e t' e é lida da forma: t avalia para t' em um passo de avaliação (PIERCE, 2002).

Figura 2.3 – Semântica operacional do cálculo lambda.

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \text{App1} \quad \frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \text{App2}$$

$$\frac{}{(\lambda x.t_{12}) v_2 \rightarrow v_2[t_{12}/x]} \text{Abs}$$

Fonte: Adaptado de Pierce (2002).

Utilizando esse conjunto de regras, a avaliação do termo $(\lambda a.\lambda b.a)c((\lambda d.e)d)$ é dada pela aplicação sequencial dessas, ilustrada na Figura 2.4.

Figura 2.4 – Avaliação através da semântica operacional.

$$\frac{\frac{}{(\lambda a.\lambda b.a) c \rightarrow c[a/\lambda b.a]} \text{Abs}}{((\lambda a.\lambda b.a) c) ((\lambda d.e)d) \rightarrow t'_1 ((\lambda d.e)d)} \text{App1}^{(1)} \quad \frac{\frac{}{((\lambda d.e)d) \rightarrow d[d/e]} \text{Abs}}{(\lambda b.c)((\lambda d.e)d) \rightarrow (\lambda b.c) t'_2} \text{App2}^{(2)}}{(\lambda b.c) e \rightarrow e[c/b]} \text{Abs}^{(3)}$$

Fonte: Adaptado de Pierce (2002).

Dentre as três regras apresentadas, apenas a regra da abstração descreve um passo efetivo de computação (β -redução). A função das regras referentes a aplicação é restringir a escolha do termo a ser avaliado no próximo passo. Enquanto o número de regras do cálculo Lambda é limitado, em linguagens mais complexas torna-se interessante a utilização de uma representação mais compacta para esse tipo de regra restritiva (NIELSON; NIELSON, 2007).

Uma técnica utilizada para tal propósito é a construção de contextos de avaliação. Um contexto de avaliação é um construto $C[\cdot]$ contendo uma abertura na qual um termo da linguagem é inserido. A abertura ($[\cdot]$) denotada no contexto aponta para o termo que deve ser avaliado atualmente. No caso do cálculo lambda, aponta para um *redex* (SABRY; FELLEISEN, 1993). Utilizando a sintaxe de contextos de avaliação, a estratégia *call-by-*

value poderia denotar as regras de aplicação da forma

$$C[\cdot] := [\cdot] \mid C[\cdot] t \mid v C[\cdot],$$

onde v representa um valor e t um termo qualquer da linguagem. O comportamento dos termos não contemplados no contexto de avaliação continua a ser especificado via regras de inferência. Retomando o exemplo anterior, uma avaliação utilizando essa metodologia é representada pelos passos da tabela 2.3:

Tabela 2.3 – Contextos de Avaliação

Termo	Regra Aplicada
$[(\lambda a. \lambda b. a) c ((\lambda d. e) d)]$	$C[\cdot] t$
$[(\lambda a. \lambda b. a) c] ((\lambda d. e) d)$	$c[\lambda b. a/a]$
$(\lambda b. c) [((\lambda d. e) d)]$	$v C[\cdot]$
$[(\lambda b. c) e]$	$e[c/b]$
c	

Das características marcantes de linguagens funcionais, o cálculo Lambda expõe a base semântica do paradigma através da operação de substituição, algumas estratégias de avaliação que fundamentam linguagens modernas e a codificação de programas através da especificação e aplicação de funções. Outro ponto característico dessas linguagens é o emprego de recursão para codificar operações repetidas.

Para explorar a recursão no cálculo Lambda, considera-se a representação dos termos booleanos $tt = \lambda t. \lambda f. t$ e $fls = \lambda t. \lambda f. t$, além da função

$$test = \lambda l. \lambda m. \lambda n. l m n,$$

adaptados de Pierce (2002). Note-se que funções no cálculo lambda são anônimas, logo tt, fls e $test$ são simplesmente nomes adotados para possibilitar a referência dessas expressões no decorrer do texto, sem nenhum significado semântico no cálculo lambda. Sumarizando as especificações apresentadas, a função $test$ recebe 3 argumentos l, m, n , reduzindo para m quando $l = tt$ ou para n quando $l = fls$. Utilizando como base as definições apresentadas, é possível especificar uma função que, recebendo o argumento fls repete o teste recursivamente no próximo argumento.

Como o cálculo Lambda utiliza apenas funções anônimas, a intuição

$$testR = \lambda l. \lambda m. \lambda n. l m (testR m n l)$$

não é permitida. Para que a descrição recursiva seja possível, é necessário transformar

test em uma função de alta ordem através da adição de uma nova abstração:

$$recF = \lambda re.\lambda l.\lambda m.\lambda n.l m (re m n l).$$

Com essa definição em mãos, resta fornecer um argumento apropriado que substitua *re* durante a avaliação (COUSINEAU; MAUNY, 1998). Fornecer uma cópia de *recF* não é suficiente, já que a avaliação da cópia exigirá o fornecimento de outra cópia, que exigirá outra cópia e assim por diante. É necessária a utilização de um termo capaz de se auto-replicar quando a aplicação da função recursiva for necessária (PIERCE, 2002). Esse termo é o chamado operador de ponto fixo ²:

$$Z = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)).$$

Utilizando o operador *Z*, é possível construir a função recursiva através da aplicação da função *recF* a *Z*, seguida da aplicação dos argumentos computáveis à função resultante via *Currying*. Na tabela 2.4, vê-se a avaliação da aplicação $Z\ recF\ tt\ tt\ fls$.

Tabela 2.4 – Avaliação do operador de ponto fixo.

Termo	Regra de Av.
1 $(\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)))\ recF\ tt\ tt\ fls$	$recF[.../f]$
2 $(\lambda x. recF (\lambda y. x x y)) (\lambda x. recF (\lambda y. x x y))\ tt\ tt\ fls$	$t_2 := (\lambda x. recF (\lambda y. x x y))$
3 $(\lambda x. recF (\lambda y. x x y))\ t_2\ tt\ tt\ fls$	$t_2[recF (\lambda y. x x y)/x]$
4 $(recF (\lambda y. t_2 t_2 y))\ tt\ tt\ fls$	expande <i>recF</i>
5 $(\lambda re.\lambda l.\lambda m.\lambda n.l m (re m n l)) (\lambda y. t_2 t_2 y)\ tt\ tt\ fls$	$(\lambda y. t_2 t_2 y)[.../re]$
6 $(\lambda l.\lambda m.\lambda n.l m ((\lambda y. t_2 t_2 y) m n l))\ tt\ tt\ fls$	$tt[\lambda m.../l]$
7 $\lambda m.\lambda n. tt m ((\lambda y. t_2 t_2 y) m n tt)\ tt\ fls$	$tt[\lambda n.../m]$
8 $\lambda n. tt tt ((\lambda y. t_2 t_2 y) tt n tt)\ fls$	$fls[tt.../n]$
9 $tt tt ((\lambda y. t_2 t_2 y) tt fls tt)$	$tt := (\lambda t.\lambda f.t)$
10 $(\lambda t.\lambda f.t) (\lambda t.\lambda f.t) ((\lambda y. t_2 t_2 y) tt fls tt)$	$(\lambda t.\lambda f.t)[\lambda f.t/t]$
11 $(\lambda f.(\lambda t.\lambda f.t)) ((\lambda y. t_2 t_2 y) tt fls tt)$	$((\lambda y...)\ tt...)[(\lambda t.\lambda f.t)/f]$
12 $(\lambda t.\lambda f.t)$	tt

O comportamento recursivo do operador de ponto fixo pode ser evidenciado alterando os argumentos da função anterior para $(fls\ tt\ tt)$. Com esses argumentos, a avaliação procede como na tabela 2.5, que omite os passos iniciais de avaliação (1 ao 8) já ilustrados na tabela 2.4.

Uma comparação dos termos revela que o passo 14 na tabela 2.5 é idêntico ao passo 2 da tabela 2.4, sendo os passos de avaliação subsequentes omitidos. Em conjunto, as quadros evidenciam a semântica recursiva de uma função definida através do operador de ponto fixo.

²O operador *Z* apresentado é definido para o cálculo lambda não tipado sob a estratégia de avaliação *call-by-value*. Outras estratégias de avaliação possuem seu próprio operador de ponto fixo, por exemplo o *combinador Y* para o cálculo lambda reduzido via *call-by-name*.

Tabela 2.5 – Avaliação do operador de ponto fixo 2.

	Termo	Regra de Av.
1-8	Tabela 2.4	
9	$f\ l\ s\ t\ t\ ((\lambda y. t_2\ t_2\ y)\ f\ l\ s\ t\ t\ f\ l\ s))$	$f\ l\ s := \lambda t. \lambda f. f$
10	$(\lambda t. \lambda f. f)\ (\lambda t. \lambda f. t)\ ((\lambda y. t_2\ t_2\ y)\ f\ l\ s\ t\ t\ f\ l\ s))$	$(\lambda t. \lambda f. t)[\lambda f. f/t]$
11	$(\lambda f. f)\ ((\lambda y. t_2\ t_2\ y)\ f\ l\ s\ t\ t\ f\ l\ s))$	$((\lambda y \dots) t\ t \dots)[f/f]$
12	$((\lambda y. t_2\ t_2\ y)\ t\ t\ t\ f\ l\ s))$	$t\ t[\lambda y \dots / y]$
13	$(t_2\ t_2)\ t\ t\ t\ f\ l\ s$	expande t_2
14	$(\lambda x. recF\ (\lambda y. x\ x\ y))\ (\lambda x. recF\ (\lambda y. x\ x\ y))\ t\ t\ t\ f\ l\ s$	

Matematicamente, o ponto fixo de uma função é um valor x tal que $f(x) = x$, ou seja um valor que, suprido como argumento de uma função, gera a si mesmo (GUNTER, 1992). Substituindo f por uma função de alta ordem $recF$, então o ponto fixo de $recF$ é dado pela equação:

$$\begin{aligned}
 fix &= recF\ fix \\
 &\rightarrow recF\ (recF\ fix) \\
 &\rightarrow recF\ (recF\ (recF\ fix)) \\
 &\dots
 \end{aligned}$$

A versão simplificada do cálculo lambda apresentada até aqui não oferece nenhuma garantia de terminação. De fato, aplicar a função recursiva anterior aos argumentos $f\ l\ s\ f\ l\ s\ f\ l\ s$ gera uma avaliação que nunca chegaria ao fim. Indo além, não existem restrições impostas aos valores que podem ser utilizados como parâmetros de qualquer função, dificultando a verificação e prova do comportamento das funções, mesmo as não recursivas. Oferecer garantias de terminação e possibilitar a verificação prévia do comportamento de um programa exige a adoção de mecanismos mais robustos. As especificações sintática e semântica de linguagens funcionais apresentam refinamentos sobre os mecanismos básicos do cálculo lambda de forma a alcançar esses propósitos, como a classificação de termos através de sistemas de tipos (PIERCE, 2002) e a inclusão de equações e *pattern-matching* para descrever e limitar os valores de computações (HUDAK, 1989).

No que se refere a sintaxe, programas expressados em linguagens funcionais são caracterizados por uma notação concisa, similar a notação matemática tradicional (HUDAK, 1989). Como um exemplo, considera-se a utilização de variáveis ilustrada pelo pseudocódigo da Figura 2.5. Em linguagens funcionais, variáveis cumprem um papel similar a sua versão matemática: são símbolos aplicados para representar algum valor imutável, obtido através da avaliação do termo apresentado após o sinal de igualdade (COUSINEAU; MAUNY, 1998). Qualquer que seja a expressão, o processo de avaliação pode sempre substituir a variável pela equação definida após o sinal de

igualdade, desde que respeitando o escopo da definição. No caso da Figura 2.5, é possível determinar imediatamente o valor do argumento passado a função `print`, mesmo tendo o restante do código omitido. Como o valor das variáveis é imutável, é possível substituir a equação $(y/2) + 2$ por $((x*3) / 2) + 2$ e substituir a nova equação pelo valor definido para x , computando enfim a operação aritmética que determina o valor de z .

Outra característica importante das linguagens funcionais é a noção de pureza de execução. Em uma linguagem funcional, uma chamada de função gera somente um efeito: computar seu valor (HUGHES, 1989). Em um paradigma imperativo, onde o programador interage com um estado implícito através de construtos como atribuição a variáveis (HUDAK, 1989), é necessária a manutenção de uma sequência precisa de execução, de forma que o programador consiga acompanhar as mudanças de estado. Em linguagens funcionais, a pureza de funções possibilita que a avaliação de expressões ocorra a qualquer momento (HUGHES, 1989). A especificação de programas no paradigma funcional tem uma natureza mais descritiva, resumida por Hudak (1989) na frase "Programas funcionais descrevem *o que* está sendo computado, ao invés de descrever *como* computá-los.

Essas características facilitam a rápida prototipação de sistemas, utilizando código de alto nível cujas propriedades podem ser analisadas mais facilmente. É importante deixar claro que, na prática, várias linguagens funcionais possibilitam a inclusão de efeitos colaterais através de construções de propósito específico, embora sua utilização não seja de todo encorajada (HUDAK, 1989).

Funções em linguagens funcionais são mais abrangentes do que as abstrações presentes no cálculo lambda. Em primeiro lugar, funções podem ser nomeadas através de um identificador, que é utilizado para representar a definição completa da função durante o decorrer do programa. Um exemplo simples é a definição de uma função geradora para a sequência de Fibonacci para números inteiros positivos, cuja definição matemática é apresentada na equação 2.1. Nesse caso, a função resulta no valor 1 quando o seu argumento de entrada n é menor que 2 e no valor resultante da equação $fib(n - 1) + fib(n - 2)$ para os demais valores.

$$fib(n) = \begin{cases} 1 & ; n < 2 \\ fib(n - 1) + fib(n - 2) & ; n \geq 2 \end{cases} \quad (2.1)$$

Figura 2.5 – Utilização de variáveis em linguagens de programação.

```

1 x = 2
2 y = x * 3
3 ..
4 z = (y / 2) + 2
5 print z

```

Linguagens funcionais providenciam a capacidade de especificar diferentes comportamentos para uma mesma função através de *pattern-matching*³ (HUDAK, 1989). A Figura 2.6 apresenta a definição da função de Fibonacci na sintaxe da linguagem Haskell. Nessa construção, cada linha define um padrão, a esquerda do sinal de igualdade, e uma equação representando o valor resultante a direita.

Comumente, os padrões são denotados de forma explícita após o nome da função, precedendo uma equação. De forma geral, os padrões podem consistir de valores, variáveis, construtores e caracteres coringa (THOMPSON, 1999). Durante o processo de avaliação, o argumento aplicado será comparado aos padrões definidos na função, possibilitando a escolha da equação adequada para o valor fornecido. Para padrões definidos por valores específicos, o procedimento de *pattern-matching* equivale a uma comparação de igualdade. Se um padrão é definido por uma variável, o procedimento atribui a essa variável o valor do argumento passado a função, substituindo-a por tal valor na equação a direita da igualdade.

Diferente da definição matemática, a aplicação `fib 1` na definição funcional pode incorrer em ambiguidades. A primeira vista, o termo corresponde adequadamente a segunda definição de padrões, porém, como a variável `n` pode ser substituída por qualquer valor no corpo da função, é possível combinar o termo também com a terceira definição de padrões.

Em casos assim, a escolha da equação adequada é uma decisão de projeto específica a cada linguagem. Linguagens como Haskell escolhem pela aplicação de um *pattern-matching* sequencial, iniciando pela análise do padrão no topo da função e concluindo na primeira correspondência (ALLEN; MORONUKI, 2017). Nesse caso, os padrões mais restritivos devem sempre ser especificados acima de padrões mais genéricos. Outra possibilidade é exigir que todos os padrões sejam disjuntos, ou seja, o conjunto de valores com possível correspondência a um padrão não deve possuir intersecção com o conjunto de valores que correspondem a qualquer padrão distinto naquela função (HUDAK, 1989).

Dada a filosofia do paradigma, funções são incluídas no conjunto de valores das linguagens (COUSINEAU; MAUNY, 1998). Assim sendo, as mesmas podem ser utilizadas como argumentos aplicadas a outras funções ou retornadas como resultado de uma avaliação. Funções que apresentam esse comportamento são denominadas funções de

³O termo *pattern-matching* pode ser traduzido em casamento de padrões. A forma em inglês é mantida durante o decorrer do texto devido a sua prevalência na área de linguagens de programação.

Figura 2.6 – Definição da função de Fibonacci através de *pattern-matching*

```

1 fib 0 = 1
2 fib 1 = 1
3 fib n = fib n-1 + fib n-2

```

alta ordem (THOMPSON, 1999). A presença dessas funções estende a modularidade dos programas, servindo como um mecanismo para agrupar fragmentos distintos do programa (HUGHES, 1989).

Um exemplo de função de alta ordem com uso comum no meio funcional é a definição recursiva de `map` da Figura 2.7. A função `map` aplica alguma operação f a todos os elementos de uma lista, sendo a operação fornecida como um argumento da função. Na Figura, o operador `:` é um construtor de listas, `head` é uma função que extrai o primeiro elemento da lista e `tail` uma função cujo retorno é a lista completa com exceção do primeiro elemento.

Ainda considerando a função `map`, especificar o seu comportamento não requer informações sobre a estrutura da lista ou sobre a operação que será aplicada. Esse tipo de especificação ilustra o conceito de abstração de dados que embora não seja uma exclusividade do paradigma funcional (CARDELLI; WEGNER, 1985), favorece a natureza descritiva dos programas. É interessante porém garantir que a operação f fornecida possui comportamento definido para os elementos da lista, de forma a evitar efeitos indesejados durante a avaliação. Um mecanismo de abstração de dados que permite esse tipo de garantia é chamado sistema de tipos.

2.2 SISTEMAS DE TIPOS

Um sistema de tipos é um método formal para garantir que o comportamento de um sistema seja o esperado pelos seus projetistas (PIERCE, 2002). No campo de linguagens de programação, estes sistemas são utilizados para comprovar propriedades estáticas da linguagem em foco, conciliando desenvolvimento e verificação em um único sistema (THOMPSON, 1991).

Em uma linguagem de programação simples, desprovida de um sistema de tipos, provas sobre o comportamento do programa precisam ser realizadas para cada programa construído. Comparativamente, uma linguagem equipada com um sistema de tipos traz a garantia de corretude para qualquer programa especificado de acordo com a formalização da sintaxe (THOMPSON, 1991).

Sistemas de tipos bem especificados garantem a inexistência de uma classe de comportamentos erráticos, normalmente denominados erros de execução, em programas

Figura 2.7 – Definição da função recursiva `map`.

```

1 map f [] = []
2 map f list = f (head list) : map f (tail list)

```

Fonte: (ALLEN; MORONUKI, 2017).

que passem pela verificação formal do sistema (CARDELLI, 1996). A verificação dessa proposta pode ser realizada através de ferramentas automatizadas inclusas no processo de compilação dos programas (PIERCE, 2002), percorrendo o código descrito e realizando uma checagem dos tipos declarados pelo programador e dos tipos atribuídos às expressões. Essas ferramentas são denominadas *typecheckers*.

No contexto da ciência da computação, tipos podem ser entendidos como uma classificação para os valores possíveis para uma expressão em um determinado programa (PIERCE, 2002). Tomando como exemplo uma variável x qualquer, uma linguagem tipada determina limites para a amplitude de valores que x pode assumir. Esse limite de valores é chamado **tipo** (CARDELLI, 1996). Se uma variável apresenta o tipo `Int`, tal variável pode, durante a execução do programa, assumir valores correspondentes aos números inteiros. Uma atribuição de um valor do tipo ponto flutuante para a variável x seria uma violação do sistema de tipos. A verificação de que o tipo de x é consistente, garante que qualquer função $f(x)$ tem um comportamento bem definido para qualquer execução do programa. Em uma linguagem desprovida de tipos, a função acima poderia ser aplicada a argumentos inválidos, resultando em comportamentos não esperados (CARDELLI, 1996).

Em linguagens que possuem um sistema de tipos, a verificação de possíveis anomalias no comportamento podem ser verificadas em tempo de compilação, garantindo que, se uma expressão não passar pelo crivo do *typechecker*, o código não será compilado e a expressão não será avaliada. Essa verificação, quando realizada durante a compilação, é denominada checagem de tipos e consiste em uma verificação estática (CARDELLI, 1996).

Alguns erros não podem ser detectados pela checagem estática. Por exemplo, detectar um acesso indevido a uma lista é um teste que precisa ser realizado em tempo de execução. O *typechecker* não exime o programador de realizar testes próprios, deixando que a execução prossiga às cegas (CARDELLI, 1996).

A utilização de sistemas de tipos traz benefícios tanto aos desenvolvedores da linguagem, quanto aos programadores que venham a utilizá-la. Questões de segurança, detecção de erros e eficiência são abrangidas pelos benefícios destas ferramentas. Para o programador, um sistema de tipos provê uma forma de detecção precoce de erros, de forma que um código com falhas não passe pelo processo de compilação. Um componente capaz de checar a tipagem da linguagem costuma expor uma gama de problemas no código, os quais, sem esta ferramenta, seriam encontrados apenas em tempo de execução. Programas que manipulam uma grande variedade de estruturas de dados tendem a obter maior benefício destas ferramentas (PIERCE, 2002).

Similarmente, a checagem de tipos é utilizada para prover a segurança da linguagem, de forma que operações proibidas não sejam executadas, como a aplicação de operações aritméticas em valores não numéricos. Através desta checagem, uma

linguagem consegue proteger suas estruturas, garantindo que o acesso as estruturas de dados seja feito apenas por meio das abstrações criadas para este propósito. Além disso, sistemas de tipos podem ser utilizados para garantir propriedades estáticas de uma linguagem, tais como homogenizar a estrutura da definição de funções ou verificar que uma variável não recebe dois valores distintos em um mesmo escopo (PIERCE, 2002).

A checagem estática elimina a necessidade de diversos testes em tempo de execução utilizados para garantir a segurança da linguagem (PIERCE, 2002), proporcionando aos desenvolvedores da mesma ganhos de eficiência. No geral, a compilação de um programa com informações de tipos leva a aplicação de operações em tempo de execução sem a necessidade de testes computacionalmente caros (CARDELLI, 1996). Um exemplo é a utilização de diferentes representações para valores inteiros e valores reais, a qual possibilita a realização de uma operação de adição sem a necessidade de um teste prévio, pois o sistema de tipos já garante a utilização de valores coerentes com a operação.

No que consta a formalização de linguagens, um sistema de tipos é comumente expresso através de um conjunto de julgamentos representados por regras de inferência que associam os termos da linguagem aos tipos adequados (PIERCE, 2002). Um conjunto restrito de regras de inferência pode ser visualizado na tabela 2.8.

Figura 2.8 – Exemplos de regras de tipos.

$$\frac{}{\Gamma \vdash true : Bool} \quad \frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

Fonte: (PIERCE, 2002)

No caso do exemplo fornecido, o axioma afirma que o termo *true* é uma construção pertencente ao conjunto de tipos *Bool*. A regra de tipos da direita especifica formalmente como o tipo da expressão condicional pode ser inferido através dos seus componentes: o termo t_1 deve apresentar o tipo *Bool*, enquanto t_2 e t_3 devem apresentar o mesmo tipo *T*. Verificadas as premissas, pode-se inferir que o tipo da expressão condicional é o tipo *T*, resultante da avaliação de uma das subexpressões.

Para qualquer termo pertencente a linguagem, a notação $t : T$ é uma afirmação, indicando que o termo t é um membro do conjunto formado pelo tipo *T* (THOMPSON, 1991). O símbolo Γ representa um ambiente de tipagem, também chamado de contexto de tipos, formado por uma lista de variáveis e seus respectivos tipos (CARDELLI, 1996). O contexto de tipos indica um conjunto de suposições sob o qual a inferência está sendo realizada. Quando acompanhado de uma vírgula em uma premissa, o ambiente de tipagem está sendo estendido através da inclusão da afirmação que segue, fazendo com que a checagem prossiga sob o novo contexto. Quando o uso da vírgula é feito na conclusão da regra, essa sintaxe indica que a conclusão será verdadeira somente quando a variável após a vírgula estiver presente no contexto com o mesmo tipo. a Figura 2.9

apresenta as duas variações.

Figura 2.9 – Extensão e teste do conteúdo do contexto de tipos.

$$\frac{\Gamma, x : T \vdash y : T_1}{\Gamma \vdash \lambda x. y : T_1} \quad \frac{}{\Gamma, x : T \vdash x : T}$$

Fonte: Adaptado de Pierce (2002).

As regras de inferência determinam o comportamento formal da checagem de tipos na linguagem. Para determinar o tipo de uma construção sintática, aplicam-se as regras de inferência de forma a construir uma prova indutiva de que a construção é bem formada de acordo com o sistema de tipos: aplica-se as regras cabíveis para as expressões acima da linha de forma a obter uma derivação do julgamento abaixo da linha (THOMPSON, 1991).

Considerando a afirmação: `if true then x else y : T`, a prova indutiva pode ser construída através das regras apresentadas nos exemplos, obtendo a seguinte árvore de derivação:

$$\frac{\frac{}{\Gamma \vdash true : Bool} \quad \frac{}{\Gamma, x : T \vdash x : T} \quad \frac{}{\Gamma, x : T \vdash x : T}}{\Gamma \vdash \text{if } true \text{ then } x \text{ else } y : T}$$

Uma prova é concluída quando todas as derivações alcançam axiomas. Assumindo que as variáveis x e y estejam contidas no contexto Γ com tipo T , a árvore de derivação da expressão prova que o condicional expressado é do tipo T . Se, durante o processo de derivação dos julgamentos de tipos, não for possível prosseguir com a derivação de um julgamento, a expressão original será considerada mal formada.

2.3 LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL REVERSÍVEL

Os modelos clássicos de computação abstrata consideram a computação como um processo estritamente irreversível (YOKOYAMA; AXELSEN; GLÜCK, 2008). Uma operação irreversível, no contexto computacional, é uma operação em que os valores de entrada não podem ser recuperados unicamente a partir do valor de saída (LANDAUER, 1961). Um exemplo destas computações é a função lógica `nand`, cuja irreversibilidade pode ser evidenciada pela tabela verdade (tabela 2.6).

Pela observação da tabela, é visível que não se pode precisar os bits de entrada da função observando apenas o valor de saída 1. Computações, de um modo geral, envolvem transformações do tipo muitos para um, resultando no descarte de informações referentes ao histórico das computações realizadas (BENNETT, 1973), denominado apagamento de bits (*bit erasure*). Segundo os trabalhos de Landauer (1961), o apagamento de bits resulta obrigatoriamente em um custo energético, realizado pela dissipação de calor.

Tabela 2.6 – Tabela verdade de nand com 2 bits.

Entrada		Saída
a	b	z
0	0	1
0	1	1
1	0	1
1	1	0

Esse conceito é formalizado através da noção de entropia da informação definida por Shannon (2001), definida na equação 2.2. Considerando uma variável x qualquer com valores possíveis representados por x^1, x^2, \dots, x^n , p_i representará a probabilidade da variável x assumir o valor x^i , sendo H a entropia da variável x considerando a unidade de medida arbitrária k .

$$H = -k \sum_1^n p_i \log p_i. \quad (2.2)$$

Tomando como exemplo a função nand apresentada anteriormente, cujos possíveis valores de entrada são os pares: (0,0),(0,1),(1,0) e (1,1), em um sistema no qual todos tem a mesma probabilidade de ocorrência 1/4, a entropia da entrada da função nand é dada por⁴:

$$k \left(\frac{1}{4} \log 4 + \frac{1}{4} \log 4 + \frac{1}{4} \log 4 + \frac{1}{4} \log 4 \right) \approx 0,6020k.$$

Na mesma função, com a mesma consideração sobre as probabilidades dos valores de entrada, as probabilidades dos valores de saída 1 e 0 serão respectivamente 3/4 e 1/4. Assim sendo, a entropia da saída da função nand é:

$$k \left(\frac{3}{4} \log \frac{4}{3} + \frac{1}{4} \log 4 \right) \approx 0,2442k.$$

A diferença de entropia da entrada e da saída contabiliza a informação perdida pela realização da computação descrita por nand e apresenta um valor mínimo de dissipação de calor para esta operação irreversível (LANDAUER, 1961). Uma função na qual a entropia dos valores de saída é igual a entropia dos valores de entrada é uma função que preserva informação (JAMES; SABRY, 2012).

Essa relação de dissipação de calor existe por conta do contraste entre o modelo abstrato de computação e os sistemas físicos em que esse modelo é implementado. Os modelos microscópicos da física são presumidamente reversíveis, o que faz com que, na implementação de circuitos computacionais, seja necessária a transposição

⁴A equação mostrada é a forma reduzida do somatório, considerando a propriedade do logaritmo: $-\log 1/4 = \log 4/1$.

entre a irreversibilidade da computação e a reversibilidade da física (TOFFOLI, 1980). Comumente, esta transição é realizada em baixo nível através de portas lógicas que devem obedecer às leis físicas de reversibilidade enquanto executam computações irreversíveis. Estes mecanismos implementam uma irreversibilidade macroscópica através da transformação de trabalho em calor (TOFFOLI, 1980).

A computação reversível é a proposição de um modelo computacional capaz de transpor o espaço entre o comportamento irreversível desejado no modelo abstrato e a reversibilidade do modelo físico de uma forma explícita. De acordo com Toffoli (1980), toda função lógica finita possui uma versão reversível equivalente que pode ser obtida mediante a alteração de seus argumentos e valores de saída. Funções finitas são funções nas quais o conjunto de valores possíveis para a entrada e o conjunto de valores possíveis para a saída são ambos conjuntos finitos (TOFFOLI, 1980) e podem ser descritas da forma $\{f: X^m \rightarrow Y^n \mid X, Y \text{ são conjuntos finitos}\}$. Uma função ϕ^1 descrita da forma $\phi^1: \text{bool}^2 \rightarrow \text{bool}^1$, é uma função lógica que aceita como entrada 2 valores do tipo `bool` e produz como saída 1 valor do tipo `bool`, onde o tipo `bool` é um conjunto fechado formado pelos valores $\{\text{true}, \text{false}\}$, logo ϕ^1 é uma função finita.

A transformação de uma função ϕ qualquer em uma função reversível é definida pelo teorema fundamental da computação reversível (TOFFOLI, 1980):

Teorema 1. Para toda função finita $\phi: \text{bool}^m \rightarrow \text{bool}^n$ existe uma função finita reversível $\phi^R: \text{bool}^{R+m} \rightarrow \text{bool}^{R+m}$, com $R \leq n$, de forma que $\phi(x_1, \dots, x_m) = (y_1, \dots, y_n)$ se e somente se $\phi^R(x_1, \dots, x_m, \text{false}, \dots, \text{false}) = (\dots, y_1, \dots, y_n)$.

De forma simplificada, este teorema dita que a tradução de uma função lógica irreversível para uma função lógica reversível é possível através da adição de argumentos booleanos (*fonte*) e de valores de saída (*dissipador*), resultando em uma função com o mesmo número de valores recebidos como entrada e retornados como saída. Quando os valores da fonte são fixados no valor *false*, obtém-se o comportamento da função original ao ignorar os valores do dissipador (TOFFOLI, 1980).

Tomando como exemplo a função não reversível $\text{nand}: \text{bool}^2 \rightarrow \text{bool}$, é possível definir uma função equivalente $\text{nand}^R: \text{bool}^3 \rightarrow \text{bool}^3$, cujo comportamento é descrito pela tabela 2.7, onde os argumentos adicionais são destacados entre as linhas tracejadas, com o bit **c** atuando como argumento fonte e os bits **x** e **y** como valores de dissipação. O bit **z** contém o valor resultante da operação.

Todas as sequências de bits da saída são únicas em relação a entrada, logo é possível precisar a combinação de bits original observando a combinação resultante, evidenciando a reversibilidade desta função. Para a entropia, desde que cada tripla de entrada $(a, b, c)^i$ ocorra com a mesma probabilidade $p_i^e = 1/8$, é verificável que cada tripla $(x, y, z)^i$ de saída tem a mesma probabilidade de ocorrência $p_i^s = 1/8$. Logo a entropia da entrada $(-\sum_1^8 p_i^e \log p_i^e)$ será igual a entropia da saída $(-\sum_1^8 p_i^s \log p_i^s)$ e nand^R é uma

Tabela 2.7 – Tabela verdade de nand^R .

Entrada			Saída		
a	b	c	x	y	z
0	0	0	0	0	1
0	1	0	0	0	1
1	0	0	1	0	1
1	1	0	1	1	0
0	0	1	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	1	1	1	1

função que preserva informação.

Comparando a parte superior da tabela 2.7, onde o bit **c** tem o valor 0, com a tabela 2.6, pode-se verificar a equivalência de comportamentos das funções nand e nand^R . Já a segunda parte da tabela, onde o bit de controle possui o valor 1 apresenta o comportamento esperado de uma função lógica and . O comportamento da função nand^R é implementado pela porta lógica de Toffoli, CCNOT (controlled-controlled-not), componente fundamental para a construção de circuitos reversíveis, que podem apresentar dissipação energética interna próxima de zero (TOFFOLI, 1980).

Da mesma forma que os circuitos clássicos não consideram a reversibilidade, as linguagens de programação clássicas também não consideram essa propriedade, fazendo com que a construção de programas reversíveis capazes de tirar proveito dos circuitos seja de responsabilidade do programador. Uma forma de se evitar essa complexidade adicional de programação é abstrair os detalhes dos circuitos através da criação de linguagens de programação reversíveis. Estas linguagens implementam a reversibilidade de forma explícita através de sua semântica e de sua sintaxe, garantindo que todo programa expressado de acordo com tais especificações será reversível (YOKOYAMA; AXELSEN; GLÜCK, 2008).

Um exemplo de linguagem funcional reversível é a linguagem II (JAMES; SABRY, 2012). A linguagem II é construída com base na noção de isomorfismo de tipos, uma propriedade que evidencia relações de reversibilidade entre os valores da linguagem. Por definição, dois tipos são isomórficos se é possível definir um mapeamento bijetivo entre todos os valores restritos aos seus domínios (JAMES; SABRY, 2012). De uma forma mais simples, para qualquer função $f : a \rightarrow b$, todo valor x do tipo a pode ser relacionado com um único valor y do tipo b de forma que $f(x) = y$ e $f^{-1}(y) = x$.

Partindo dessa noção, a linguagem codifica tipos de soma ($a + b$) e pares ($a \times b$), acompanhados por um tipo primitivo (1) e define pares de operadores através das relações isomórficas. a Figura 2.10 apresenta alguns exemplos desses operadores. Programas na

Figura 2.10 – Exemplos de operadores isomórficos.

$$\begin{array}{lll}
 \text{swap}^+ : & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \text{swap}^+ \\
 \text{assocl}^+ : & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : \text{assocr}^+ \\
 \text{unite} : & 1 \times b \leftrightarrow b & : \text{uniti} \\
 & \dots &
 \end{array}$$

Fonte: (JAMES; SABRY, 2012).

linguagem II descrevem circuitos combinatórios construídos a partir da composição dos operadores básicos. A sintaxe dessas construções aproxima-se de linguagens de baixo nível, divergindo dos padrões comuns a linguagens de uso geral.

Mais próxima da sintaxe comum ao paradigma funcional é a linguagem reversível proposta por Yokoyama et.al. (YOKOYAMA; AXELSEN; GLÜCK, 2011). Nesta linguagem a reversibilidade é embutida explicitamente na semântica de um termo, cuja regra de avaliação permite a execução reversa de funções definidas pelo usuário. A linguagem destaca-se pela simplicidade de sua gramática, dividida em dois tipos de expressões. *Left-expressions* englobam variáveis, construtores e um operador de duplicação/igualdade ($[\cdot]$). Desses, variáveis e construtores atuam da forma usual, enquanto o operador $[\cdot]$ é utilizado para testes de igualdade e para possibilitar a duplicação de variáveis de forma explícita.

O restante das expressões engloba *left-expressions* (l), ligações através de uma expressão *let*, testes condicionais através da expressão *case* e um construtor *rlet*, utilizado para invocar a semântica inversa de uma função (YOKOYAMA; AXELSEN; GLÜCK, 2011). De forma geral, *left-expressions* são utilizadas no lado esquerdo de definições, aparecendo como nomes em ligações *let* e padrões em testes *case*. Uma função é definida pela construção $fl = e$ e um programa é constituído de uma coleção de funções.

Destaca-se também a codificação da reversibilidade através de regras de sintaxe e da semântica operacional, evitando a criação de um sistema de tipos. Quanto a sintaxe, variáveis não podem ser duplicadas implicitamente, aparecendo uma única vez em cada padrão. Além disso, seu uso deve ser linear: se uma variável é utilizada no lado esquerdo de uma expressão, a mesma variável deve aparecer uma única vez no lado direito (YOKOYAMA; AXELSEN; GLÜCK, 2011). Para qualquer função f bem formada, a função inversa de f pode ser obtida pela expressão

$$rlet\ y = f\ x\ in\ y$$

Quanto a semântica, a aplicação de valores a funções é tratada via *pattern-matching* entre valores e *left-expressions*, utilizando a substituição em variáveis. Em especial, o operador de duplicação/igualdade é seu próprio inverso, fazendo com que uma duplicação na ordem usual seja revertida com um teste de igualdade na ordem inversa. As

regras de avaliação da linguagem são especificadas através de semântica operacional e codificam ambas as ordens de execução em um mesmo conjunto de regras. Esse comportamento é evidenciado pela comparação das regras para as expressões *let* e *rlet*, onde o operador \triangleleft representa um julgamento via *pattern-matching*:

$$\frac{\sigma_{in} \vdash_q f l_{in} \hookrightarrow v_{out} \quad v_{out} \triangleleft l_{out} \rightsquigarrow \sigma_{out} \quad \sigma_{out} \uplus \sigma_e \vdash_q e \hookrightarrow v}{\sigma_{in} \uplus \sigma_e \vdash_q \text{let } l_{out} = f l_{in} \text{ in } e \hookrightarrow v} \text{LetExp}$$

$$\frac{v_{in} \triangleleft l_{in} \rightsquigarrow \sigma_{in} \quad \sigma_{out} \vdash_q f l_{out} \hookrightarrow v_{in} \quad \sigma_{out} \uplus \sigma_e \vdash_q e \hookrightarrow v}{\sigma_{in} \uplus \sigma_e \vdash_q \text{rlet } l_{in} = f l_{out} \text{ in } e \hookrightarrow v} \text{RLetExp}$$

De forma simplificada, enquanto a avaliação usual das premissas é realizada da esquerda para a direita, resultando em um valor v , a regra semântica da expressão *rlet* força a avaliação das demais expressões na ordem inversa, fornecendo para a função f o valor resultante da avaliação usual (v_{in}). Dessa maneira, avaliar a função conhecendo seu valor de saída resulta no mapeamento σ entre as variáveis originais e os valores utilizados como argumentos na geração do valor v_{in} .

2.4 LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL QUÂNTICA

A computação clássica emprega entidades abstratas chamadas bits para representar teoricamente os sistemas físicos que implementam as operações computacionais. O conjunto de estados possíveis para bits é formado pelos valores 0 e 1, que podem ser representados por vetores bidimensionais na forma $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ e $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Utilizando a representação vetorial, uma combinação de n bits é dada matematicamente pelo produto tensor (\otimes) dos estados de cada bit (MERMEN, 2003). Uma combinação de 3 bits é mostrada na equação 2.3 (MERMEN, 2003).

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \otimes \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \otimes \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_1 x_2 x_3 \\ x_1 \cdot y_1 \cdot z_1 \\ x_1 \cdot y_1 \cdot z_1 \\ x_1 \cdot y_2 \cdot z_1 \\ x_2 \cdot y_1 \cdot z_1 \\ x_2 \cdot y_1 \cdot z_2 \\ x_2 \cdot y_2 \cdot z_1 \\ x_2 \cdot y_2 \cdot z_2 \end{pmatrix}. \quad (2.3)$$

De forma análoga, a teoria da computação quântica faz uso de objetos matemáticos para abstrair as especificidades do sistema físico utilizado para a implantação das operações (MERMEN, 2003). Estes objetos são denominados bits quânticos, ou *qubits*. O estado de um qubit é comumente dado na notação *Dirac-ket* $|_ \rangle$, sendo o conjunto de estados possíveis formado pelos estados $|0\rangle$ e $|1\rangle$ correspondentes aos estados de bits clássicos, estendido por estados denominados superposições (NIELSEN; CHUANG, 2002). Uma superposição é uma combinação linear de estados representada por $\alpha|0\rangle + \beta|1\rangle$, onde α e β são números complexos determinando a amplitude dos estados

associados (AARONSON, 2013). Vetorialmente, o estado genérico de um qubit em superposição é representado pelo vetor $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$.

Em um conjunto geral de n qubits em superposição, não é possível determinar o estado individual de cada qubit (MERMIN, 2003), sendo o par $(\alpha|0\rangle + \beta|1\rangle)(\gamma|0\rangle + \delta|1\rangle)$ reescrito na forma:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \cdot \gamma \\ \alpha \cdot \delta \\ \beta \cdot \gamma \\ \beta \cdot \delta \end{pmatrix} \equiv \alpha \cdot \gamma |00\rangle + \alpha \cdot \delta |01\rangle + \beta \cdot \gamma |10\rangle + \beta \cdot \delta |11\rangle.$$

Desta forma, o estado geral de um conjunto de n -qubits pode ser representado pelo somatório⁵ apresentado na equação 2.4.

$$|\psi\rangle = \sum_{0 \leq x < 2^n} \alpha_x |x\rangle_n. \quad (2.4)$$

A extração de valores de um qubit em superposição é realizada através da medição (*measurement*). A medição de um estado $|\psi\rangle$ de n -qubits resulta em um inteiro x onde $0 \leq x < 2^n$ (MERMIN, 2003). A relação entre o valor de x e o estado $|\psi\rangle$ é a probabilidade de se obter o resultado x definida por:

$$p_x = |\alpha_x|^2.$$

Todas as operações utilizadas para modificar o estado de qubits são transformações lineares, codificadas na forma de matrizes unitárias. O conjunto disponível de operações é construído através do produto de transformações simples que agem em apenas 1 ou 2 qubits.

Da mesma forma que circuitos clássicos são construídos a partir da combinação de portas lógicas que atuam em um conjunto restrito de bits, circuitos (e algoritmos) quânticos são construídos a partir da combinação de transformações unitárias.

Uma operação clássica que compõe circuitos quânticos é a transformada de Hadamard (*Hadamard-gate*), definida pela equação 2.5.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \quad (2.5)$$

Esta operação é uma forma de colocar um qubit em superposição. Sua aplicação leva um qubit do estado $|0\rangle$ ao estado $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$, denotado simplesmente por $|+\rangle$ e um qubit

⁵Onde \mathbf{x} é a representação decimal de um valor binário composto por n dígitos. Como exemplo, os valores 00, 01, 10 e 11 são representados na forma: $|0\rangle_2, |1\rangle_2, |2\rangle_2, |3\rangle_2$

do estado $|1\rangle$ ao estado $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$, representado por $|-\rangle$. A representação vetorial desses estados é dada por $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ e $(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$, respectivamente.

Uma das características da computação quântica que fornece vantagens de desempenho sobre o paradigma clássico é o fenômeno conhecido por paralelismo quântico. Em essência, o paralelismo quântico permite a avaliação simultânea da aplicação de múltiplos argumentos de uma função (NIELSEN; CHUANG, 2002). Formalmente, para qualquer função f possuindo n bits como argumentos de entrada, gerando um único bit na saída, é possível construir uma transformação unitária U_f equivalente que recebe $n + 1$ qubits na entrada e gera $n + 1$ qubits na saída. Nessa transformação, os n primeiros qubits compõem o chamado registro de dados, sendo o último qubit chamado de registro alvo (YING, 2016).

Antes de aplicado a função, o conjunto de qubits é preparado através dos passos:

1. Todos os qubits são inicializados no estado $|0\rangle$;
2. A transformada de Hadamard é aplicada paralelamente ao registro de dados, procedimento denotado por denotada por $H^{\otimes n}$, gerando a superposição

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n.$$

A aplicação do estado $|\psi\rangle|0\rangle$ é dada pela equação 2.6 (adaptada de Ying (2016)).

$$|\psi\rangle|0\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n |0\rangle \xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n |f(x)\rangle. \quad (2.6)$$

Como uma ilustração, considera-se a aplicação de uma função $\text{and}: \text{bool}^2 \rightarrow \text{bool}$ utilizando o paralelismo quântico. Uma transformação $U_{\text{and}}: \text{bool}^3 \rightarrow \text{bool}^3$ é especificada de forma que a avaliação de $U_{\text{and}}(x)$ é dada pelos passos:

1. Os qubits são inicializados com estados $|000\rangle$;
2. A aplicação paralela de Hadamard ao registro de dados $|00\rangle$ gera

$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right) \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right) \xrightarrow{H^{\otimes 2}} \frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle);$$

3. Por fim, a aplicação de U_{and} ao registro de dados resulta no estado

$$\frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle) |0\rangle \xrightarrow{U_{\text{and}}} \frac{1}{2} (|00\rangle|0\rangle + |01\rangle|0\rangle + |10\rangle|0\rangle + |11\rangle|1\rangle).$$

Como pode ser observado na extensão do exemplo, a superposição gerada ao final da avaliação contém informações sobre a aplicação de and a todas as combinações possíveis de 2 bits.

A generalização do fenômeno para qualquer valor do qubit alvo (y) é dada pela equação 2.7. O símbolo \oplus representa a adição módulo 2, definida em um contexto de aplicação a bits por: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$ e $1 \oplus 1 = 0$.

$$|\psi\rangle|y\rangle \xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n |y \oplus f(x)\rangle. \quad (2.7)$$

O algoritmo de Deutsch-Jozsa ilustra como é possível extrair a informação armazenada no estado resultante de uma aplicação do paralelismo quântico. Esse algoritmo descreve uma solução quântica para o problema de determinar se uma função $f: \text{bool}^n \rightarrow \text{bool}$, possui comportamento constante, para qualquer x , a aplicação $f(x)$ resulta em 1, ou balanceado, apresentando o valor de saída 0 para metade dos argumentos de entrada e o valor 1 para os demais argumentos. Esse algoritmo toma proveito da característica da interferência quântica, sendo capaz de unificar a informação resultante de $f(x)$ com o valor mantido no registro de dados (YING, 2016). O algoritmo é executado com os passos:

1. Os n qubits do registro de dados são inicializados em $|0\rangle$, com o qubit alvo inicializado em $|1\rangle$;
2. A transformada de Hadamard é aplicada a todos os $n + 1$ qubits:

$$|\psi\rangle|1\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n \left[\frac{0-1}{\sqrt{2}} \right];$$

3. A transformação unitária U_f é aplicada aos qubits:

$$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle_n \left[\frac{0-1}{\sqrt{2}} \right] \xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_x (-1)^{f(x)} |x\rangle_n \left[\frac{0-1}{\sqrt{2}} \right];$$

4. A transformação de Hadamard é aplicada novamente ao registro de entrada:

$$\xrightarrow{H^{\otimes n}} \sum_z \sum_x \frac{(-1)^{f(x)+x \cdot z}}{2^n} |z\rangle \left[\frac{0-1}{\sqrt{2}} \right];$$

5. Os qubits do registro de dados são medidos:

$$\frac{1}{2^n} \left| \sum_x (-1)^{f(x)} \right|^2 = \begin{cases} 1 & \text{se } f \text{ é constante.} \\ 0 & \text{se } f \text{ é balanceada.} \end{cases}$$

A Figura 2.11 apresenta duas derivações simplificadas buscando clarificar o resultado apresentado no passo 3 do algoritmo. Na figura, apresenta-se a aplicação de U_f em um registro de dados consistindo de um único qubit no estado $|1\rangle$ para os dois

possíveis resultados de $f(1)$. É visível que resultado de ambas as derivações pode ser representado pela expressão $(-1)^{f(x)}|1\rangle|-\rangle$ e o mesmo é válido com o registro de dados no estado $|0\rangle$. Considerando os resultados apresentados até agora, verifica-se que a aplicação $U_f|+\rangle|-\rangle = \frac{1}{\sqrt{2}}U_f|0\rangle|-\rangle + \frac{1}{\sqrt{2}}U_f|1\rangle|-\rangle$, leva ao resultado final expressado por $\frac{1}{\sqrt{2}}\left((-1)^{f(0)}|0\rangle|-\rangle + (-1)^{f(1)}|1\rangle|-\rangle\right)$, equivalente ao passo 3 do algoritmo para o valor de $n = 1$.

Figura 2.11 – Derivação simplificada do passo 3 do algoritmo de Deutsch-Jozsa.

$ \begin{aligned} f(1) = 0 &\Rightarrow U_f(1\rangle \left \frac{0-1}{\sqrt{2}} \right\rangle) \\ &= \frac{1}{\sqrt{2}}U_f(1\rangle 0\rangle) + \frac{-1}{\sqrt{2}}(1\rangle 1\rangle) \\ &= \frac{(1\rangle 0 \oplus f(1)\rangle) - (1\rangle 1 \oplus f(1)\rangle)}{\sqrt{2}} \\ &= \frac{1}{\sqrt{2}}(1\rangle 0\rangle) - \frac{1}{\sqrt{2}}(1\rangle 1\rangle) \\ &\equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \\ &\equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \equiv 1\rangle \left \frac{0-1}{\sqrt{2}} \right\rangle \end{aligned} $	$ \begin{aligned} f(1) = 1 &\Rightarrow U_f(1\rangle \left \frac{0-1}{\sqrt{2}} \right\rangle) \\ &= \frac{1}{\sqrt{2}}U_f(1\rangle 0\rangle) + \frac{-1}{\sqrt{2}}(1\rangle 1\rangle) \\ &= \frac{(1\rangle 0 \oplus f(1)\rangle) - (1\rangle 1 \oplus f(1)\rangle)}{\sqrt{2}} \\ &= \frac{1}{\sqrt{2}}(1\rangle 1\rangle) - \frac{1}{\sqrt{2}}(1\rangle 0\rangle) \\ &\equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\ &\equiv - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \equiv - 1\rangle \left \frac{0-1}{\sqrt{2}} \right\rangle \end{aligned} $
---	--

Fonte: Adaptadas de Amin e Labelle (2008).

Linguagens de programação quântica devem exibir construções adequadas para a expressão de estados e transformações de qubits, incluindo a capacidade de emaranhamento de qubits e inversão das operações. Uma abordagem comum na criação dessas é a adoção de uma linguagem usual pertencente a um paradigma de programação específico, enriquecendo-a com recursos específicos para a expressão e o tratamento de operações quânticas. A incorporação de construtos quânticos em um programa deve levar em conta a sua aplicação em estruturas de controle de fluxo, tendo em vista que a natureza probabilística das operações quânticas não é bem representada pelo formalismo usual (ÖMER, 1998).

O controle de fluxo refere-se as operações realizadas para levar um objeto de seu estado inicial ao seu estado final, incluindo desvios condicionais e estruturas de repetição. Um exemplo de estrutura de controle é a expressão *case*, na qual diversos fluxos de programa são ligados a expressões guardas distintas. A escolha de fluxo é feita através da avaliação de uma variável de teste e sua correspondência com uma das guardas. Em um

contexto quântico, a avaliação da variável teste não é uma operação trivial, já que dados quânticos não podem ser manipulados da maneira clássica.

Duas abordagens existem para a conciliação do controle de fluxo com a estrutura quântica (YING, 2016). A primeira consiste em visualizar os programas como a manipulação de circuitos quânticos (SABRY; VALIRON; VIZZOTTO, 2017). Nesse caso, os circuitos são considerados um tipo especial de dados, sujeitos a aplicação de operações quânticas universais. Operações de controle de fluxo são estendidas porém mantidas fora do contexto quântico, não permitindo a execução de uma superposição da operação (SELINGER, 2004). No exemplo da expressão *case*, uma extensão do construto é possível para os casos em que a variável de teste é uma variável quântica, denominado *measurement-based case* (YING, 2016). Nesse construto, a avaliação da variável de teste é realizada após a medição da variável quântica. Como a medição de um estado quântico resulta em estados clássicos, o controle de fluxo permanece no contexto clássico.

Exemplos dessa abordagem são vistos nas obras de SELINGER (2004) e Ömer (1998). Cunhando a frase "dados quânticos, controle clássico", o trabalho de Sellinger especifica uma linguagem funcional para a programação quântica, utilizando um sistema de tipos com checagem estática para garantir o bom comportamento das operações quânticas. Como um segundo exemplo, a linguagem QCL (ÖMER, 1998) é uma linguagem de programação imperativa para a programação quântica, contendo operações e tipos clássicos, bem como registradores quânticos e funções para sua manipulação. Em ambos os casos, o controle de fluxo é explicitamente clássico. Isso significa que uma expressão condicional cuja variável de controle é uma superposição, só será executada no resultado de uma medição da variável.

Uma segunda abordagem para a criação de linguagens quânticas considera a criação de um controle de fluxo quântico. Com a noção de controle quântico embutida na linguagem, entende-se que um circuito quântico pode ser manipulado como um programa completo. Operações de controle de fluxo são então definidas no contexto quântico, atuando sem a necessidade de medição. O cálculo lambda quântico de Tonder (2004) é uma linguagem que aponta nessa direção. A linguagem descreve uma versão do cálculo lambda para a computação quântica, incluindo a descrição de uma semântica com raciocínio equacional. Um dos pontos marcantes da linguagem é a escolha por manter a semântica puramente em um contexto quântico, evitando a definição de operações de medição e a descrição de programas clássicos.

Outra linguagem que ilustra o controle quântico é a linguagem funcional QML (ALTENKIRCH; GRATTAGE, 2005), que apresenta a definição de um condicional quântico, permitindo a execução de testes condicionais sem a extração para o contexto clássico. Nesse caso, a operação condicional controlada por uma variável em superposição é avaliada para todos os valores da superposição, gerando fluxos distintos para o programa.

Um questionamento ainda em aberto é a definição de operações de repetição no contexto quântico. A descrição de uma linguagem apresentando recursão quântica é dada por Ying (2016), cujo trabalho descreve uma linguagem imperativa capaz de codificar recursão no contexto quântico, utilizando um sistema de *moedas* para descrever o comportamento recursivo. Similarmente, a linguagem utilizada como base deste trabalho possibilita a expressão de recursão quântica através do uso de listas, descritas como análogas ao sistema de *moedas* descrito por Ying (SABRY; VALIRON; VIZZOTTO, 2017).

3 UMA LINGUAGEM FUNCIONAL REVERSÍVEL COM CONTROLE QUÂNTICO

Este capítulo apresenta formalmente a linguagem utilizada como base para a implementação deste trabalho. Serão apresentados os conceitos básicos da sintaxe e semântica da linguagem, incluindo sistemas de tipos, a codificação da reversibilidade e a expressão de termos e controle quântico.

3.1 PATTERN-MATCHING SIMÉTRICO

Em linguagens funcionais, o controle de fluxo de um programa é expressado através de *pattern-matching* (casamento de padrões). Nesse modelo de controle, a análise e seleção de componentes a serem executados é realizado através do casamento de padrões especificados no código com valores obtidos através da avaliação dos termos da linguagem (THOMPSON, 1991).

A expressão do controle de fluxo via *pattern-matching* possibilita a verificação de duas propriedades importantes: **padrões não sobrepostos** e **cobertura exaustiva de tipos de dados**. A primeira propriedade garante que, para qualquer valor, existe apenas um único padrão no qual o valor pode se encaixar, garantindo que a execução de uma função sempre terá a mesma avaliação quando aplicada aos mesmos parâmetros (SEBESTA, 2009). Já a segunda indica que, ao especificar um tipo de entrada para a função, a definição de padrões para a execução da mesma deve contemplar todo o domínio representado pelo tipo de entrada. Por exemplo, ao definir uma função com tipo de entrada `(bool, bool)`, os padrões definidos devem contemplar os 4 possíveis pares de valores: `(true, false)`, `(true, true)`, `(false, false)`, `(false, true)`.

Quando impostas por linguagens de programação existentes, essas propriedades aplicam-se apenas ao lado esquerdo das definições de funções, sendo as expressões que definem o comportamento da função regidas por regras distintas. No caso de uma linguagem reversível, ambos os lados da definição de uma função devem observar as duas propriedades (SABRY; VALIRON; VIZZOTTO, 2017). A combinação dessas duas propriedades em ambos os lados das definições, permite a especificação de funções reversíveis simples através da combinação de padrões formando um *pattern-matching* simétrico.

Exemplo 1. Simmetric pattern-matching:

```
1   rf :: Either Int Int <-> (Bool, Int)
2   rf (Left 0)           = (True, 0)
3   rf (Left (n+1))      = (False, n)
4   rf (Right n)         = (True, n+1)
```

No exemplo 1 é demonstrada a utilização de um pattern-matching simétrico através de uma função construída com uma sintaxe baseada na linguagem Haskell. Nesta função, o operador \leftrightarrow ilustra a possibilidade de execução da função em ambos os sentidos. Quando executada recebendo como parâmetros um valor do tipo (Either Int Int) , a função `rf` executa da forma tradicional (*left-to-right*). O inverso da função poderia ser executado fornecendo um valor composto de um par do tipo $(\text{Bool}, \text{Int})$.

A noção de *pattern-matching* simétrico como base de expressão de funções reversíveis é suportada pela noção de isomorfismo de tipos destacada em James e Sabry (2012).

No caso do exemplo, os tipos Either Int Int e $(\text{Bool}, \text{Int})$ são isomórficos pois é possível estabelecer um mapeamento bijetivo (1 para 1) entre todos os habitantes de ambos os tipos. Esta relação de isomorfismo reforça o cumprimento das propriedades destacadas acima, já que, para garantir a relação isomórfica, todos os membros do conjunto definido pelos tipos devem ser unicamente contemplados na definição da função.

A linguagem funcional reversível é definida a partir de duas camadas de tipos: uma contendo os tipos de valores e de termos, e uma camada referente aos isomorfismos de tipos, que encapsulam as computações da linguagem. A gramática da linguagem apresentando o conjunto de tipos, termos e valores formalizados é apresentada na Figura 3.1.

Figura 3.1 – Gramática de uma linguagem reversível com controle quântico.

Value and term types	$a, b ::= \mathbb{1} \mid a \oplus b \mid a \otimes b \mid [a]$
Iso types	$T ::= a \leftrightarrow b \mid (a \leftrightarrow b) \rightarrow T$
Values	$v ::= () \mid x \mid \text{inj}_l v \mid \text{inj}_r v \mid \langle v_1, v_2 \rangle$
Combination	$e ::= v \mid e_1 + e_2 \mid \alpha e$
Products	$p ::= () \mid x \mid \langle p_1, p_2 \rangle$
extended Values	$e ::= v \mid \text{let } p_1 = \omega p_2 \text{ in } e$
Isos	$\omega ::= \{ \mid v_1 \leftrightarrow e_1 \mid v_2 \leftrightarrow e_2 \dots \} \mid \lambda f. \omega \mid f \mid \omega_1 \omega_2 \mid \mu f. \omega$
Terms	$t ::= () \mid x \mid \text{inj}_l t \mid \text{inj}_r t \mid \langle t_1, t_2 \rangle \mid \omega t \mid \text{let } p = t_1 \text{ in } t_2 \mid t_1 + t_2 \mid \alpha \cdot t$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

O tipo $\mathbb{1}$ é um tipo primitivo para termos, sendo seu único valor possível representado por $()$. Os demais tipos são construídos através da composição utilizando os operadores \oplus e \otimes . Um tipo definido pela soma $(a \oplus b)$ permite a criação de valores distinguíveis a partir do uso dos construtores inj_l e inj_r , enquanto o operador de produtos $(a \otimes b)$ permite a construção de tuplas de valores como $\langle v_1, v_2 \rangle$. Além disso, a linguagem engloba ainda um tipo recursivo $([a])$ que possibilita a construção de listas.

Os termos da linguagem possibilitam o uso de variáveis, construtores inj_l e inj_r , tuplas, aplicação de isos, recursões e let-binding. Para a representação de operações

quânticas, é permitida a combinação linear de termos e valores, bem como a multiplicação por escalares α , restritos a condições de normalização. A inclusão dos termos para a computação quântica será apresentada em maiores detalhes posteriormente.

Os valores da linguagem são um subconjunto dos termos, enquanto os valores estendidos (*extended values*) são valores nos quais as variáveis livres foram substituídas com o resultado da aplicação de um ou mais isos (SABRY; VALIRON; VIZZOTTO, 2017).

Isos podem ser compreendidos como a descrição de computações fechadas em relação a entrada. A linguagem permite, além da especificação de funções de primeira ordem utilizando cláusulas da forma $\{|v_1 \leftrightarrow e_1|v_2 \leftrightarrow e_2\dots\}$, a criação de funções de segunda ordem com abstrações lambda e aplicação de isos, utilizando f para representar variáveis de isos. Para aumentar a expressividade da linguagem, construção de funções recursivas são possibilitadas pela inclusão de um operador de ponto fixo $\mu.f.\omega$.

Por representarem computações fechadas, isos podem ser duplicadas ou apagadas sem prejudicar a reversibilidade da linguagem (SABRY; VALIRON; VIZZOTTO, 2017). Essa distinção faz com que essas construções sejam submetidas a dois tipos distintos: o primeiro, representado pela seta bidirecional \leftrightarrow descreve operações de primeira ordem, computações reversíveis. Já a seta unidirecional \rightarrow é utilizada na tipagem de operações de mais alta ordem ($\lambda f.\omega$), parametrizadas por uma ou mais isos representadas pela variável f . A distinção de tipos é importante para a definição da inversão de isos:

$$(a \leftrightarrow b)^{-1} := (b \leftrightarrow a), \quad ((a \leftrightarrow b) \rightarrow T)^{-1} := ((b \leftrightarrow a) \rightarrow T^{-1}).$$

A primeira equação apresenta a inversão de isos de primeira ordem: o tipo utilizado como entrada na ordem usual é tomado como o tipo resultante da operação inversa. No caso das abstrações $\lambda f.\omega$, a inversão não tem efeito na seta representado a composição dos isomorfismos. A inversão atua diretamente no tipo individual de cada iso que parametriza a abstração:

$$((a \leftrightarrow c) \rightarrow (a \leftrightarrow b) \rightarrow (b \leftrightarrow c))^{-1} := (c \leftrightarrow a) \rightarrow (c \leftrightarrow b) \rightarrow (c \leftrightarrow b).$$

O procedimento de avaliação será detalhado posteriormente, note-se por hora que, considerando uma abstração $(\lambda f.\lambda g.\omega)$ com o tipo apresentado na equação anterior, a aplicação de isos ω_1 e ω_2 com tipos adequados transforma a composição em um único isomorfismo com tipo $(a \leftrightarrow c)$, inverso $(c \leftrightarrow a)$.

3.2 SISTEMA DE TIPOS

As regras do sistema de tipos da linguagem são divididas em dois conjuntos distintos. O primeiro é dedicado a tipagem de termos, que devem ser tratados de forma linear: não podem ser apagados nem duplicados. Isso implica que variáveis presentes a esquerda de uma cláusula devem aparecer uma única vez no lado direito da definição da cláusula, o mesmo vale na direção oposta.

O segundo conjunto trata singularmente da tipagem de isomorfismos, que podem ser apagados ou duplicados sem prejudicar a reversibilidade da linguagem (SABRY; VALIRON; VIZZOTTO, 2017). Termos e valores podem conter tanto variáveis de termo da forma $x : a$, quanto variáveis de isos na forma $f : T$. Por esse motivo, faz-se a utilização de dois contextos de tipos na definição das regras: o contexto Δ realiza um mapeamento de nomes e tipos das variáveis de termos, sendo completamente linear, enquanto o contexto Ψ é utilizado para mapear tipos em variáveis de isos.

As regras de tipos para os valores são apresentadas na Figura 3.2. A tipagem dos valores e termos é feita a partir da composição dos tipos das subexpressões presentes na construção.

Os demais termos da linguagem possuem regras de tipo mais envolvidas, apresentadas na Figura 3.3. Para a aplicação de termos em isos, é necessário checar o tipo da iso provida na construção do termo, relacionado ao tipo encontrado para o termo t . A aplicação só é bem construída quando ω for um conjunto de cláusulas ou uma variável de isos com tipo de função ($a \leftrightarrow b$), e quando o tipo do termo é compatível com o tipo do lado esquerdo da definição das cláusulas (a). Nesse caso, o tipo completo da aplicação de t em ω é o tipo do resultado do isomorfismo (b).

A tipagem das expressões `let` é realizada com o conhecimento de que a expressão é restrita a produtos de termos, atuando como um açúcar sintático para a composição de isos (SABRY; VALIRON; VIZZOTTO, 2017). Com isso em mente, a regra de tipo precisa induzir o tipo do termo t_1 , adicionando as variáveis do par $\langle x, y \rangle$ ao contexto Δ antes de prosseguir com a verificação do termo t_2 . O tipo verificado de t_2 é atribuído a expressão `let`.

No contexto da combinação de termos, a regra de tipos garante que a combinação só é possível para termos de mesmo tipo. A utilização do escalar α não tem influência na

Figura 3.2 – Regras de tipos para valores.

$$\begin{array}{c}
 \frac{\Delta; \Psi \vdash_v t : a}{\Delta; \Psi \vdash_v \text{inj}_l t : a \oplus b} \quad \frac{\emptyset; \Psi \vdash_v () : \mathbb{1}}{\Delta; \Psi \vdash_v t : b} \quad \frac{x : a; \Psi \vdash_v x : a}{\Delta; \Psi \vdash_v \langle t_1, t_2 \rangle : a \otimes b} \\
 \frac{\Delta; \Psi \vdash_v t : b}{\Delta; \Psi \vdash_v \text{inj}_r t : a \oplus b}
 \end{array}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

tipagem do termo associado, sendo o tipo da expressão $\alpha \cdot t$ o tipo do termo t .

A avaliação dos tipos de isomorfismos pode ser visualizada na Figura 3.8.

A regra de tipos para cláusulas da forma $\{ | v_1 \leftrightarrow e_1 | v_2 \leftrightarrow e_2 \dots \}$, codifica as duas propriedades necessárias para a reversibilidade. Primeiramente, verifica-se de que todos os padrões especificados são do tipo especificado para o iso. Nesta regra $OD_a\{S\}$ representa uma decomposição ortogonal do tipo a , onde o conjunto S de padrões cobre todas as possibilidades de valores do tipo a (cobertura exaustiva de tipos), sendo todos os padrões ortogonais em relação aos seus pares ($\forall i \neq j \cdot v_i \perp v_j$), garantindo a não repetição de padrões.

O conjunto $OD_{\mathbb{1}}\{\{\}\}$, definido para o tipo primitivo $\mathbb{1}$ forma a base para a decomposição ortogonal dos demais tipos. Por exemplo, o tipo $(\mathbb{1} \oplus \mathbb{1})$ gera a decomposição ortogonal $OD_{\mathbb{1} \oplus \mathbb{1}}\{\text{inj}_l(), \text{inj}_r()\}$. Generalizando, a decomposição ortogonal de tipos de soma $(a \oplus b)$ é obtida pela equação 3.1

$$OD_{(a \oplus b)} = \{\text{inj}_l v \mid v \in OD_a\} \cup \{\text{inj}_r v' \mid v' \in OD_b\}. \quad (3.1)$$

Os demais valores tem suas decomposições ortogonais especificadas pelas regras:

$$OD_a\{x\} = \{x\}$$

$$OD_{(a \times b)} = \{\langle v_1, v_2 \rangle \mid v_1 \in OD_a, v_2 \in OD_b, \text{FV}(v_1) \cap \text{FV}(v_2) = \emptyset\},$$

onde $\text{FV}(v_1)$ indica o conjunto de variáveis livres de v_1 . O conjunto OD_b^{ext} é construído através da decomposição ortogonal do valor ao final de uma expressão let. Para o valor estendido **let** $p = \omega p_2$ **in** v com tipo b , a decomposição ortogonal equivalente seria $OD_b^{ext} = OD_b\{v\}$.

Por fim, a regra impõe uma condição necessária para que uma combinação de valores seja válida no contexto quântico: a matriz formada pelos escalares α deve ser unitária.

Figura 3.3 – Regras de tipos para termos.

$$\frac{\Psi \vdash_{iso} \omega : a \leftrightarrow b \quad \Delta; \Psi \vdash_v t : a}{\Delta; \Psi \vdash_v \omega t : a} \quad \frac{\Psi \vdash_{iso} t_1 : a \otimes b \quad \Delta, x : a, y : b; \Psi \vdash_v t_2 : c}{\Delta; \Psi \vdash_v \text{let}\langle x, y \rangle = t_1 \text{ in } t_2 : c}$$

$$\frac{\Delta; \Psi \vdash_v t : a}{\Delta; \Psi \vdash_v \alpha \cdot t : a} \quad \frac{\Delta; \Psi \vdash_v t_1 : a \quad \Delta; \Psi \vdash_v t_2 : a}{\Delta; \Psi \vdash_v t_1 + t_2 : a}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017)

3.3 SEMÂNTICA DA LINGUAGEM

A avaliação de *pattern-matching* na linguagem é realizada através do casamento de um padrão p com um valor v , representado formalmente pela expressão $\sigma[p] = v$. O casamento é realizado através da atribuição de valores a variáveis, sendo σ um mapeamento parcial de um conjunto finito de variáveis a um conjunto de valores (SABRY; VALIRON; VIZZOTTO, 2017). As regras de avaliação são apresentadas na Figura 3.4.

De forma geral, a avaliação dos padrões é realizada através da aplicação das regras nas subexpressões na busca de variáveis ou do valor $()$. No caso da avaliação de pares, a premissa $\text{supp}(\sigma_1) \cap \text{supp}(\sigma_2)$ é necessária para manter a reversibilidade da linguagem. Nessa premissa, $\text{supp}(\sigma)$ denota o conjunto de variáveis mapeadas em σ , chamado de suporte. A existência de uma intersecção entre os conjuntos de suporte invalidaria a condição do uso linear de variáveis, violando a reversibilidade.

Para os demais termos, a redução observa a estratégia *call-by-value*, sendo apresentada na Figura 3.6. Nas regras, a notação $\sigma(p)$ equivale a substituição das variáveis de p pelos respectivos valores mapeados em σ . A notação $M[N \setminus x]$ é similar a substituição do cálculo Lambda, descrevendo uma operação na qual todas as ocorrências livres de x em M são substituídas por N .

As regras de avaliação são definidas através da semântica operacional da linguagem e de um contexto aplicativo ($C[-]$), apresentado na Figura 3.5. Esse contexto codifica a ordem na qual os termos da linguagem são reduzidos, enquanto a semântica operacional apresenta os detalhes da computação. Como um exemplo, a expressão $\{\dots\}C[-]$ denota que qualquer termo aplicado a um conjunto de padrões será reduzido antes da avaliação da aplicação. Isso é necessário já que funções na linguagem aceitam apenas valores como argumentos. Com a definição do contexto, a regra de avaliação para a aplicação menciona apenas o caso em que o argumento é um valor. O mesmo é válido para a definição das ligações let, onde o contexto de avaliação nos garante que a variável p é ligada a um valor e não a um termo qualquer.

É possível exemplificar a criação de funções na linguagem através da definição de um condicional simples **if**, o qual atua em dois isomorfismos do tipo $(a \leftrightarrow b)$ e é definida em termos do isomorfismo $(\mathbb{1} \oplus \mathbb{1} \otimes a \leftrightarrow \mathbb{1} \oplus \mathbb{1} \otimes b)$. No exemplo, **true**¹ é *syntactic sugar*

Figura 3.4 – Regras de *pattern-matching*.

$$\frac{}{\sigma[()] = ()} \quad \frac{\sigma = \{x \rightarrow v\}}{\sigma[x] = v} \quad \frac{\sigma[p] = v}{\sigma[\text{inj}_l p] = \text{inj}_l v} \quad \frac{\sigma[p] = v}{\sigma[\text{inj}_r p] = \text{inj}_r v}$$

$$\frac{\sigma_2[p_1] = v_1 \quad \sigma_1[p_2] = v_2 \quad \text{supp}(\sigma_1) \cap \text{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\sigma[\langle p_1, p_2 \rangle] = \langle v_1, v_2 \rangle}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

Figura 3.5 – Contextos de avaliação.

$$\begin{aligned} C[-] ::= & [-] \mid \text{inj}_l C[-] \mid \text{inj}_r C[-] \mid (C[-])\omega \mid \{\cdot \cdot \cdot\}C[-] \\ & \mid \text{let } p = C[-] \text{ in } t_2 \mid \langle C[-], v \rangle \mid \langle v, C[-] \rangle \end{aligned}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

Figura 3.6 – Semântica Operacional

$$\begin{aligned} & \frac{\sigma[p] = v_1}{\text{let } p = v_1 \text{ in } t_2 \rightarrow \sigma(t_2)} \text{ LetE} \\ & \frac{\sigma[v_i] = v}{\{ \mid v_1 \leftrightarrow t_1 \mid \cdots \mid v_n \leftrightarrow t_n \} v \rightarrow \sigma(t_i)} \text{ IsoApp} \\ & \frac{}{(\lambda f.\omega)\omega_2 \rightarrow \omega[\omega_2/f]} \text{ HIsoApp} \end{aligned}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

para $\text{inj}_l()$ e **false** para $\text{inj}_r()$.

Exemplo 2. $\text{if}:(a \leftrightarrow b) \rightarrow (a \leftrightarrow b) \rightarrow (\mathbb{1} \oplus \mathbb{1} \otimes a \leftrightarrow \mathbb{1} \oplus \mathbb{1} \otimes b)$

$$\text{if} = \lambda g.\lambda h \left(\begin{array}{l} \langle \text{true}, x \rangle \leftrightarrow \text{inj}_l(g x) \\ \langle \text{false}, x \rangle \leftrightarrow \text{inj}_r(h x) \end{array} \right)$$

3.4 COMBINAÇÕES LINEARES PARA O CONTROLE QUÂNTICO

Para a codificação do controle quântico é necessário estabelecer na linguagem uma representação para os valores de qubits associados a sua amplitude, bem como para as matrizes unitárias que codificam as transformações quânticas.

Sabendo que os estados clássicos 0 e 1 podem ser representados por termos (t) do tipo *Bool* na linguagem base, a representação de estados quânticos e suas amplitudes se dá através do termo αt , onde α é um número complexo. O estado geral de um qubit é representado como uma combinação linear dos estados clássicos $\alpha|0\rangle + \beta|1\rangle$, codificada em um termo $\alpha t_1 + \beta t_2$. Os valores de α e β estão sujeitos a condição de normalização $\alpha^2 + \beta^2 = 1$. Como açúcar sintático, termos do tipo $1t$ são representados apenas por t , enquanto combinações $1t_1 + 0t_2$ são denotadas simplesmente por t_1 .

Além da combinação de termos, combinações lineares são adicionadas ao conjunto de valores estendidos da linguagem. Via a escolha de restringir as computações quânticas a um sistema fechado, não são definidos operadores de medição.

¹O tipo booleano \mathbb{B} é definido por $\mathbb{1} \oplus \mathbb{1}$.

No que se refere aos sistemas de tipos, um termo $\alpha \cdot t$ possui o tipo de t , enquanto combinações lineares da forma $\alpha \cdot t_1 + \dots + \beta \cdot t_2$ são permitidas para qualquer t_n desde que o conjunto de termos $\{t_1, \dots, t_n\}$ pertença a um só tipo.

Combinações lineares de termos observam as propriedades de associatividade e comutatividade descritas pela Figura 3.7

Transformações unitárias são representadas através da extensão das cláusulas de isomorfismos, fazendo com que o lado direito das definições seja composto de combinações lineares. Com essa extensão, uma transformação unitária é representada por isos da forma:

$$\left\{ \begin{array}{l} v_1 \leftrightarrow a_{11}v'_1 + a_{21}v'_2 + a_{31}v'_3 \\ v_2 \leftrightarrow a_{12}v'_1 + a_{22}v'_2 + a_{32}v'_3 \\ v_3 \leftrightarrow a_{13}v'_1 + a_{23}v'_2 + a_{33}v'_3 \end{array} \right\},$$

adotando restrições adequadas para garantir que os coeficientes a_{ij} formem uma matriz unitária visualizada pela Figura 3.9. A regra de tipos para a extensão das cláusulas é dada pela Figura 3.8.

Com estas construções a matriz unitária que representa a transformada de Hadamard é definida pela iso $\text{Had}:(\mathbb{B} \leftrightarrow \mathbb{B})$

$$\text{Had} : (\mathbb{B} \leftrightarrow \mathbb{B}) := \left(\begin{array}{l} \text{true} \leftrightarrow \frac{1}{\sqrt{2}}\text{true} + \frac{1}{\sqrt{2}}\text{false} \\ \text{false} \leftrightarrow \frac{1}{\sqrt{2}}\text{true} - \frac{1}{\sqrt{2}}\text{false} \end{array} \right).$$

Devido as propriedades algébricas das combinações lineares, as definições de cláusulas da Figura 3.10 são equivalentes, logo a especificação de cláusulas da primeira forma representa uma matriz unitária, sendo consistente com a regra de tipos da linguagem e servindo como açúcar sintático para a forma com combinações lineares.

Quanto as regras de avaliação, apenas a regra de aplicação de termos a isos é modificada. Lembrando que isos aceitam como argumento apenas valores, aplicações da forma $\text{iso}(\alpha \cdot t_1 + \beta \cdot t_2)$ não são contempladas pela regra existente. Antes da aplicação, uma combinação linear de termos deve ser reduzida a uma forma normal generalizada pela fórmula $\sum_{i=1}^n \alpha_i \cdot v_i$. Em uma iso com o tipo $a \leftrightarrow b$, para qualquer combinação linear

Figura 3.7 – Propriedades algébricas das combinações lineares de termos.

$$\begin{array}{l} \alpha \cdot (e_1 + e_2) = \alpha \cdot e_1 + \alpha \cdot e_2 \\ \alpha \cdot e + \beta \cdot e = (\alpha + \beta) \cdot e \\ 0 \cdot e_1 + e_2 = e_2 \end{array} \qquad \begin{array}{l} 1 \cdot e = e \\ \alpha \cdot (\beta \cdot e) = (\alpha\beta) \cdot e \end{array}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

Figura 3.8 – Regra de tipos para cláusulas com combinações lineares.

$$\begin{array}{c}
\Delta_1 \vdash_v v_1 : a \quad \dots \quad \Delta_n \vdash_v v_n : a \\
\Delta_1 \vdash_v e_1 : b \quad \dots \quad \Delta_n \vdash_v e_n : b \\
OD_a\{v_1, \dots, v_n\} \quad \quad \quad OD_b^{ext}\{e_1, \dots, e_n\}
\end{array}
\begin{array}{c}
\left(\begin{array}{ccc}
\alpha_{11} & \dots & \alpha_{1n} \\
\vdots & & \vdots \\
\alpha_{n1} & \dots & \alpha_{nn}
\end{array} \right) \text{ é unitária}
\end{array}$$

$$\Psi \vdash_{iso} \left\{ \begin{array}{l}
v_1 \leftrightarrow \alpha_{11} \cdot e_1 + \dots + \alpha_{1n} \cdot e_n \\
\vdots \\
v_n \leftrightarrow \alpha_{n1} \cdot e_1 + \dots + \alpha_{nn} \cdot e_n
\end{array} \right\} : a \leftrightarrow b$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

Figura 3.9 – Representação de matrizes unitárias através de isos.

$$\begin{array}{c}
v_1 \quad v_2 \quad v_3 \\
v'_1 \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \\
v'_2 \\
v'_3
\end{array}
\quad
\begin{array}{c}
true \quad false \\
true \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \\
false
\end{array}$$

Generalização de isos.

Visualização de Had.

Fonte: Adaptada de Sabry, Valiron e Vizzotto (2017).

com tipo a , a aplicação da iso a combinação linear é dada por

$$W_v = \sum_{i=1}^n \alpha_i^n \cdot w_i^n,$$

onde α_i^n é a amplitude do valor de entrada v_i e w_i é o valor resultante da aplicação de v_i ao isomorfismo. Em suma, um isomorfismo é aplicado a todos os valores da combinação linear, sendo a amplitude original de cada valor (α_i^n) associada ao resultado da avaliação.

O processo de avaliação é ilustrado através da aplicação dupla da transformada de Hadamard a um valor *false*, denotada na sintaxe da linguagem por *Had* (*Had ff*):

Figura 3.10 – Definições de cláusulas com e sem combinações lineares.

$$\left\{ \begin{array}{l}
v_1 \leftrightarrow e_1 \\
v_2 \leftrightarrow e_2
\end{array} \right\}
\quad
\left\{ \begin{array}{l}
v_1 \leftrightarrow 1 \cdot e_1 + 0 \cdot e_2 \\
v_2 \leftrightarrow 0 \cdot e_1 + 1 \cdot e_2
\end{array} \right\}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

$$\begin{aligned}
\text{Had}(\text{Had } ff) &= \text{Had}\left(\frac{1}{\sqrt{2}} \cdot tt - \frac{1}{\sqrt{2}} \cdot ff\right) \\
&= \sum_{i=1}^2 \alpha_i^2 \cdot w_i^2 = \frac{1}{\sqrt{2}} \cdot (\text{Had } tt) + \left(-\frac{1}{\sqrt{2}}\right) \cdot (\text{Had } ff) \\
&= \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} \cdot tt + \frac{1}{\sqrt{2}} \cdot ff\right) + \left(-\frac{1}{\sqrt{2}}\right) \cdot \left(\frac{1}{\sqrt{2}} \cdot tt - \frac{1}{\sqrt{2}} \cdot ff\right) \\
&= \left(\frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} \cdot tt + \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} \cdot ff\right) + \left(-\frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} \cdot tt + \left(-\frac{1}{\sqrt{2}}\right) \cdot \left(-\frac{1}{\sqrt{2}}\right) \cdot ff\right) \\
&= \left(\frac{1}{2} \cdot tt + \frac{1}{2} ff\right) + \left(-\frac{1}{2} \cdot tt + \frac{1}{2} ff\right) \\
&= ff.
\end{aligned}$$

3.5 RECURSIVIDADE VIA PONTO FIXO

A inclusão de uma estrutura de repetição na linguagem é dada através de funções recursivas atuando em listas de valores, expressada através de um construtor de *ponto fixo* $\mu f.\omega$. No âmbito quântico, a construção de programas recursivos requer uma garantia de terminação, dada pela definição de recursividade estrutural traduzida de Sabry, Valiron e Vizzotto (2017):

Definição 2. Um tipo estruturalmente recursivo é da forma $[a] \otimes b_1 \otimes \dots \otimes b_n$. Seja $\omega = \{v_i \leftrightarrow e_i\}, i \in I$ um isomorfismo tal que $f : a \leftrightarrow b \vdash_{\omega} \omega : a \leftrightarrow c$, onde a é um tipo estruturalmente recursivo. Diz-se $\mu f.\omega$ estruturalmente recursivo se, para cada $i \in I$, o valor v_i é da forma $\langle [], p_1, \dots, p_n \rangle$ ou da forma $\langle h : t, p_1, \dots, p_n \rangle$. No primeiro caso, e_i não contém f entre suas variáveis livres. No último, e_i é da forma $C[f\langle t, p'_1, \dots, p'_n \rangle]$, sendo C um contexto aplicativo: $C[-] ::= [-] \mid \text{let } p = C[-] \text{ in } t \mid \text{let } p = t \text{ in } C[-]$.

Um único tipo recursivo é incluído na linguagem, o tipo de listas $[a]$. O tipo de listas pode ser descrito como $\mathbb{1} \otimes (a \times [a])$, sendo a lista vazia representada pelo valor $\text{inj}_l()$, e o construtor de listas, denotado pelo açúcar sintático $v_1 : l$, escrito como o valor $\text{inj}_r\langle v_1, l \rangle$.

Em conjunto com as regras de cobertura exaustiva de tipos, define-se que ao menos uma das cláusulas cobrirá o caso de listas vazias, no qual a função recursiva não deve aparecer como variável livre (SABRY; VALIRON; VIZZOTTO, 2017). Cláusulas com listas vazias são o ponto de término da recursividade da função. As demais cláusulas devem apresentar valores formados pelo construtor de listas $h : t$, representando *head* e *tail*. Para esses casos, a derivação a direita da cláusula deve aplicar a função recursiva a variável t . Essa condição garante que cada aplicação da função recursiva será dada em uma lista

menor do que a aplicação atual, alcançando eventualmente o caso de terminação da lista vazia.

A regra de avaliação de funções recursivas apresentada na Figura 3.11 descreve o comportamento atribuído ao operador de *ponto fixo*. Sendo f o identificador da função recursiva com tipo $(a \leftrightarrow b)$, a avaliação gera uma função ω carregando n cópias de f em seu corpo. Como a terminação é garantida via a recursividade estrutural, é possível afirmar que n é um número finito de cópias.

Figura 3.11 – Regra de avaliação do operador de ponto fixo.

$$\frac{\psi, f : a \leftrightarrow b \vdash_{\omega} \omega : (a_1 \leftrightarrow b_1) \rightarrow \dots \rightarrow (a_n \leftrightarrow b_n) \rightarrow (a \leftrightarrow b)}{\mu f. \omega \rightarrow \lambda f_1 \dots f_n. (\omega [((\mu f. \omega) f_1 \dots f_n) / f]) f_1 \dots f_n}$$

Fonte: (SABRY; VALIRON; VIZZOTTO, 2017).

Como ilustração, considera-se a função do exemplo 3, mapeando uma transformação m em uma lista de booleanos.

Exemplo 3. $boolMap : (\mathbb{B} \leftrightarrow \mathbb{B}) \rightarrow ([\mathbb{B}] \leftrightarrow [\mathbb{B}])$

$$boolMap = \lambda m. \mu f. \left(\begin{array}{ccc} [] & \leftrightarrow & [] \\ h : t & \leftrightarrow & \text{let } x = m h \text{ in} \\ & & \text{let } y = f t \text{ in } x : y \end{array} \right).$$

A função do exemplo pode ser utilizada para aplicar uma negação booleana a uma lista através do termo $boolMap not [false, true, true]$. Considerando que as cláusulas da função $boolMap$ são copiadas e referidas pelos identificadores f_1, \dots, f_n , a derivação dessa função resulta na construção:

$$\lambda f_1. \lambda f_2. \lambda f_3. \lambda f_4 \left(\begin{array}{ccc} [] & \leftrightarrow & [] \\ h : t & \leftrightarrow & \text{let } x = not h \text{ in} \\ & & \text{let } y = f_1 t \text{ in } x : y \end{array} \right) (f_1)(f_2)(f_3)(f_4).$$

A partir desse ponto, a função é avaliada como um conjunto de aplicações, substituindo os identificadores da forma apropriada. O resultado é uma função finita com comportamento definido para listas de booleanos com um número de elementos menor ou igual a 3. Essa função finita pode, então, ser aplicada a lista de booleanos.

4 IMPLEMENTAÇÃO DO INTERPRETADOR E *TYPECHECKER*

Este capítulo apresenta a implementação do interpretador e *typechecker* da linguagem, provendo uma visão interna a construção do código. No decorrer do capítulo, serão apresentados os módulos que compõem o código-fonte do interpretador, visando contextualizar as escolhas realizadas durante o processo.

A divisão do capítulo inicia com uma seção que apresenta brevemente noções de sintaxe e alguns componentes da linguagem Haskell, escolhida como base para a implementação, visando facilitar a compreensão dos fragmentos de código no decorrer do texto. A implementação é relatada através de seções detalhando a representação da sintaxe abstrata da linguagem, a apresentação componentes utilitários com funções que abrangem mais de um módulo, a implementação do *typechecker* e uma discussão sobre os métodos de implementação da semântica da linguagem. Fragmentos de código são inseridos em pontos que necessitam contextualização, priorizando pontos que clarifiquem as decisões de projeto. O código fonte do projeto é disponibilizado através de um repositório público de código¹.

O capítulo é concluído com a apresentação de exemplos de algoritmos implementados na linguagem base para o trabalho e executados pelo interpretador. Em geral, os exemplos apresentam algoritmos quânticos, sua avaliação por parte do interpretador e a relação de inversão desses, com o intuito de demonstrar a correteude do sistema implementado.

4.1 HASKELL

O interpretador resultante deste trabalho foi implementado utilizando a linguagem Haskell. Haskell é uma linguagem funcional caracterizada pela utilização de funções puras, pelo seu sistema de tipos e pela escolha da avaliação preguiçosa (*lazy evaluation*) (THOMPSON, 1999). Essas características influenciam no projeto de programas na linguagem, sendo sua compreensão importante para o bom entendimento da implementação deste trabalho.

Um dos atrativos da linguagem é a criação de código puro. Em funções puras, não existe interferência externa, mantendo seus dados inalterados e seu comportamento consistente (O'SULLIVAN; GOERZEN; STEWART, 2008). Quando a interação externa é necessária, existe uma distinção entre as funções puras e os fragmentos de código que realizam a interação de forma explícita. Essa propriedade facilita a verificação de correteude

¹O código fonte é encontrado no link <<https://github.com/ezancanaro/QuantumInterpreter>>.

da especificação.

Um programa em Haskell é uma coleção de expressões, generalizadas através de funções (HUDAK et al., 2008). O resultado do programa é obtido através da aplicação de argumentos a uma função, que retorna o resultado obtido. Como característica de linguagens funcionais, o mesmo argumento de entrada retornará sempre o mesmo resultado (ALLEN; MORONUKI, 2017). A Figura 4.1 exemplifica duas formas de declarar funções em Haskell.

Haskell utiliza um sistema de tipos estático com capacidade de inferência, permitindo a omissão da informação de tipo em casos mais simples. Na Figura 4.1 a função `soma2` contém informações sobre argumento de entrada e valor de saída de forma explícita: ambos são do tipo **Int**. Já a função `mult3` omite essa informação, que pode ser obtida via inferência pelo compilador. Haskell permite ainda declarar funções com polimorfismo de parâmetros através de variáveis de tipo. Essas variáveis são nomes iniciados por letras minúsculas presentes na declaração de tipos. Tal funcionalidade permite, por exemplo, a utilização de uma função `count :: [a] -> Int`, que recebe uma lista contendo valores de qualquer tipo, e retorna o número de elementos da lista.

A linguagem permite o uso simples de listas através da sintaxe `[a]`, acompanhada dos operadores `[]` lista vazia, `:` construtor de listas, permitindo a inclusão de um elemento em uma lista e `++` concatenação, permitindo a união de duas listas. As funções `head` e `tail` estão presentes para manipular os elementos da lista.

Tuplas são representadas pela sintaxe `(a,b)` e são acompanhadas das funções `fst` e `snd` para manipulação dos valores inclusos na primeira e segunda posição.

Outro aspecto comum aos programas da linguagem é a utilização de um estilo *point-free*, priorizando a composição de funções e a omissão de parênteses. Para tal, o uso dos operadores de composição `(.)` e aplicação `$` é prevalente em alguns programas.

O operador `(.)` representa a composição de funções, sendo seu tipo definido por `(.) :: (b -> c) -> (a -> b) -> a -> c` (THE UNIVERSITY OF GLASGOW, 2010). Duas formas de composição são ilustradas na Figura 4.2, onde a função `fg` é formada através da composição das funções `f` e `g`. O primeiro caso utiliza o operador para compor as funções `f` e `g`, aplicando o argumento `x` a tal composição. Os parênteses são necessários pois a precedência do operador de composição é menor do que a precedência da aplicação da função `f` ao argumento. Na segunda definição, o uso do operador permite a omissão do argumento de entrada, que é inferido pelo compilador.

Figura 4.1 – Declaração de funções em Haskell.

```

1 soma2 :: Int -> Int
2 soma2 a = 2 + a
3
4 mult3 a = 3 * a

```

Figura 4.2 – Exemplo de composição de funções.

```

1 f :: a -> b
2 ...
3 g :: b-> c
4 ...
5 fg :: a -> c
6 fg x = (g . f) x
7 fg = g . f

```

O operador `$`, por sua vez, é uma maneira conveniente de omitir o uso de parênteses em algumas expressões (ALLEN; MORONUKI, 2017). Esse operador representa a aplicação e é normalmente omitido, sendo a expressão `fun arg` equivalente a `fun $ arg`. A utilidade desse operador encontra-se na sua precedência, que é baixa e associativa a direita (THE UNIVERSITY OF GLASGOW, 2010), fazendo com que a expressão localizada a sua direita seja avaliada antes do restante da expressão. Na Figura 4.3 as funções `ex2` e `ex3` são equivalentes, aplicando o resultado da avaliação da expressão `f x` à função `func2`. No caso do `ex1`, a função `f` é passada como um argumento para `func1`, juntamente com a variável `x`.

Mônadas são utilizadas para a detecção de erros. A mônada `Maybe` é uma delas. Uma função cujo tipo de saída é `Maybe Int` é uma função que, em caso de aplicação bem sucedida retorna um valor inteiro acompanhado pelo construtor `Just`, ou, em casos excepcionais, retorna apenas o construtor de tipo `Nothing`. Seu uso no código permite a especificação de funções em que condições de falha, tal como a busca de uma variável em um contexto de tipos, não devem finalizar a execução do programa.

O uso de `Maybe` permite uma verificação simplificada de erros, mas não fornece ferramentas suficientes para a identificação desses. Para casos em que identificar o motivo da exceção é importante, utiliza-se a construção `Either`. Essa mônada fornece um tipo de soma para a linguagem, permitindo a especificação de funções que avaliem para dois tipos diferentes. Por exemplo, considerando um tipo de dados abstrato `Err` especificado pelo programador, uma função pode apresentar o tipo de saída `Either Err Int`, que pode resultar em um valor válido `Right Int`, ou um resultado de erro `Left Err` que contenha informações úteis para a depuração.

Pela sua natureza funcional, Haskell não utiliza variáveis mutáveis. Toda chamada de função é realizado com uma cópia imutável das variáveis. Nos casos em que um mesmo valor pode ser acessado ou modificado por diversas funções, é necessário transmitir

Figura 4.3 – Uso do operador `$`.

```

1 func1 :: (a->b) -> a -> c
2 func2 :: b -> c
3 ex1 x = func1 f x
4 ex2 x = func2 $ f x
5 ex3 x = func2 (f x)

```

esse valor através das chamadas de função, acrescentando-o como parâmetro. Muitas vezes esse parâmetro precisa ser adicionado a funções que não fazem uso do mesmo para computações relevantes, somente o transmitem a uma função em seu corpo. A recorrência dessas funções ocasiona casos em que o código incorre em complexidade desnecessária, tornando mais difícil a identificação de funções que realmente necessitam desses argumentos.

Em linguagens imperativas, é comum a definição de variáveis ou estados que podem ser acessados em diferentes escopos de função. A natureza de imutabilidade das variáveis impede a utilização de uma estratégia similar.

A mônada `State` pode ser utilizada para auxiliar em casos similares. Definida como `State s a`, essa mônada carrega um estado do tipo `s`, e um resultado do tipo `a`. Com ela, é possível "esconder" o estado do restante das computações, utilizando funções explícitas para acessá-lo quando necessário. O uso dessa mônada é explícito e declarado no tipo de saída da função, como pode ser visualizado na Figura 4.4, onde o estado representa um log de operações aritméticas através de uma `String`.

Para simplificar a sintaxe monádica, utiliza-se da notação `do`. Essa notação propicia um código mais similar ao código imperativo, declarando uma sequência de ações a serem executadas. A grande diferença é que todas essas ações devem estar sob o contexto de uma mesma mônada.

O operador `<-` é um operador de atribuição, onde o operando a esquerda é um nome para a ação descrita à direita do operador. No caso da mônada `State`, as funções `get` e `put` são utilizadas para obter e modificar o estado da computação, respectivamente. Essa sintaxe deixa explícita a alteração de estados, realizada na linha 7 da Figura 4.4. Como todas as modificações de estado devem ser precedidas pela obtenção do mesmo via `get`, a identificação dessas no código é simplificada.

A palavra reservada `return` é utilizada para injetar um valor em uma mônada, garantindo que o resultado da computação será do tipo `State String Int`. Diferente do uso em linguagens imperativas, `return` não termina a execução do bloco. No exemplo, é possível substituir a linha 8 por `st2 <- return $ f x' + $`, anexando mais ações nas linhas posteriores.

Figura 4.4 – Exemplo de uso da mônada `State`.

```

1 f :: Int -> Char -> State String Int
2 f x '*' =
3   do
4     stateString <- get
5     let x' = x * x in
6       do
7         put $ stateString ++ " * "
8         st2 <- f x' +
9         return st2

```

Figura 4.5 – Definição dos Tipos Abstratos da linguagem.

```

1  data A = One
2      | Sum A B
3      | Prod A B
4      | Rec A
5      | TypeVar Char
6  type B = A
7
8  data T = Iso A B
9      | Comp A B T
10
11 data V = EmptyV
12     | Xval String
13     | InjL V
14     | InjR V
15     | PairV V V
16     | Evaluate E
17
18 data Iso =Lambda String Iso
19     | IsoVar String
20     | App Iso Iso
21     | Clauses [(V,E)]
22     | ponto fixo String Iso
23
24 type Delta = [(String,A)]
25 type Psi = [(String,T)]

```

```

1  data P = EmptyP
2      | Xprod String
3      | PairP P P
4
5  data E = Val V
6      | LetE P Iso P E
7      | Combination E E
8      | AlphaVal Alpha E
9
10 data Term = EmptyTerm
11     | XTerm String
12     | InjLt Term
13     | InjRt Term
14     | PairTerm Term Term
15     | Omega Iso Term
16     | Let P Term Term
17     | EXTENSION TERMS
18     | CombTerms Term Term
19     | AlphaTerm Alpha Term
20     | ValueT V
21
22 type Alpha = Complex CReal
23 type TypingState = (Delta ,Psi)
24 type OD = [V]
25 type ODext = [E]

```

Fonte: Elaborado pelo autor, 2018.

4.2 TIPOS ABSTRATOS DE DADOS

A Figura 4.5 apresenta os Tipos Abstratos definidos para a implementação do interpretador. Através desses tipos de dados forma-se a árvore de sintaxe abstrata dos programas a serem avaliados. Os tipos possíveis para termos da linguagem são representados através do tipo A, juntamente com a declaração de um sinônimo B, especificados de forma a possibilitar que as definições do código espelhem a especificação formal da linguagem. Utilizando as definições abstratas, o tipo $\mathbb{1} \times (\mathbb{1} + \mathbb{1})$ é traduzido em Prod One (Sum One One).

Além dos construtores que representam os tipos concretos da linguagem, essa definição apresenta um construtor TypeVar, possibilitando a inclusão de variáveis de tipos não previstas na especificação da linguagem. A inclusão desse construtor é feita de forma a auxiliar na implementação do *Typechecker*, mais especificamente, na checagem de tipos de abstrações λ . Como variáveis de tipos não possuem semântica formal na linguagem, seu uso é restrito as rotinas de avaliação, não sendo adotados como parte da sintaxe formal da linguagem. Assim, um parser sintático não deverá gerar tokens que constituam valores desse construtor. Tipos de isomorfismos são definidos em T. O construtor Iso codifica o tipo de funções $a \rightarrow b$, enquanto o construtor Comp define a composição de isomorfismos de alta ordem.

Na definição dos valores da linguagem, a construção de variáveis é permitida

através de qualquer `String`, com o valor `()` representado por `EmptyV`. Os construtores `InjL`, `InjR` e `PairV` implementam os construtores de valores de soma e produtos da linguagem. Para além dos termos irreduzíveis definidos na especificação formal, o tipo `V` aceita um construtor `Evaluate`. O controle quântico estabelecido na linguagem limita-se a um sistema fechado, possibilitando a manipulação de superposições de qubits, representadas pela combinação de valores `Combination` definida em `E`. Com essa restrição, não existe uma operação de medição que reduza uma combinação linear a um valor simples. Como a linguagem permite a especificação de isomorfismos que geram superposições, como o exemplo `Had`, faz-se necessária a implementação de uma maneira de reportar uma superposição como resultado final da avaliação desses isomorfismos. O construtor `Evaluate` é utilizado para encapsular as combinações em um valor sem modificar a estrutura dos tipos concretos.

Os valores estendidos da linguagem são codificados no tipo `E`. O construtor `Val` encapsula valores, enquanto `LetE` permite a especificação de expressões `let` dentro das cláusulas de um isomorfismo. O construtor `AlphaVal` codifica a representação de qubits na linguagem, sendo `Alpha` um número complexo. O tipo `Alpha` é definido como um sinônimo para o tipo `Complex CReal`, obtido através da inclusão dos módulos homônimos das bibliotecas Haskell. O módulo `Complex` é fornecido via pacote de instalação `Cabal`, provendo, além da representação de números complexos, um número de funções para sua manipulação. Já o Tipo `CReal` é proveniente do módulo `Numbers`² e fornece a implementação de números reais. A inclusão de um módulo externo de uma implementação numérica com precisão fixa é realizada devido a limitação das representações numéricas padrão da linguagem, sujeitas a erros de arredondamento e comportamento errático em comparações.

Na definição dos termos, o construtor `Omega` codifica a aplicação de um termo a um isomorfismo, enquanto os construtores `CombTerms` e `AlphaTerm` são os termos quânticos equivalentes aos valores estendidos apresentados anteriormente. A inclusão de um construtor `ValueT` é o único ponto que difere da definição formal, representando um termo completamente reduzido. Seu propósito é encapsular um valor em um termo para a aplicação em um isomorfismo. Mais especificamente, a semântica da linguagem define que um termo deve ser reduzido a um valor antes de ser aplicado a uma função. Encapsular esse valor com uma construção de termo permite o prosseguimento da avaliação sem necessidade de alterar o tipo de entrada da avaliação.

Isomorfismos são representados pelo tipo abstrato `Iso`. Abstrações λ e construtores de ponto fixo μ são representados através da inclusão de uma `String` nomeando a variável que identifica o construto, seguida de uma definição de isomorfismo. Cláusulas são implementadas através de uma lista de pares (V, E) , mapeando os valores

²O pacote de módulos `Numbers` é disponibilizado sob a licença BSD no link: <https://hackage.haskell.org/package/numbers>.

(V) que definem o lado esquerdo das cláusulas a expressões (E) que definem o lado direito. A ordem de inclusão dos pares equivale a ordem sintática da definição de cláusulas, sendo o conjunto de cláusulas $\{v_1 \leftrightarrow e_1 | v_2 \leftrightarrow e_2 | v_3 \leftrightarrow e_3\}$ traduzido na construção `Clauses [(V1,E1), (V2,E2), (V3,E3)]`.

Os tipos `Delta` e `Psi` definem mapeamentos entre tipos e variáveis, relacionando o nome de variáveis aos seus tipos. Os construtores `OD` e `ODext` equivalem respectivamente a listas de valores e valores estendidos, sendo utilizados na implementação das decomposições ortogonais. Para finalizar a definição dos dados abstratos, um par das listas `Delta` e `Psi` é utilizado para compor o estado das computações no typechecker (`TypingState`), unindo ambos os contextos de tipos sob um mesmo nome.

4.3 FUNÇÕES UTILITÁRIAS

O módulo `Utils.hs` implementa funções utilitárias com propósito geral, aplicáveis tanto no módulo da checagem de tipos quanto no módulo de avaliação. Para facilitar a construção de programas diretamente na sintaxe abstrata da linguagem, alguns identificadores para a representação de termos com uso constante são definidos, tais como `tt` e `ff`, o tipo `bool` e as amplitudes `alpha` e `beta`. Com o mesmo propósito a função `boolLists` é utilizada para simplificar a criação de listas de booleanos na especificação de casos de teste. O propósito dessa função é traduzir uma lista contendo valores Haskell `True` e `False` em uma lista formada pela representação de listas na sintaxe abstrata da linguagem base do interpretador. Também são definidas funções para a tradução de um número inteiro em conjuntos de n bits.

O tratamento de erros é realizado pelas funções `wrap` (Figura 4.6) e `catchMaybe`. Como entrada, `wrap` aceita valores restritos pela mônada `Either`. Adotando como padrão que o valor `Left` da mônada contém sempre o caso de falha, a primeira cláusula de implementação simplesmente encerra o programa e apresenta o erro na tela do usuário. Nos demais casos, o valor resultante da avaliação é extraído da mônada e fornecido como resultado. Com a utilização das variáveis de tipo, a função pode ser aplicada tanto a verificação de tipos de termos (A), quanto a verificação de tipos de isos (T). Similarmente, a função `catchMaybe` é utilizada para verificar computações com condição de falha esperada. Em ambos os casos, o tratamento de erros é realizado através da finalização do programa e exibição de uma mensagem na tela do usuário. Em versões futuras, a inclusão de rotinas de tratamento de erro pode ser realizada através da modificação dessas funções.

A função `matchTypes` (Figura 4.7) utiliza a capacidade de definição de cláusulas com guardas, onde a verificação de igualdade dos argumentos é realizada pelo próprio pattern-matching da linguagem. A utilização de variáveis de tipos permite a aplicação da

Figura 4.6 – Função para tratamento de erros.

```

1 wrap :: Show b => Either b a -> a
2 wrap (Left err) = error (show err)
3 wrap (Right val) = val

```

Fonte: Elaborada pelo autor,2018.

função na verificação de tipos de valores (V) ou de termos (Term). Para os casos em que o tipo fornecido corresponde ao termo encontrado, a função retorna o tipo "empacotado" na mônada `Either` pelo construtor `Right`. Em outros casos, retorna um erro através do construtor `Left`, permitindo o tratamento correto pela função `wrap`.

Esse módulo define duas funções utilizadas para auxiliar na manipulação das amplitudes associadas a uma combinação linear. A função `getLinearTerms` extrai as amplitudes de uma lista de combinações lineares, retornando-as na forma de uma lista de listas. Seu propósito é possibilitar a criação de uma matriz de amplitudes para a verificação da unitariedade na definição de cláusulas de isomorfismos. Seu comportamento é dado pela função `getLinearAlphas` (Figura 4.8), que extrai as amplitudes de uma única combinação linear.

Finalmente, a função `grabPatternTypesFromAnnotation` é especificada para obter o tipo das variáveis em padrões de acordo com a anotação de tipo dos isomorfismos, armazenando-os em um contexto `Delta`. Essa operação inicializa o contexto de tipos com as suposições sobre os tipos esperados para as variáveis, possibilitando a verificação correta das cláusulas.

4.4 IMPLEMENTAÇÃO DO TYPECHECKER

A verificação de tipos ocorre recursivamente, recebendo em cada passo um termo e o tipo assumido para este, retornando um contexto de variáveis atualizado e, em caso de verificação correta, o tipo verificado para o termo. Nos casos de falha na verificação (o tipo atribuído difere do tipo encontrado pela verificação), retorna um erro de tipo. Através do uso das mônadas `State` e `Either`, a função é representada pela assinatura: `Term -> A -> State TypingState (Either TypeErrors A)`.

Figura 4.7 – Teste de correspondência de tipos.

```

1 matchTypes :: Show a => A -> a -> B -> Either TypeErrors B
2 matchTypes supplied t found
3     | supplied == found = Right supplied
4     | otherwise = Left $ VarError "Variable type doesnt match supplied A " (show t
      ++ ":" ++ show found ++ "not " ++ show supplied)

```

Fonte: Elaborado pelo autor, 2018.

Figura 4.8 – Extração das amplitudes de uma combinação linear.

```

1 getLinearAlphas :: E -> [Alpha]
2 getLinearAlphas (Combination (AlphaVal a v1) v2) = a : getLinearAlphas v2
3 getLinearAlphas (Combination (Val v) v2) = (1 :+ 0) : getLinearAlphas v2 — 1*CVaI =
   CVaI
4 getLinearAlphas (Val v) = (1 :+ 0):[]
5 getLinearAlphas (AlphaVal a _) = a:[]
6 getLinearAlphas (LetE _ _ _ e) = getLinearAlphas e

```

Fonte: Elaborado pelo autor, 2018.

Na maior parte dos casos, a sintaxe funcional faz com que a implementação espelhe a definição formal das regras de tipos da linguagem. Alguns dos casos que trazem interesse na definição são retratados na Figura 4.10. É interessante ressaltar que, por conta da inclusão da mônada `Either` e da função `wrap`, não é necessário incluir verificações condicionais na definição das cláusulas de typechecking. Qualquer falha na verificação será tratada pelas funções adequadas, permitindo a continuação da avaliação em casos bem sucedidos.

Para a verificação de tipos de variáveis (`XTerm`), obtém-se o estado atual da computação através da função `get`. Como o estado é formado por dois contextos de tipos, o resultado de `get` deve ser ligado a um par. Porém, para a verificação de tipos de termos, o contexto de variáveis de isos é irrelevante. Assim sendo, nomeia-se apenas um dos componentes do par (`delta`), ignorando o segundo componente, garantindo que não existe possibilidade de alteração do contexto de isos por parte dessa função. Obtido o contexto de tipos, a função verifica se a variável `x` está no contexto, recupera o tipo armazenado e verifica a correspondência com o tipo fornecido na expressão (`a`).

A definição da checagem de tipos para aplicações de isos em termos (`Omega f t`) faz uso da inclusão de variáveis de tipos na sintaxe abstrata. Para checar o tipo da aplicação, é necessário primeiro verificar o tipo do isomorfismo ao qual o termo é aplicado. Porém, a verificação de tipos requer, como argumento de entrada, o tipo suposto para o isomorfismo. Dessa maneira, é necessário estabelecer uma forma de sinalizar ao `Typechecker` que o tipo do isomorfismo deve ser obtido utilizando-se das definições anteriores. Por exemplo, se `Omega` é uma variável de iso, a verificação pode obter seu tipo analisando o contexto de tipos de variáveis de iso. Obtendo um tipo para o isomorfismo, verifica-se se esse coincide com o tipo de retorno fornecido pela aplicação.

Figura 4.9 – Verificação de tipos para variáveis de valores.

```

1 mytermTypeCheck :: Term -> A -> State TypingState (Either TypeErrors A)
2 mytermTypeCheck (XTerm x) a = do
3     (delta, _) <- get
4     return $ Right (wrap $ matchTypes a (XTerm x) $ wrap $
   myxType x $ xInContext x delta)

```

Fonte: Elaborado pelo autor, 2018.

Figura 4.10 – Implementação da verificação de tipos em termos.

```

1 mytermTypeCheck (Omega f t) b = do
2     st1 <- isoTypeCheck f (Iso (TypeVar 'a') b)
3     let isoInputType = wrap $ checkIsoReturnType b $ wrap
        $ st1
4     in do
5         st2 <- mytermTypeCheck t isoInputType
6         if (wrap st2 == isoInputType) then return $
7             Right b
            else return $ Left $ OmegaError "Omega
                input of the wrong type" (Omega f t)

```

Fonte: Elaborado pelo autor, 2018.

Em caso afirmativo, o tipo de entrada é extraído e fornecido juntamente com o termo para o prosseguimento da verificação de tipos.

Os termos $\text{InjLt } t$ e $\text{InjRt } t$ podem assumir dois tipos distintos. O primeiro é atribuído a forma usual para a construção de somas $\text{Sum } A \ B$. Uma segunda possibilidade é a utilização desses termos para a construção de listas com tipo $\text{Rec } A$. A distinção dos dois casos é facilmente tratada via *pattern-matching*, como pode ser visto na Figura 4.11. em essência, se um tipo recursivo é fornecido para um termo $\text{InjLt } t$, então t deve obrigatoriamente ser EmptyT , sinalizando a lista vazia. Similarmente, o termo InjRt deve ser acompanhado de um par para que a atribuição do tipo recursivo seja válida, podendo verificar então se o primeiro termo é do tipo da lista e se o segundo termo é realmente uma lista. Em qualquer outro caso, a checagem de tipos incorre em uma falha.

A checagem de tipos para valores é simplesmente um subconjunto das regras de checagem de tipos de termos.

Isomorfismos são checados pela função `isoTypeCheck`. Variáveis de iso, diferentemente das variáveis de valores e termos possuem dois casos distintos de tipagem. O primeiro é o caso simples, onde a variável é adicionada ao contexto com o tipo fornecido para a função. O segundo caso é a verificação de uma variável de iso com tipo composto

Figura 4.11 – Diferenciação entre tipos de soma e o tipo recursivo de listas.

```

1 mytermTypeCheck (InjLt t) (Sum a b) =
2     do
3         return $ Right $ Sum (wrap $ matchTypes a t $ wrap st1) b
4 mytermTypeCheck (InjLt EmptyTerm) (Rec a) = return $ Right $ Rec a
5 mytermTypeCheck (InjRt t) (Sum a b) =
6     do
7         st1 <- mytermTypeCheck t b
8         return $ Right $ Sum a (wrap $ matchTypes b t $ wrap st1)
9 mytermTypeCheck (InjRt (PairTerm t1 t2)) (Rec a) =
10    do
11        a' <- mytermTypeCheck t1 a
12        recA <- mytermTypeCheck t2 (Rec a)
13        case wrap $ recA of
14            Rec r -> if (wrap a' == a) then return $ Right $ Rec a
15                    else return $ Left $ CustomError "Inserting wrong typed term in list"
16            otherwise -> return $ Left $ CustomError "Right element of cons not a list"

```

Fonte: Elaborado pelo autor, 2018.

por variáveis, mencionado anteriormente, necessário pois variáveis de iso não carregam anotações de tipo sintaticamente. Nesse caso, a operação é invertida. Ao invés de adicionar o tipo da variável no contexto, o procedimento recupera o tipo armazenado anteriormente.

A verificação do tipo de aplicações de isomorfismo, apresentada na Figura 4.12 faz uso dessa distinção. Como o tipo de uma aplicação é simplesmente o tipo do isomorfismo resultante, não há como identificar o tipo individual do primeiro isomorfismo, logo é necessário fornecer um tipo artificial na verificação desse. Sabendo disso, para que a aplicação seja válida, o primeiro isomorfismo deve possuir o tipo $\text{Comp } A \ B \ T$, sendo separado na função nos tipos $\text{Iso } A \ B$, identificado por `iso1Left`, e T identificado por `iso1Right`. A partir daí, basta verificar que o isomorfismo sendo aplicado é coerente com o tipo esperado pelo primeiro ($\text{Iso } A \ B$), e que o tipo resultante da aplicação é coerente com o fornecido inicialmente.

Um ponto importante da verificação de tipos em isomorfismos é a verificação das cláusulas que os compõem. A Figura 4.13 demonstra a rotina utilizada para tal propósito. O tratamento das cláusulas requer a verificação individual dos valores que compõem os padrões (`vList`) e das expressões (`eList`), ambas submetidas a rotina que garante que os valores formam uma decomposição ortogonal dos tipos do isomorfismo. Além disso, é necessário verificar que as expressões do lado direito das cláusulas formam uma matriz unitária coerente com a extensão para combinações lineares. No caso de isos compostas por combinações com uma única amplitude 1 e as demais 0 (isos no âmbito clássico), uma verificação simplificada é realizada pela função `oZ`. Nesse caso, não é necessário construir a matriz e verificar a unitariedade, bastando garantir a presença de um único valor com amplitude 1.

Para isos nas quais a matriz é formada por amplitudes variáveis, o conjunto de amplitudes de todas as cláusulas é inicialmente extraído das expressões. A partir daí o módulo para o tratamento de matrizes em Haskell é utilizado para criar as matrizes inversa e transposta e verificar a igualdade entre ambas. A distinção entre os casos é realizada de forma a evitar o custo de performance incorrido na criação e teste das matrizes em casos

Figura 4.12 – Verificação do tipo para a aplicação de isomorfismos.

```

1  isoTypeCheck (App iso1 iso2) t =
2    do
3      st1 <- isoTypeCheck iso1 $ Comp (TypeVar 'a') (TypeVar 'b') t
4      let iso1Type = breakIsoType $ wrap st1
5          iso1Left = fst iso1Type
6          iso1Right = snd iso1Type
7          in do
8            st2 <- isoTypeCheck iso2 iso1Left
9            if (wrap st2) == iso1Left && iso1Right == t then return $ Right t
10           else return $ Left $ AppError "Cannot app isos" iso1 iso2

```

Fonte: Elaborado pelo autor, 2018.

de definições simplificadas e pode ser visualizada na Figura 4.14.

4.5 AVALIAÇÃO

A execução de um programa em uma linguagem funcional é realizada a partir da aplicação de um argumento a uma função. No caso da sintaxe abstrata desse trabalho, uma aplicação ($f(t)$) de um termo t a um isomorfismo f , é representada pela construção $\Omega f t$. Utilizando aplicações como o argumento inicial de avaliação, o interpretador foi construído com chamadas recursivas de funções de avaliação, obtendo como resultado a redução completa do termo. A Figura 4.15 mostra um grafo representando o fluxo básico do interpretador. Na Figura, são apresentados os nomes das funções utilizadas para a avaliação, acompanhados de setas dirigidas identificadas pela forma generalizada dos termos avaliados por cada função.

A função `applicativeContext` é a função central do interpretador. Seu propósito é determinar a ordem de redução dos termos da linguagem, efetuando as chamadas para a função apropriada. O fragmento de código da Figura 4.16 ilustra o comportamento da função. Para termos em sua forma normal, variáveis e o termo `()`, a função simplesmente os traduz nos valores equivalentes. Para os demais termos, a função aplica-se recursivamente aos componentes t dos construtores, reduzido-os a valores antes de efetivar a tradução. Aplicações de termos a funções e construtores `let` são reduzidos através da função `reductionRules`, sendo os valores obtidos através da redução recursiva encapsulados pelo construtor `ValueT`.

A função `reductionRules` codifica as regras de inferência que definem a semântica operacional da linguagem. É a partir dessa função que o fluxo de código desdobra em chamadas distintas. Em essência, essa função identifica a composição do termo a ser reduzido, fornecendo o mesmo para funções especializadas. Como um exemplo,

Figura 4.13 – Verificação de tipos das cláusulas de isomorfismos.

```

1 isoTypeCheck (Clauses list) (Iso a b) =
2   do
3     (delta ,_) <- get
4     let vList = map fst list
5         eList = map snd list
6         odA = wrap $ orthogonalDecomposition delta a [] vList
7         odB = wrap $ extOrthogonalDecomposition delta b [] eList
8         unitary = testUnit eList
9     in do
10      vTypes <- valueTypes vList a
11      eTypes <- estendedValueTypes eList b
12      if (checkODs odA odB a b) == (wrap eTypes) == b && (wrap vTypes) == a
13      then
14        if (unitary) then return $ Right $ Iso a b
15        else return $ Left $ IsoError "Not a unitary Matrix."
16      else return $ Left $ IsoError "Iso Clauseess dont match type."

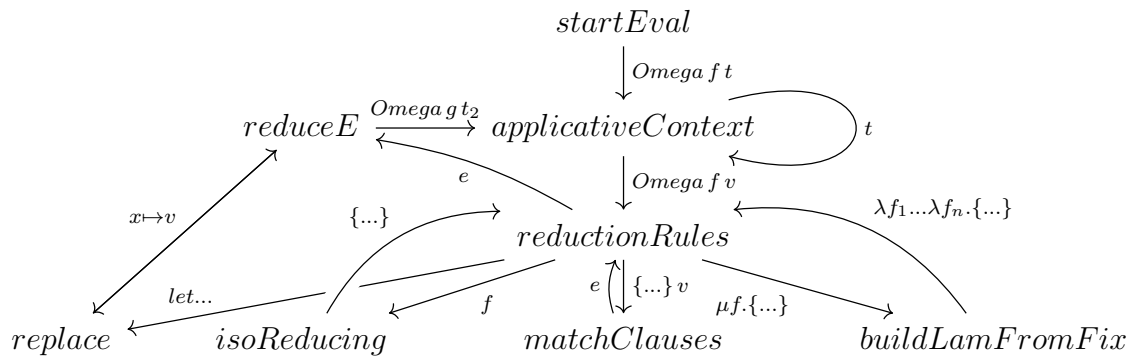
```

Figura 4.14 – Verificação da unitariedade das transformações lineares.

```

1 isUnitary :: [[Alpha]] -> Bool
2 isUnitary lists
3   | oZ lists = True
4   | otherwise =
5     let mat = debug(show lists ++ "\n")
6         fromLists lists —Create matrix from lists
7         conjugateTranspose = fmap conjugate $ Data.Matrix.transpose mat
8         inverseMat = wrap $ inverse mat —The inverse matrix
9         in if (conjugateTranspose) == inverseMat then True
10        else False

```

Figura 4.15 – Fluxo da avaliação de um termo Ω f t 

Fonte: Elaborado pelo autor, 2018..

Figura 4.16 – Implementação do contexto de avaliação.

```

1 applicativeContext :: Term -> V
2 applicativeContext (XTerm x) = Xval x
3 applicativeContext (InjRt t) = InjR $ applicativeContext t
4 applicativeContext (Omega iso t) = reductionRules $ Omega iso $ ValueT $
   applicativeContext t
5 applicativeContext (Let p t1 t2) = let v = ValueT $ applicativeContext t1
6   in reductionRules (Let p v t2)

```

a redução de termos `let` é realizada através da substituição de variáveis e casamento de padrões, enquanto a redução da funções de alta ordem é tratada por `isoReducing`. A redução de funções de alta ordem e e funções recursivas resulta em um conjunto de cláusulas onde os nomes de funções são substituídos pela sua definição, retornado para a função `reductionRules`, que inicia o processo de *pattern-matching* aplicando-as ao valor fornecido pela função anterior.

No que se refere a aplicação de valores a funções, existem dois casos específicos que precisam ser tratados. O primeiro, é a aplicação de um valor simples a um conjunto de cláusulas, resolvido pela aplicação de *pattern-matching* no valor de forma a escolher a cláusula adequada, seguida da redução do lado direito da cláusula para obter o valor resultante. Ambos os casos são mostrados na Figura 4.17.

O segundo caso envolve a aplicação de uma combinação linear de valores a um conjunto de cláusulas. Nesse caso, o trabalho delegado a função `matchLinearCombinations` consiste nos passos:

1. Aplicar as propriedades algébricas a combinação de valores.
2. Separar os valores de suas amplitudes, armazenando ambos.
3. Aplicar o *pattern-matching* a todos os valores da combinação.
4. Reduzir as expressões definidas nas cláusulas obtidas pelo passo anterior.
5. Construir a combinação linear resultante, agregando as amplitudes do resultado do passo anterior com as amplitudes originais dos valores.
6. Aplicar as propriedades algébricas a combinação resultante.

A implementação desses passos é visualizada na Figura 4.18. Estritamente falando, a aplicação das propriedades algébricas no primeiro momento não é de todo necessária, já que combinações lineares surgidas de aplicações serão reduzidas pelo último passo da função. No entanto, a aplicação inicial garante a uniformidade dos argumentos para o restante das computações, em especial quando da aplicação de uma função a uma combinação linear especificada diretamente pelo usuário.

Definindo `Sigma` como um mapeamento entre nomes de variáveis e valores, o *pattern-matching* é implementado através da função `matchClauses`, apresentada na

Figura 4.17 – Redução da aplicação de um valor a um isomorfismo.

```

1 reductionRules (Omega (Clauses isoDefs) (ValueT v)) =
2   let match = matchClauses isoDefs v 0
3       i = snd match
4       term = snd $ isoDefs !! i
5       in reduceE (fst match) term
6 reductionRules (Omega (Clauses isoDefs) (ValueT (Evalue e))) =
7   wrap $ matchLinearCombinations isoDefs e 1

```

Figura 4.18 – Processo para a avaliação de combinações lineares aplicadas a isomorfismos.

```

1 matchLinearCombinations :: [(V,E)] -> E -> Int -> Either [Char] V
2 matchLinearCombinations ve e i =
3     let e' = algebraicProperties e
4         vlist = grabValuesFromCombinations e'
5         (alphas,vs) = listsFromPairs vlist
6         sigmas = [matchClauses ve (v) 0 | v <- vs]
7         wi = [reduceE (fst s) (snd $ ve !! (snd s)) | s <- sigmas]
8         summs = sumWi alphas wi
9         result = Evaluate $! algebraicProperties summs
10        in Right result

```

Figura 4.19. Como a falha em uma aplicação sequencial do casamento de padrões é um comportamento esperado, o tipo de retorno `Maybe` com valor `Nothing` é utilizado para sinalizar uma não-correspondência. Em caso de sucesso, o resultado da função é um par contendo o mapeamento `Sigma` entre as variáveis e os valores que deverão assumir, acompanhado pelo índice da cláusula de padrões escolhida. Devido a natureza recursiva da aplicação do *pattern-matching*, a aplicação de um valor em um conjunto de padrões que não o aceitem resulta em uma substituição vazia `[]`, acompanhada por um índice superior ao tamanho da lista de cláusulas.

A redução de isomorfismos, dada pela função `IsoReducing` é responsável pela composição de isos. Em uma aplicação a um isomorfismo de alta ordem `App (Lambda f c1) iso2`, realiza a substituição das variáveis de `iso f` nas cláusulas `c1`, pela definição formal do isomorfismo `iso2`. Nesse ponto, é necessário discutir o problema da captura de variáveis. No caso das variáveis de isomorfismo, o problema é resolvido através da manutenção de uma lista de variáveis ligadas. Sempre que a substituição encontra um isomorfismo `Lambda f c1`, a variável `f` é adicionada a lista antes de prosseguir com a substituição na definição das cláusulas `c1` (Figura 4.20). A captura de variáveis de termos não é um problema dessa implementação pois seu escopo é sempre local ao isomorfismo. Pela sintaxe da linguagem reversível, a atribuição de valor a variáveis de termos só ocorre em padrões a esquerda das cláusulas, ou em expressões `let` a direita. No primeiro caso, o valor das variáveis é sempre obtido via *pattern-matching* na aplicação do isomorfismo a um valor, que é sempre iniciado com um contexto `Sigma` vazio. Para o segundo caso, o valor da variável será definido localmente pela aplicação que segue o sinal de igualdade.

A regra de avaliação para o operador de ponto fixo dita que uma função recursiva

Figura 4.19 – Implementação do *pattern-matching* em um conjunto de cláusulas e um valor.

```

1 matchClauses :: [(V,E)] -> V -> Int -> (Sigma, Int)
2 matchClauses [] v i = ([], i)
3 matchClauses (ve:list) v i = let sig = matching [] (fst ve) v
4                               in case sig of
5                                   Just sigma -> (sigma, i)
6                                   Nothing  -> matchClauses list v $ i+1

```

Figura 4.20 – Fragmento de código da substituição em isomorfismos.

```

1 substitution :: [String] -> String -> Iso -> Iso -> Maybe Iso
2 substitution boundVars f omega2 (IsoVar f') = if f' == f && not (f `elem` boundVars)
3                                           then Just omega2
4                                           else Nothing
5 substitution boundVars f omega2 (Lambda g iso) =
6   Just $ Lambda g $ testSubs iso $ substitution (g:boundVars) f omega2 iso
7 substitution boundVars f omega2 (ponto fixo g iso) =
8   Just $ ponto fixo g $ testSubs iso $ substitution (g:boundVars) f omega2 iso

```

fix f , pode ser reescrita em uma função não recursiva $\text{lam } f'$, aplicada em uma cópia f_0 de f , de forma a substituir a ocorrência da recursão nas cláusulas. Esse processo é repetido, resultando em uma corrente de lambdas e aplicações $\text{App } (\text{lam } f.\text{lam } f_0.\dots.\text{lam } f_n. \{\dots\}) (f, f_0, \dots, f_n)$.

Um dos desafios da implementação de operadores de ponto fixo é encontrar uma função f_n que represente o ponto de parada da recursão. Considerando o escopo de recursividade apresentado na linguagem, a obtenção de f_n pode ser simplificada. A linguagem apresenta recursão definida para listas, com uma regra bem definida para terminação: o ponto de parada da recursão é determinado pela aplicação de uma lista vazia a função. Os argumentos que podem ser aplicados a uma função recursiva são do tipo: lista vazia (InjL EmptyV) ou construtor de listas (InjR (Pair h t)). Assim, para determinar se a função é o término da recursão, basta analisar o argumento de entrada. Se o argumento é uma lista vazia, a função atual é um ponto de parada. Se o argumento é um construtor de listas, uma nova variável e iso é criada e adicionada a uma lista f . A partir daí, ignora-se o primeiro valor do construtor de listas (h) e o processo é repetido considerando a aplicação de t na cópia criada. O resultado dessa operação é uma lista contendo n variáveis de iso, representando a quantidade de iterações necessárias para obter-se a terminação. Com isso em mãos, é possível construir um isomorfismo de alta ordem representado por uma sequência de λ s, aplicada a n cópias da definição das cláusulas (Figura 4.21).

Figura 4.21 – Função para expandir a recursão em um isomorfismo finito.

```

1 buildLamFromFix :: String -> V -> Iso -> Iso
2 buildLamFromFix f v fix = let names = findFixedPoint f v fix
3                             lambdaChain = lambdaBuilding names fix
4                             appChain = appBuilding fix lambdaChain
5                             in appChain
6
7 findFixedPoint :: String -> V -> Iso -> [String]
8 findFixedPoint f (InjL EmptyV) fix = [f]
9 findFixedPoint f (PairV (InjL v) _) fix = findFixedPoint f (InjL v) fix
10 findFixedPoint f (PairV (InjR v) _) fix = findFixedPoint f (InjR v) fix
11 findFixedPoint f (PairV _ v2) fix = findFixedPoint f v2 fix
12 findFixedPoint f (InjR (PairV h t)) fix
13   | Evaluate (Val v') <- h = findFixedPoint f (InjR (PairV v' t)) fix
14   | Evaluate (Val v') <- t = findFixedPoint f (InjR (PairV h v')) fix
15   | otherwise = f : findFixedPoint f t fix

```

No que consta a reversibilidade, a reversão de isomorfismos é inicialmente simples (Figura 4.22). Como especificado pelo formalismo da linguagem, a inversão de cláusulas consiste em substituir o valor à esquerda das definições pelo valor que representa o resultado da avaliação da expressão a direita. Como a sintaxe define que o lado direito da definição deve ser um valor estendido, esse valor será a própria expressão em casos sem o uso de `let`, ou será o valor apresentado ao final de uma cadeia de `let`. No segundo caso, além da substituição dos valores, é necessário alterar adequadamente as variáveis nas subexpressões `let`. A inversão através dessa técnica é válida para qualquer iso onde os valores resultantes da expressão a direita não dependam de combinações lineares. Quando considerados isomorfismos definidos a partir de combinações lineares porém, a inversão simplificada de isomorfismos não gera construções válidas, como pode ser evidenciado pela equação 4.1.

$$\begin{array}{l} Had :: (\mathbb{B} \leftrightarrow \mathbb{B}) \\ \left\{ \begin{array}{l} tt \leftrightarrow \frac{1}{\sqrt{2}}tt + \frac{1}{\sqrt{2}}ff \\ ff \leftrightarrow \frac{1}{\sqrt{2}}tt - \frac{1}{\sqrt{2}}ff \end{array} \right\}^{-1} \end{array} \neq \begin{array}{l} Had^{-1} :: (\mathbb{B} \leftrightarrow \mathbb{B}) \\ \left\{ \begin{array}{l} \frac{1}{\sqrt{2}}tt + \frac{1}{\sqrt{2}}ff \leftrightarrow tt \\ \frac{1}{\sqrt{2}}tt - \frac{1}{\sqrt{2}}ff \leftrightarrow ff \end{array} \right\} \end{array} \quad (4.1)$$

O lado esquerdo de uma definição deve, obrigatoriamente, consistir de valores puros, sem combinações lineares. Essa relação é ainda mais evidente pelo fato que a implementação do *pattern-matching* não trata casos de combinações lineares. Nesses casos, a inversão das cláusulas requer um passo mais sutil: as combinações lineares devem ser invertidas antes da substituição de valores.

Como o *typechecker* garante que as combinações lineares no corpo de isomorfismos formam uma matriz unitária, é possível afirmar que existe uma matriz inversa para qualquer definição de isomorfismos consistente com a verificação de tipos. Com essa afirmação, é possível inverter isomorfismos através da operação ilustrada da equação 4.2, respeitando a igualdade

$$\begin{pmatrix} a'_{11} + a'_{21} + a'_{31} \\ a'_{12} + a'_{22} + a'_{32} \\ a'_{13} + a'_{23} + a'_{33} \end{pmatrix} = \begin{pmatrix} a_{11} + a_{21} + a_{31} \\ a_{12} + a_{22} + a_{32} \\ a_{13} + a_{23} + a_{33} \end{pmatrix}^{-1}$$

Figura 4.22 – Inversão simplificada das cláusulas de um isomorfismo.

```

1 invertClauses :: [(V,E)] -> [(V,E)]
2 invertClauses [] = []
3 invertClauses (ve:listVE) = let e' = invertextendedValue $ snd ve
4                               v' = bottomValue $ snd ve
5                               in buildInverted ve e' v' : invertClauses listVE

```

$$\begin{aligned} & \left\{ \begin{array}{l} \mathbf{v}_1 \leftrightarrow a_{11}\mathbf{v}'_1 + a_{21}v'_2 + a_{31}v'_3 \\ \mathbf{v}_2 \leftrightarrow a_{12}v'_1 + a_{22}\mathbf{v}'_2 + a_{32}v'_3 \\ \mathbf{v}_3 \leftrightarrow a_{13}v'_1 + a_{23}v'_2 + a_{33}\mathbf{v}'_3 \end{array} \right\}^{-1} \\ & = \left\{ \begin{array}{l} \mathbf{v}'_1 \leftrightarrow a'_{11}\mathbf{v}_1 + a'_{21}v'_2 + a'_{31}v'_3 \\ \mathbf{v}'_2 \leftrightarrow a'_{12}v'_1 + a'_{22}\mathbf{v}_2 + a'_{32}v'_3 \\ \mathbf{v}'_3 \leftrightarrow a'_{13}v'_1 + a'_{23}v'_2 + a'_{33}\mathbf{v}_3 \end{array} \right\} \end{aligned} \quad (4.2)$$

Antes de explorar a implementação desse caso, é importante expor algumas convenções sobre a ordem na qual os valores aparecem nas combinações lineares. Considerando os valores ao lado esquerdo das cláusulas, o primeiro padrão (v_1 no topo da definição) é relacionado ao primeiro valor da combinação linear (v'_1), o segundo padrão relaciona-se com o segundo valor na combinação linear e assim por diante. Nos casos em que o isomorfismo representa uma operação quântica, a ordenação deve ser respeitada para que a definição do isomorfismo defina adequadamente a matriz unitária da operação. Em outros casos, a implementação de um *parser* sintático pode garantir a ordenação dos valores, transformando um isomorfismo simples em uma matriz linear:

$$\left\{ \begin{array}{l} v_1 \leftrightarrow v'_1 \\ v_2 \leftrightarrow v'_2 \\ v_3 \leftrightarrow v'_3 \end{array} \right\} \xrightarrow{\text{Parser}} \left\{ \begin{array}{l} v_1 \leftrightarrow 1v'_1 + 0v'_2 + 0v'_3 \\ v_2 \leftrightarrow 0v'_1 + 1v'_2 + 0v'_3 \\ v_3 \leftrightarrow 0v'_1 + 0v'_2 + 1v'_3 \end{array} \right\}.$$

Com a presunção de que a ordem dos valores será respeitada em qualquer definição, a inversão das cláusulas é dada pelas funções da Figura 4.23. Na inversão de cláusulas, valores e amplitudes são separados em duas listas, utilizando a segunda para construir e inverter algebricamente uma matriz. Após a inversão, a matriz de amplitudes é transformada novamente em um conjunto de listas, utilizadas para reconstruir as combinações lineares utilizando os valores originais. Feito isso, substituem-se os valores do lado esquerdo da definição pelo valor na posição adequada da combinação.

4.6 EXEMPLOS DE CONTROLE QUÂNTICO

Esta seção apresenta exemplos e algoritmos especificados e executados no interpretador descrito nesse trabalho. Os exemplos foram escolhidos por demonstrar a codificação de propriedades quânticas, incluindo o controle quântico em estruturas de condicional e de repetição. Durante a exposição no texto, os exemplos serão demonstrados através da sintaxe abstrata da linguagem. A expressão $x \mapsto v$ é utilizada para indicar o mapeamento de uma variável x ao valor v . Em geral, as equações utilizam a notação

Figura 4.23 – Implementação da inversão de cláusulas considerando transformações lineares.

```

1  invertCl :: [(V,E)] -> [(V,E)]
2  invertCL list = let (values,linearEs) = listsFromPairs list
3                    matrix = (fromLists . getLinearTerms) linearEs
4                    inverseLinearAlphas = toLists $ wrap $ inverse matrix
5                    inverseLinears = rebuildEs linearEs inverseLinearAlphas
6                    newClauses = invertLinearClauses values inverseLinears 0
7                    in newClauses
8
9  invertLinearClauses :: [V] -> [E] -> Int -> [(V,E)]
10 invertLinearClauses v (e:elist) i = let (v',e') = swapCombinationVals v e i
11                                       invE = invertextendedValue e'
12                                       in (v',invE): invertLinearClauses v elist (i+1)

```

Fonte: Elaborado pelo autor, 2018.

$x \mapsto v$ para simplificar o termo: $\text{let } x = v \text{ in } e$. A seta \rightarrow indica a transformação de valores através de passos de avaliação, nomeados acima da mesma.

Na ausência de um *parser* para a sintaxe concreta da linguagem, os exemplos foram codificados diretamente na sintaxe abstrata. O arquivo `IsoDefinitions.hs` contém funções Haskell que definem isomorfismos e seus tipos. Algumas das definições são apresentadas no Apêndice A. Funções de teste foram especificadas para possibilitar a verificação individual da checagem de tipos e avaliação das funções utilizadas como exemplo.

4.6.1 Exemplo 1 - Algoritmo de Deutsch

O algoritmo de Deutsch possui como entrada um oráculo reversível implementando uma função f e 2 qubits iniciados no estado $|01\rangle$. O resultado da execução do algoritmo através de uma medição do primeiro qubit é 0 para funções constantes e 1 para funções balanceadas. O passo a passo da execução do algoritmo foi apresentado no capítulo 2.

Utilizando a sintaxe da linguagem, o algoritmo de Deutsch pode ser descrito através do isomorfismo da Figura 4.24. Na figura, cada passo do algoritmo de Deutsch é construído como uma aplicação de isomorfismos utilizando o construto `let`. A utilização de funções de alta ordem permite que o oráculo seja compreendido como uma caixa-preta, externa a definição do algoritmo. As definições dos isomorfismos *Oracle* e *HadId* são apresentadas nas Figuras 4.25 e 4.26 e discutidas posteriormente.

O teste prático do algoritmo depende da definição prévia de um isomorfismo que descreva o comportamento esperado para a caixa-preta descrita pelo oráculo de uma função f . A implementação desse tipo de oráculo é generalizada pela construção da Figura 4.25. Considerando que a linguagem não permite a expressão de funções irreversíveis, não é possível prover um oráculo genérico na forma de uma função de alto nível que receba a função original f como argumento, sendo necessária a especificação

Figura 4.24 – Iso codificando o algoritmo de Deutsch.

$$\text{Deutsch}:: (\mathbf{B} \leftrightarrow \mathbf{B}) \rightarrow (\mathbf{B}^* \mathbf{B} \leftrightarrow \mathbf{B}^* \mathbf{B}) \rightarrow (\mathbf{B}^* \mathbf{B} \leftrightarrow \mathbf{B}^* \mathbf{B}) \rightarrow (\mathbf{B}^* \mathbf{B} \leftrightarrow \mathbf{B}^* \mathbf{B}) =$$

$$\left\{ \begin{array}{l} \lambda \text{Had} . \lambda \text{Oracle} . \lambda \text{HadId} . \\ \langle x, y \rangle \leftrightarrow \text{let } h1 = \text{Had } x \text{ in} \\ \quad \text{let } h2 = \text{Had } y \text{ in} \\ \quad \text{let } q = \text{Oracle } \langle h1, h2 \rangle \text{ in} \\ \quad \text{let } out = \text{HadId } q \text{ in } out \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

Figura 4.25 – Iso generalizando um Oráculo reversível.

$$\text{Oracle} = \left\{ \begin{array}{l} \langle x, tt \rangle \leftrightarrow 1 \cdot \langle x, f x \rangle + 0 \\ \langle x, ff \rangle \leftrightarrow 0 + 1 \cdot \langle x, \text{not } (f x) \rangle \end{array} \right\}$$

de um oráculo específico para a função (Figura 4.27).

A Figura 4.26 apresenta um isomorfismo que atua em um par de booleanos, aplicando a transformação de Hadamard ao primeiro, mantendo o segundo valor inalterado. Esse isomorfismo é equivalente ao produto tensor entre a transformação unitária de Hadamard e a matriz identidade, representada pela equação:

$$H \otimes Id = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & -\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Definidos os isomorfismos, o algoritmo pode ser executado através da composição dos isomorfismos $(((\text{Deutsch Had}) \text{Oracle}) \text{HadId})$, aplicada ao valor de entrada $\langle tt, ff \rangle$. A avaliação é ilustrada para a aplicação do algoritmo com um oráculo implementando a função balanceada *not*. O primeiro passo de avaliação substitui as variáveis de iso pelas definições das cláusulas na composição $(((\text{Deutsch Had}) \text{Oracle}) \text{HadId})$ obtendo um único isomorfismo contendo aplicações de valores a isos em seu corpo. Através do *pattern-matching* no valor de entrada, têm-se o mapeamento de variáveis:

$$\sigma = \{x \mapsto tt; y \mapsto ff\}.$$

Figura 4.26 – Iso codificando uma operação em 2 qubits, aplicando a transformada de Hadamard no primeiro e a função identidade no segundo.

$$\text{HadId}:: (\mathbf{B} \otimes \mathbf{B}) \leftrightarrow (\mathbf{B} \otimes \mathbf{B}) =$$

$$\left\{ \begin{array}{l} \langle tt, y \rangle \leftrightarrow \text{let } q = \text{Id } y \text{ in } \frac{1}{\sqrt{(2)}} \langle tt, q \rangle + \frac{1}{\sqrt{(2)}} \langle ff, q \rangle \\ \langle ff, y \rangle \leftrightarrow \text{let } q = \text{Id } y \text{ in } \frac{1}{\sqrt{(2)}} \langle tt, q \rangle - \frac{1}{\sqrt{(2)}} \langle ff, q \rangle \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

Figura 4.27 – Iso generalizando um Oráculo reversível.

$$\text{OracleNot}:: (\mathbb{B} \otimes \mathbb{B}) \leftrightarrow (\mathbb{B} \otimes \mathbb{B}) = \left\{ \begin{array}{l} \langle tt, tt \rangle \leftrightarrow \langle tt, ff \rangle \\ \langle tt, ff \rangle \leftrightarrow \langle tt, tt \rangle \\ \langle ff, tt \rangle \leftrightarrow \langle ff, tt \rangle \\ \langle ff, ff \rangle \leftrightarrow \langle ff, ff \rangle \end{array} \right\}$$

Com as definições de isomorfismos já apresentadas, a avaliação da primeira aplicação de *Had* segue substituindo x pelo valor apresentado em σ :

$$h1 \mapsto \text{Had} \, tt = \frac{1}{\sqrt{2}} tt + \frac{1}{\sqrt{2}} ff \text{ in } e.$$

Considerando o fato de que o *pattern-matching* só é possível entre valores simples, o mapeamento direto de $h1$ para a combinação linear não pode prosseguir. Ao invés disso, aplica-se a propriedade distributiva das expressões let, fazendo com que a avaliação prossiga em dois ramos:

$$\frac{1}{\sqrt{2}} h1 \mapsto tt \text{ in } e_1 + \frac{1}{\sqrt{2}} h1 \mapsto ff \text{ in } e_2.$$

A avaliação prossegue para cada e_i , obtendo a aplicação da transformação de Hadamard na variável y :

$$\begin{aligned} & \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} h2 \mapsto tt \text{ in } e_1 - \frac{1}{\sqrt{2}} h2 \mapsto ff \text{ in } e_2 \right) + \\ & \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} h2 \mapsto tt \text{ in } e_3 - \frac{1}{\sqrt{2}} h2 \mapsto ff \text{ in } e_4 \right), \end{aligned}$$

gerando a avaliação da combinação linear:

$$\begin{aligned} & \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} q \mapsto \text{Oracle} \langle tt, tt \rangle \text{ in } e_1 - \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} q \mapsto \text{Oracle} \langle tt, ff \rangle \text{ in } e_2 + \\ & \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} q \mapsto \text{Oracle} \langle ff, tt \rangle \text{ in } e_3 - \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} q \mapsto \text{Oracle} \langle ff, ff \rangle \text{ in } e_4 \\ & \xrightarrow{\text{Evals}} \frac{1}{2} q \mapsto \langle tt, ff \rangle \text{ in } e_1 - \frac{1}{2} q \mapsto \langle tt, tt \rangle \text{ in } e_2 + \\ & \frac{1}{2} q \mapsto \langle ff, tt \rangle \text{ in } e_3 - \frac{1}{2} q \mapsto \langle ff, ff \rangle \text{ in } e_4. \end{aligned} \tag{4.3}$$

Os passos de avaliação destacados demonstram a forma pela qual o paralelismo quântico é codificado na linguagem. Devido as propriedades algébricas dos valores, a aplicação do oráculo ocorre em todos os valores possíveis para o par $\langle x, y \rangle$. A implementação atual do interpretador realiza todas as avaliações sequencialmente porém, uma extensão de código para realizá-las em paralela é possível. Continuando com a

aplicação de *HadId* em todos os ramos de avaliação:

$$\begin{aligned}
\frac{1}{2}out &\mapsto \frac{1}{\sqrt{2}}\langle tt, ff \rangle + \frac{1}{\sqrt{2}}\langle ff, ff \rangle - \frac{1}{2}out \mapsto \frac{1}{\sqrt{2}}\langle tt, tt \rangle + \frac{1}{\sqrt{2}}\langle ff, tt \rangle + \\
\frac{1}{2}out &\mapsto \frac{1}{\sqrt{2}}\langle tt, tt \rangle - \frac{1}{\sqrt{2}}\langle ff, tt \rangle - \frac{1}{2}out \mapsto \frac{1}{\sqrt{2}}\langle tt, ff \rangle - \frac{1}{\sqrt{2}}\langle ff, ff \rangle \\
&\xrightarrow{Distr} \frac{1}{2} \cdot \left(\frac{1}{\sqrt{2}}out \mapsto \langle tt, ff \rangle + \frac{1}{\sqrt{2}}out \mapsto \langle ff, ff \rangle \right) - \\
&\quad \frac{1}{2} \cdot \left(\frac{1}{\sqrt{2}}out \mapsto \langle tt, tt \rangle + \frac{1}{\sqrt{2}}out \mapsto \langle ff, tt \rangle \right) + \\
&\quad \frac{1}{2} \cdot \left(\frac{1}{\sqrt{2}}out \mapsto \langle tt, tt \rangle - \frac{1}{\sqrt{2}}out \mapsto \langle ff, tt \rangle \right) - \\
&\quad \frac{1}{2} \cdot \left(\frac{1}{\sqrt{2}}out \mapsto \langle tt, ff \rangle - \frac{1}{\sqrt{2}}out \mapsto \langle ff, ff \rangle \right).
\end{aligned}$$

Logo, o valor resultante da aplicação do isomorfismo é dado pela aplicação das propriedades algébricas de combinações lineares:

$$\begin{aligned}
&\frac{1}{2\sqrt{2}}\langle tt, ff \rangle + \frac{1}{2\sqrt{2}}\langle ff, ff \rangle - \frac{1}{2\sqrt{2}}\langle tt, tt \rangle - \frac{1}{2\sqrt{2}}\langle ff, tt \rangle + \\
&\frac{1}{2\sqrt{2}}\langle tt, tt \rangle - \frac{1}{2\sqrt{2}}\langle ff, tt \rangle - \frac{1}{2\sqrt{2}}\langle tt, ff \rangle + \frac{1}{2\sqrt{2}}\langle ff, ff \rangle \\
&\xrightarrow{Algeb} -\frac{1}{2\sqrt{2}}\langle ff, tt \rangle + \left(-\frac{1}{2\sqrt{2}}\langle ff, tt \rangle \right) + \frac{1}{2\sqrt{2}}\langle ff, ff \rangle + \frac{1}{2\sqrt{2}}\langle ff, ff \rangle \\
&\xrightarrow{Algeb} -\frac{1}{\sqrt{2}}\langle ff, tt \rangle + \frac{1}{\sqrt{2}}\langle ff, ff \rangle.
\end{aligned}$$

Para confirmar a equivalência da avaliação com o resultado do algoritmo de Deutsch aplicando uma função balanceada, considere-se que o valor resultante da avaliação é análogo ao par $\langle ff, -\frac{1}{\sqrt{2}}tt + \frac{1}{\sqrt{2}}ff \rangle$ via distributividade, porém a sintaxe da linguagem só permite a presença de valores simples em tuplas. Como a linguagem reversível não apresenta construções semânticas para a operação de medição, a execução do algoritmo de Deutsch termina nesse passo, sendo trivial perceber que uma medição no primeiro qubit resultaria no resultado ff .

A avaliação inversa do algoritmo de Deutsch é dada pela inversão do isomorfismo que o codifica. O interpretador é capaz de realizar a inversão de todos os isomorfismos da composição através da invocação da rotina de inversão `invertIsos` (`((Deutsch Had) Oracle) HadID`), resultando na construção da Figura 4.28. Aplicar o valor resultante da avaliação usual do algoritmo ao isomorfismo inverso resulta no valor original $\langle tt, ff \rangle$.

Uma generalização do algoritmo de Deutsch para strings de n -qubits, conhecida por

Figura 4.28 – Inversão do isomorfismo representando o algoritmo de Deutsch.

$$\text{Deutsch}^{-1}:: (\mathbb{B} \leftrightarrow \mathbb{B}) \rightarrow (\mathbb{B}^* \mathbb{B} \leftrightarrow \mathbb{B}^* \mathbb{B}) \rightarrow (\mathbb{B}^* \mathbb{B} \leftrightarrow \mathbb{B}^* \mathbb{B}) \rightarrow (\mathbb{B}^* \mathbb{B} \leftrightarrow \mathbb{B}^* \mathbb{B}) =$$

$$\left\{ \begin{array}{l} \lambda \text{Had}^{-1}. \lambda \text{Oracle}^{-1}. \lambda \text{Had} \text{Id}^{-1}. \\ \text{out} \leftrightarrow \text{let } q = \text{Had} \text{Id}^{-1} \text{ out in} \\ \quad \text{let } \langle h1, h2 \rangle = \text{Oracle}^{-1} q \text{ in} \\ \quad \text{let } y = \text{Had}^{-1} h2 \text{ in} \\ \quad \text{let } x = \text{Had}^{-1} h1 \text{ in } \langle x, y \rangle \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

algoritmo de Deutsch-Jozsa, é possibilitada através do uso de listas de valores booleanos. Para tal, uma iso recursiva aplicando Had em $n - 1$ qubits e Id em 1 qubit é definida na Figura 4.29. Com essa definição, o algoritmo de Deutsch-Jozsa é codificado pela sintaxe da Figura 4.30. Ambas as aplicações da transformada de Hadamard são realizadas por funções recursivas, com comportamento bem definido para quaisquer listas com tamanho finito. Dessa maneira, desde que o programa seja suprido de uma iso bem formada implementando o comportamento do oráculo ($\text{Oracle}N$), a execução do algoritmo prossegue como o esperado.

4.6.2 Exemplo 2 - Quantum-walks recursivos

Uma caminhada unidimensional (*one-dimensional walk*) é uma função W que atua sobre um par indicando uma direção e um índice ($\langle d, i \rangle$), de forma que:

$$W(\langle d, i \rangle) = \begin{cases} W(\langle R, i \rangle) = \langle R, i + 1 \rangle. \\ W(\langle L, i \rangle) = \langle L, i - 1 \rangle. \end{cases}$$

Uma versão quântica de W é obtida permitindo a superposição dos estados da direção ($\alpha R + \beta L$).

A Figura 4.34 apresenta um isomorfismo codificando W na linguagem base desse trabalho. Na definição, utiliza-se um valor booleano para representar as direções ($tt = R$ e $ff = L$), em conjunto com uma representação binária de valores inteiros, implementada

Figura 4.29 – Iso recursiva atuando em uma lista de qubits.

$$\text{HadNId}:: (\mathbb{B} \leftrightarrow \mathbb{B}) \rightarrow ([\mathbb{B}] \leftrightarrow [\mathbb{B}]) =$$

$$\lambda \text{Id}. \mu f$$

$$\left\{ \begin{array}{l} [] \leftrightarrow [] \\ q : [] \leftrightarrow \text{let } z = \text{Id } q \text{ in} \\ \quad \text{let } w = f [] \text{ in } 0 + (z : w) + 0 + 0 \\ tt : t \leftrightarrow \text{let } w = f t \text{ in } 0 + 0 + \frac{1}{\sqrt{2}}(tt : w) + \frac{1}{\sqrt{2}}(ff : w) \\ ff : t \leftrightarrow \text{let } w = f t \text{ in } 0 + 0 + \frac{1}{\sqrt{2}}(tt : w) - \frac{1}{\sqrt{2}}(ff : w) \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

Figura 4.30 – Isomorfismo codificando o algoritmo de Deutsch-Jozsa.

$$\text{Deutsch-Jozsa}:: ([B] \leftrightarrow [B]) \rightarrow (B \leftrightarrow B) \rightarrow ([B] \leftrightarrow [B]) \rightarrow ([B] \leftrightarrow [B]) \rightarrow ([B] \leftrightarrow [B]) =$$

$$\lambda \text{Map} . \lambda \text{Had} . \lambda \text{OracleN} . \lambda \text{HadNId} .$$

$$\left\{ \begin{array}{l} \text{input} \leftrightarrow \text{let } h1 = (\text{Map } \text{Had}) \text{ input in} \\ \text{let } q = \text{OracleN } h1 \text{ in} \\ \text{let } out = \text{HadNId } q \text{ in } out \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

como uma tupla de booleanos. Com o objetivo de simplificar a especificação dos exemplos, a implementação considera uma representação binária via magnitude de números inteiros reais de 5 bits. Para evitar preocupações com a presença de duas strings de bits para o valor 0, valores positivos são interpretados como a forma binária comum de 4 bits, acompanhados de 0 no dígito mais significativo. Um número negativo i é entendido como a representação binária usual em 4 bits de $i - 1$, acompanhado de 1 no dígito mais significativo. Isso significa que o valor 1 é representado pelo conjunto de bits 00001 enquanto o valor -1 é representado pela sequência 10000. A implementação permite a representação de valores inteiros entre -17 e 16 .

Com essa representação em mente, os isomorfismos *prev* e *next* codificam as operações (-1) e $(+1)$ respectivamente, utilizando um mecanismo de *overflow* para garantir a reversibilidade das operações. Assim, a operação *prev* (-17) resulta em 16 , enquanto *next* (16) resulta em -17 .

Utilizando os isomorfismos definidos, é possível aplicar um passo da caminhada quântica em uma superposição balanceada das direções através do termo $\langle [tt, 0] \rangle$ e do isomorfismo da Figura 4.31. Considerando a construção do isomorfismo através da aplicação dos isos *Had* e *W*, a avaliação é dada por:

$$\text{singleStepWalk } \langle [tt, 0] \rangle \xrightarrow{\text{Evals}} \frac{1}{\sqrt{2}} \langle tt, -1 \rangle + \frac{1}{\sqrt{2}} \langle ff, 1 \rangle.$$

Versões recursivas do algoritmo de quantum-walk são descritas por Ying (2016) para apresentar o conceito de recursão quântica. A forma mais simples de recursão consiste em aplicar a função *W* repetidamente ao estado resultante. Essa forma é

Figura 4.31 – Isomorfismo codificando um passo único de um *quantum-walk*.

$$\text{SingleStepWalk}:: (B \leftrightarrow B) \rightarrow (B^* \text{Int} \leftrightarrow B^* \text{Int}) \rightarrow ([B]^* \text{Int} \leftrightarrow [B]^* \text{Int}) =$$

$$\lambda \text{had} . \lambda \text{W} .$$

$$\left\{ \begin{array}{l} \langle d, p \rangle \leftrightarrow \text{let } y = \text{had } d \text{ in} \\ \text{let } \langle d_1, p_1 \rangle = \text{W } \langle y, p \rangle \text{ in} \\ \langle d_1, p_1 \rangle \end{array} \right\}$$

Fonte: Elaborado pelo autor, 2018.

facilmente obtida através da avaliação de um termo na forma:

$$\begin{aligned} &\text{let } \langle x, y \rangle = \text{singleStepWalk}(tt, 0) \text{ in} \\ &\quad \text{let } \langle z, w \rangle = \text{singleStepWalk}(x, y) \text{ in} \\ &\quad \quad \text{let } \langle z1, w1 \rangle = \text{singleStepWalk}(z, w) \text{ in} \\ &\quad \quad \quad \text{let } \langle z2, w2 \rangle = \text{singleStepWalk}(z1, w1) \text{ in} \\ &\quad \quad \quad \quad \dots \end{aligned}$$

Uma versão recursiva denominada *unidirectionally recursive quantum-walk* é definida de forma que, aplicada a uma direção \leftarrow , a função executa um passo usual e é encerrada. Porém, ao receber uma direção \rightarrow a função é recursivamente replicada. Esse comportamento é descrito pelas equações:

$$\begin{cases} X(\langle R, i \rangle) = \langle R, i + 1 \rangle X(\langle d, i + 1 \rangle). \\ X(\langle L, i \rangle) = \langle L, i - 1 \rangle. \end{cases}$$

Na equação, d consiste em uma superposição balanceada das direções. No modelo proposto por Ying, a superposição balanceada d é descrita através de uma analogia com uma moeda justa (*fair coin*) e é assim referida durante o decorrer do texto. Nesse contexto, o autor explica que cada replicação recursiva X_n , incorre na inclusão de uma nova moeda d_n , externa ao sistema.

Uma segunda versão, referida por *bi-directionally recursive quantum-walk* (*quantum-walk* recursivo em duas direções) e apresenta o seguinte comportamento:

$$\begin{cases} X_2(\langle R, i \rangle) = \langle R, i + 1 \rangle X_2(\langle d, i + 1 \rangle). \\ X_2(\langle L, i \rangle) = \langle L, i - 1 \rangle X_2(\langle d, i - 1 \rangle). \end{cases}$$

Essencialmente, a função é agora replicada recursivamente após executar um passo da caminhada em qualquer direção.

Ao descrever a linguagem reversível utilizada para a base deste trabalho, Sabry, Valiron e Vizzotto (2017) afirmam que o modelo de recursão sobre listas pode ser visto como análogo a proposta das moedas. Dessa maneira, deve ser possível especificar isomorfismos na linguagem capazes de replicar o comportamento de *quantum-walks* recursivos. A Figura 4.35 apresenta a definição de um isomorfismo que equivale ao algoritmo bidirecionalmente recursivo.

Na Figura, a transformada de Hadamard (*had*) é utilizada para construir a superposição balanceada dos estados de direção armazenados em uma lista de valores booleanos. O isomorfismo é definido utilizando funções de alta ordem para incluir a transformação contendo o passo individual de caminhada W dada anteriormente. Embora o modelo apresentado por Ying sugere a aplicação da recursão em um número

desconhecido de moedas, a execução efetiva no interpretador aqui construído necessita de um número fixo de elementos na lista. Uma execução simples da caminhada recursiva é demonstrada através da avaliação do termo:

$$(X2\ had\ W)\langle(tt : tt : []), 0\rangle$$

A avaliação de funções recursivas via ponto fixo é realizada via um procedimento de *unfolding*, cujo primeiro passo é demonstrado na Figura 4.32. Como o número de aplicações recursivas é determinado pelo conteúdo da lista, a forma final do isomorfismo inclui um segundo passo de *unfolding*, substituindo *rec* pelo próprio isomorfismo no conjunto de cláusulas mais interno, substituindo também as variáveis *had* e *W* pelas respectivas definições desses isomorfismos.

Figura 4.32 – Demonstração de um passo de avaliação em uma recursão via ponto fixo.

$$\left(\begin{array}{l} \langle [], p \rangle \leftrightarrow \langle [], p \rangle + 0 \\ \langle h : t, p \rangle \leftrightarrow \\ \quad \text{let } y = \text{had } h \text{ in} \\ \quad \text{let } \langle h_1, p_1 \rangle = W \langle y, p \rangle \text{ in} \\ \quad \text{let } \langle t_1, p_2 \rangle = \left(\begin{array}{l} \langle [], p^1 \rangle \leftrightarrow \langle [], p^1 \rangle + 0 \\ \langle h^1 : t^1, p^1 \rangle \leftrightarrow \\ \quad \text{let } y^1 = \text{had } h_1 \text{ in} \\ \quad \text{let } \langle h_1^1, p_1^1 \rangle = W \langle y^1, p^1 \rangle \text{ in} \\ \quad \text{let } \langle t_1^1, p_2^1 \rangle = \text{rec } \langle t^1, p_1^1 \rangle \text{ in} \\ \quad 0 + \langle h_1^1 : t_1^1, p_2^1 \rangle \end{array} \right) \langle t, p_1 \rangle \text{ in} \\ 0 + \langle h_1 : t_1, p_2 \rangle \end{array} \right)$$

Fonte: Elaborado pelo autor, 2018.

A avaliação inicial da aplicação do isomorfismo recursivo no valor de entrada $\langle(tt : tt : []), 0\rangle$ gera o mapeamento de variáveis utilizado para a avaliação da segunda cláusula:

$$\sigma = \{h \mapsto tt; t \mapsto (tt : []); p \mapsto 0\}.$$

Com o mapeamento de variáveis em mãos, dá-se prosseguimento a avaliação do lado direito da cláusula. Expressões *let* são avaliadas sequencialmente, cada aplicação de isomorfismo atualizando o mapeamento de variáveis. Assim, a avaliação de *had h* seguida da aplicação da propriedade distributiva de construtos *let* gera:

$$y \mapsto \left(\frac{1}{\sqrt{2}}tt + \frac{1}{\sqrt{2}}ff \right) \xrightarrow{\text{Distr.}} \frac{1}{\sqrt{2}}(y \mapsto tt) + \frac{1}{\sqrt{2}}(y \mapsto ff).$$

Por conta dessa propriedade, a avaliação prossegue em dois ramos paralelos:

$$\frac{1}{\sqrt{2}}\langle h_1, p_1 \rangle \mapsto W \langle tt, 0 \rangle + \frac{1}{\sqrt{2}}\langle h_1, p_1 \rangle \mapsto W \langle ff, 0 \rangle.$$

gerando a combinação linear

$$\frac{1}{\sqrt{2}}\langle h_1, p_1 \rangle \mapsto \langle tt, -1 \rangle + \frac{1}{\sqrt{2}}\langle h_1, p_1 \rangle \mapsto \langle ff, 1 \rangle.$$

O isomorfismo é aplicado recursivamente a próxima direção da lista, acompanhado de ambos os índices obtidos pela combinação linear:

$$\frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto rec \langle t, -1 \rangle + \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto rec \langle t, 1 \rangle \mid t \mapsto (tt : []).$$

Como todos os valores da lista são iniciados em tt , os passos de avaliação são repetidos. Sabendo que o próximo passo de recursão será na lista vazia, a avaliação de $rec \langle t, -1 \rangle$ gera uma combinação linear dos conjuntos de mapeamento de variáveis:

$$\begin{aligned} & \frac{1}{\sqrt{2}}\{h^1 \mapsto tt; p^1 \mapsto -1; y^1 \mapsto tt; h_1^1 \mapsto tt; p_1^1 \mapsto -2; t_1^1 \mapsto []; p_2^1 \mapsto -2\} \\ & + \frac{1}{\sqrt{2}}\{h^1 \mapsto tt; p^1 \mapsto -1; y^1 \mapsto ff; h_1^1 \mapsto ff; p_1^1 \mapsto 0; t_1^1 \mapsto []; p_2^1 \mapsto 0\} \end{aligned},$$

e $rec \langle t, 1 \rangle$ gera os conjuntos

$$\begin{aligned} & \frac{1}{\sqrt{2}}\{h^1 \mapsto tt; p^1 \mapsto 1; y^1 \mapsto tt; h_1^1 \mapsto tt; p_1^1 \mapsto 0; t_1^1 \mapsto []; p_2^1 \mapsto 0\} \\ & + \frac{1}{\sqrt{2}}\{h^1 \mapsto tt; p^1 \mapsto 1; y^1 \mapsto ff; h_1^1 \mapsto ff; p_1^1 \mapsto 2; t_1^1 \mapsto []; p_2^1 \mapsto 2\} \end{aligned}.$$

Com esses conjuntos é possível identificar o resultado da aplicação recursiva em cada ramo de avaliação:

$$\begin{aligned} & \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \left(\frac{1}{\sqrt{2}}\langle tt : [], -2 \rangle + \frac{1}{\sqrt{2}}\langle ff : [], 0 \rangle \right) \\ & + \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \left(\frac{1}{\sqrt{2}}\langle tt : [], 0 \rangle + \frac{1}{\sqrt{2}}\langle ff : [], 2 \rangle \right) \\ & \xrightarrow{Distr.} \\ & \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \langle tt : [], -2 \rangle + \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \langle ff : [], 0 \rangle \\ & + \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \langle tt : [], 0 \rangle + \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}}\langle t_1, p_2 \rangle \mapsto \langle ff : [], 2 \rangle \end{aligned}$$

O resultado final da avaliação é dado pela substituição das variáveis no termo $\langle h_1 : t_1, p_2 \rangle$

em cada ramo da combinação linear, juntamente com a multiplicação escalar das amplitudes:

$$X2 \langle (tt : tt : []), 0 \rangle \xrightarrow{\text{Evals to}} \frac{1}{2} \langle tt : tt : [], -2 \rangle + \frac{1}{2} \langle tt : ff : [], 0 \rangle \\ + \frac{1}{2} \langle ff : tt : [], 0 \rangle + \frac{1}{2} \langle ff : ff : [], 2 \rangle$$

O resultado da avaliação do termo é compatível com o comportamento descrito no modelo descrito por Ying (2016). Isso sugere que o modelo de recursão em listas pode efetivamente ser descrito como análogo ao modelo de moedas de Ying. No que se refere a reversibilidade, a aplicação do termo resultante ao isomorfismo construído através da inversão de $X2$ (Figura 4.33) resulta em uma combinação linear com amplitudes convergindo em $\langle (tt : tt : []), 0 \rangle$, devido ao armazenamento de todos os passos de direção na lista final.

Para a demonstrar a reversibilidade, considera-se a regra para a aplicação de combinações lineares a isomorfismos:

$$X2^{-1} \left(\frac{1}{2} (\langle tt : tt : [], -2 \rangle + \langle tt : ff : [], 0 \rangle + \langle ff : tt : [], 0 \rangle + \langle ff : ff : [], 2 \rangle) \right) \\ \xrightarrow{\text{Evals}} \frac{1}{2} (X2^{-1} \langle tt : tt : [], -2 \rangle) + \frac{1}{2} (X2^{-1} \langle tt : ff : [], 0 \rangle) \\ + \frac{1}{2} (X2^{-1} \langle ff : tt : [], 0 \rangle) + \frac{1}{2} (X2^{-1} \langle ff : ff : [], 2 \rangle)$$

Sabendo que a inversão dos isomorfismos compondo a recursão é realizada de forma que $had^{-1} = had$, $prev^{-1} = next$ e $next^{-1} = prev$, a avaliação inversa é ilustrada através da aplicação $(X2^{-1} \langle tt : tt : [], -2 \rangle)$.

Considerando a utilização do sobrescrito ¹ na nomeação das variáveis, de forma que $t_1 \mapsto (tt : [])$ e $t_1^1 \mapsto []$, tem-se:

$$\langle t_1^1, p_1^1 \rangle \mapsto rec \langle [], -2 \rangle = \langle [], -2 \rangle,$$

$$\langle y^1, p^1 \rangle \mapsto W^{-1} \langle tt, -2 \rangle = \langle tt, -1 \rangle,$$

$$h^1 \mapsto had tt = \frac{1}{\sqrt{2}} tt + \frac{1}{\sqrt{2}} ff.$$

Logo, o resultado de $rec \langle tt : [], -2 \rangle$ é dado por

$$\frac{1}{\sqrt{2}} \langle t, p_1 \rangle \mapsto \langle tt : [], -1 \rangle + \frac{1}{\sqrt{2}} \langle t, p_1 \rangle \mapsto \langle ff : [], -1 \rangle.$$

Segue-se

$$\langle y, p \mapsto \frac{1}{\sqrt{2}} W^{-1} \langle tt, -1 \rangle + \frac{1}{\sqrt{2}} W^{-1} \langle tt, -1 \rangle \xrightarrow{Distr.} \frac{1}{\sqrt{2}} \langle y, p \mapsto \langle tt, 0 \rangle + \frac{1}{\sqrt{2}} \langle tt, 0 \rangle.$$

E enfim

$$h \mapsto had \left(\frac{1}{\sqrt{2}} y \mapsto tt + \frac{1}{\sqrt{2}} y \mapsto tt \right) \xrightarrow{Distr.} \frac{1}{\sqrt{2}} h \mapsto had tt + \frac{1}{\sqrt{2}} h \mapsto had tt.$$

Resultando no valor

$$v = \frac{1}{2} \langle tt : tt : [], 0 \rangle + \frac{1}{2} \langle tt : ff : [], 0 \rangle + \frac{1}{2} \langle ff : tt : [], 0 \rangle + \frac{1}{2} \langle ff : ff : [], 0 \rangle.$$

As demais avaliações dos valores na combinação linear procedem similarmente, com o valor resultante:

$$\begin{aligned} & \frac{1}{2} \left(\frac{1}{2} \langle tt : tt : [], 0 \rangle + \frac{1}{2} \langle tt : ff : [], 0 \rangle + \frac{1}{2} \langle ff : tt : [], 0 \rangle + \frac{1}{2} \langle ff : ff : [], 0 \rangle \right) \\ & + \frac{1}{2} \left(\frac{1}{2} \langle tt : tt : [], 0 \rangle - \frac{1}{2} \langle tt : ff : [], 0 \rangle + \frac{1}{2} \langle ff : tt : [], 0 \rangle - \frac{1}{2} \langle ff : ff : [], 0 \rangle \right) \\ & + \frac{1}{2} \left(\frac{1}{2} \langle tt : tt : [], 0 \rangle + \frac{1}{2} \langle tt : ff : [], 0 \rangle - \frac{1}{2} \langle ff : tt : [], 0 \rangle - \frac{1}{2} \langle ff : ff : [], 0 \rangle \right) \\ & + \frac{1}{2} \left(\frac{1}{2} \langle tt : tt : [], 0 \rangle - \frac{1}{2} \langle tt : ff : [], 0 \rangle - \frac{1}{2} \langle ff : tt : [], 0 \rangle + \frac{1}{2} \langle ff : ff : [], 0 \rangle \right) \end{aligned}$$

No qual o cancelamento das amplitudes de todos os termos exceto $\langle tt : tt : [], 0 \rangle$ é visível.

Considerando as definições para os algoritmos de *quantum-walk* apresentados nas Figuras 4.31 e 4.35, o algoritmo descrito por Ying através da equação

$$X = (T_L[p] \oplus_{H[d]} T_R[p])^n; ((T_L[p]; X) \oplus_{H[d]} (T_R[p]; X)),$$

é adaptado para a sintaxe de isomorfismos através da construção da Figura 4.36.

Figura 4.33 – Inversão do isomorfismo X2.

$$X2^{-1}:: \left\{ \begin{array}{ll} \langle [], p \rangle & \leftrightarrow \langle [], p \rangle + 0 \\ \langle h_1 : t_1, p_2 \rangle & \leftrightarrow \text{let } \langle t, p_1 \rangle = \text{rec } \langle t_1, p_2 \rangle \text{ in} \\ & \text{let } \langle y, p \rangle = W^{-1} \langle h_1, p_1 \rangle \text{ in} \\ & \text{let } h = \text{had}^{-1} y \text{ in} \\ & 0 + \langle h : t, p \rangle \end{array} \right\}$$

Fonte: Elaborada pelo autor, 2018.

Figura 4.34 – Transformação linear para um *quantum-walk*.

$$\begin{aligned}
W:: (\text{Int} \leftrightarrow \text{Int}) \rightarrow (\text{Int} \leftrightarrow \text{Int}) \rightarrow (\mathbf{B}^* \text{Int} \leftrightarrow \mathbf{B}^* \text{Int}) = \\
\lambda \text{prev} . \lambda \text{next} \\
\left\{ \begin{array}{l} \langle tt, p \rangle \leftrightarrow \text{let } n = \text{prev } p \text{ in } \langle tt, n \rangle + 0 \\ \langle ff, p \rangle \leftrightarrow \text{let } n = \text{next } p \text{ in } 0 + \langle ff, n \rangle \end{array} \right\}
\end{aligned}$$

Fonte: Elaborada pelo autor, 2018.

Figura 4.35 – *Quantum walk* bidirecionalmente recursivo.

$$\begin{aligned}
X2:: (\mathbf{B} \leftrightarrow \mathbf{B}) \rightarrow (\mathbf{B}^* \text{Int} \leftrightarrow \mathbf{B}^* \text{Int}) \rightarrow ([\mathbf{B}]^* \text{Int} \leftrightarrow [\mathbf{B}]^* \text{Int}) = \\
\lambda \text{had} . \lambda W . \mu \text{rec} \\
\left\{ \begin{array}{l} \langle [], p \rangle \leftrightarrow \langle [], p \rangle + 0 \\ \langle h : t, p \rangle \leftrightarrow \text{let } y = \text{had } h \text{ in} \\ \quad \text{let } \langle h_1, p_1 \rangle = W \langle y, p \rangle \text{ in} \\ \quad \text{let } \langle t_1, p_2 \rangle = \text{rec } \langle t, p_1 \rangle \text{ in} \\ \quad 0 + \langle h_1 : t_1, p_2 \rangle \end{array} \right\}
\end{aligned}$$

Fonte: Elaborada pelo autor, 2018.

Figura 4.36 – *Quantum-walk* recursivo descrito por Ying.

$$\begin{aligned}
\text{YingsRecursiveWalk}:: (\mathbf{B} \leftrightarrow \mathbf{B}) \rightarrow (\mathbf{B}^* \text{Int} \leftrightarrow \mathbf{B}^* \text{Int}) \rightarrow ([\mathbf{B}]^* \text{Int} \leftrightarrow [\mathbf{B}]^* \text{Int}) = \\
\lambda \text{singleStepWalk} . \mu \text{fixP} \\
\left\{ \begin{array}{l} \langle [], p \rangle \leftrightarrow \langle [], p \rangle + 0 \\ \langle d : t, p \rangle \leftrightarrow \text{let } \langle d_1, p_1 \rangle = \text{singleStepWalk } \langle y, p \rangle \text{ in} \\ \quad \text{let } \langle d_2, p_2 \rangle = \text{singleStepWalk } \langle d_1, p_1 \rangle \text{ in} \\ \quad \dots \\ \quad \text{let } \langle d_n, p_n \rangle = \text{singleStepWalk } \langle d_{n-1}, p_{n-1} \rangle \text{ in} \\ \quad \text{let } \langle t_R, p_R \rangle = \text{fixP } \langle t, p_n \rangle \text{ in} \\ \quad 0 + \langle d_n : t_R, p_R \rangle \end{array} \right\}
\end{aligned}$$

Fonte: Elaborada pelo autor, 2018.

Uma segunda maneira de denotar o programa recursivo é modificar o isomorfismo da Figura 4.36 para que as n repetições do passo sequencial sejam expressadas em um único construto com a forma

$$\text{let } \langle d_n, p_n \rangle = \text{singleStep}(\text{singleStep}(\dots(\text{singleStep}(\langle [d, p] \rangle)))) \text{ in } \dots$$

O comportamento curioso desse algoritmo é o cancelamento de execuções recursivas devido as amplitudes dos valores gerados pela aplicação sequencial de um passo único de caminhada. Como exemplo, considera-se a avaliação de 3 passos sequenciais no valor inicial $\langle tt, 0 \rangle$:

$$\begin{aligned} & \frac{1}{2\sqrt{2}} \langle tt, -3 \rangle + \frac{1}{2\sqrt{2}} \langle ff, -1 \rangle + \frac{1}{2\sqrt{2}} \langle tt, -1 \rangle - \frac{1}{2\sqrt{2}} \langle ff, 1 \rangle \\ & + \frac{1}{2\sqrt{2}} \langle tt, -1 \rangle + \frac{1}{2\sqrt{2}} \langle ff, 1 \rangle - \frac{1}{2\sqrt{2}} \langle tt, 1 \rangle + \frac{1}{2\sqrt{2}} \langle ff, 3 \rangle. \end{aligned}$$

As amplitudes destacadas em **negrito** geram o cancelamento dos termos, fazendo com que a chamada recursiva que deveria seguir a aplicação sequencial não seja executada para tais valores. Discussões sobre o significado desse tipo de cancelamento gerado pela recursão quântica e suas potenciais aplicações são abordadas em Ying (2016). No que se refere ao presente trabalho, é importante discutir como esse fenômeno se manifesta na implementação da semântica da linguagem. Quando considerada a avaliação do isomorfismo apresentado na Figura 4.36, o cancelamento das amplitudes se dá apenas após a execução recursiva em todos os valores. Isso se deve a maneira pela qual a propriedade distributiva das expressões *let* foi implementada. Em essência, uma expressão *let* deve ser completamente avaliada em ambos os ramos de uma combinação linear antes de agregar as amplitudes, fazendo com que o cancelamento de termos só ocorra ao final da avaliação completa do isomorfismo.

$$\begin{aligned} & \text{let } \langle d_1, p_1 \rangle = \text{singleStepWalk} \langle tt, 0 \rangle \text{ in } e \\ & \xrightarrow{\text{Eval}} \text{let } \langle d_1, p_1 \rangle = \frac{1}{\sqrt{2}} \langle tt, -1 \rangle + \frac{1}{\sqrt{2}} \langle ff, 1 \rangle \text{ in } e \\ & \xrightarrow{\text{Eval}} \frac{1}{\sqrt{2}} (\text{let } \langle d_1, p_1 \rangle = \langle tt, -1 \rangle \text{ in } e) + \frac{1}{\sqrt{2}} (\text{let } \langle d_1, p_1 \rangle = \langle ff, 1 \rangle \text{ in } e) \end{aligned}$$

Por outro lado, a avaliação recursiva gerada pela aplicação sequencial dos isomorfismos através de um único termo segue o comportamento esperado. Devido a aplicação das propriedades algébricas em valores após a avaliação de aplicações, o cancelamento dos termos ocorrerá após a terceira aplicação sequencial. Embora o valor final gerado por ambas as representações seja o mesmo, essa dissonância entre

o comportamento das avaliações indica uma alteração em potencial na forma pela qual as expressões let são avaliadas.

5 DISCUSSÃO FINAL

Este trabalho apresentou a implementação de um interpretador e *typechecker* para uma linguagem reversível quântica. A linguagem descreve sua reversibilidade através da noção de *pattern-matching* simétrico e codifica a noção de controle quântico, incluindo a execução de funções recursivas em um contexto quântico.

Como linguagem fonte da implementação, a escolha pelo paradigma funcional através da linguagem Haskell simplificou a especificação das regras de inferência da linguagem, fornecendo uma sintaxe próxima a notação matemática das regras. Através do uso de mônadas, a linguagem facilita o gerenciamento de estruturas de dados representando os contextos de tipos, fornecendo métodos de acesso explícito que localizam qualquer alteração.

A execução dos exemplos disponibilizados em conjunto com a implementação este trabalho demonstra que o interpretador codifica apropriadamente a semântica da linguagem, permitindo a avaliação usual e inversa de programas no contexto da computação clássica e de programas em um contexto quântico.

A principal contribuição do trabalho é a construção de um sistema capaz de simular o comportamento de algoritmos reversíveis e de sistemas quânticos. Além disso, o texto apresenta exemplos que comprovam a codificação do controle quântico na semântica da linguagem, incluindo a recursão quântica através de listas. Finalmente, na demonstração do algoritmo de *quantum-walk* recursivo, o trabalho apresenta evidências de que a semântica da linguagem possui comportamento equivalente a proposta de recursão quântica estabelecida por Ying (2016).

5.1 TRABALHOS RELACIONADOS

Pode-se mencionar relações entre este projeto e o trabalho desenvolvido por James e Sabry (2012). Os autores apresentam uma linguagem reversível simplificada, fornecendo a base para as relações de isomorfismo de tipos. Seu trabalho inclui uma implementação em Haskell de um interpretador para a linguagem, onde as relações isomórficas são codificadas via funções em código-fonte Haskell, com programas representados via composição dessas. A avaliação e inversão de programas é modelada em funções recursivas que simulam o comportamento da linguagem. Apesar de possuir uma abordagem similar, o escopo da semântica implementada pelos autores é reduzido, não possibilitando a construção e avaliação de funções de alta ordem ou recursão.

Outra implementação relacionada de uma linguagem recursiva é realizada por Thomsen e Axelsen (2015). Os autores escolheram pela implementação na

linguagem Haskell, fazendo amplo uso de mônadas para a implementação semântica. A reversibilidade da linguagem é codificada através de funções específicas para a inversão de programas, fazendo com que a interpretação inversa deva ser invocada manualmente após a inversão. A implementação de um *typechecker* para a linguagem é um projeto em curso. O trabalho aqui desenvolvido é similar na maneira com a qual a inversão de programas é tratada, permitindo a avaliação inversa através de chamadas manuais. Além disso, o sistema aqui apresentado é estendido para possibilitar a expressão e simulação de programas quânticos, codificando transformações unitárias através de isomorfismos de tipos, e permitindo a expressão de valores quânticos através de combinações lineares de amplitudes complexas. Por fim, os autores discutem ainda a implementação de um auto-interpretador, ou seja, um programa escrito na linguagem reversível capaz de interpretar programas na mesma linguagem, tópico não abordado na construção do interpretador resultado deste trabalho.

5.2 TRABALHOS FUTUROS

Um ponto levantado durante a codificação do algoritmo de *quantum-walk* recursivo refere-se ao cancelamento de amplitudes. Na implementação atual, quando um algoritmo quântico é representado através de uma sequência de expressões let, o cancelamento de amplitudes ocorre somente após a avaliação completa de todos os ramos gerados pela propriedade distributiva, mesmo que existam amplitudes canceláveis nos passos intermediários. A Figura B.1 (Apêndice B) ilustra o procedimento de avaliação de expressões let através da propriedade distributiva, enquanto a Figura B.2 ilustra uma implementação alternativa da avaliação. A primeira abordagem visivelmente gera passos de avaliação extra quando comparada com a segunda, utilizando mais recursos. Uma vantagem dessa implementação é a facilidade na adoção de avaliação em paralelo no futuro, já que não existem interações entre os ramos de avaliação, sendo necessária a sincronização ao final de cada avaliação. Já a segunda reduz o número de avaliações executadas mas incorre no custo de comparar as amplitudes a cada passo de forma síncrona. A investigação aprofundada das duas abordagens pode ser um tópico de trabalho futuro, visando gerar um sistema de tomada de decisão que avalie qual o modelo mais adequado.

A eficiência das simulações possibilitadas pelo projeto resultante deste trabalho é um tópico interessante para investigações futuras. Embora o trabalho não tenha especificado métricas de desempenho formais, pode-se afirmar que a implementação é ineficiente tanto no quesito tempo computacional quanto no quesito uso de memória, especialmente visível na avaliação inversa de isomorfismos que resultam em combinações lineares. O maior fator nesses quesitos é a implementação das amplitudes complexas

através de uma representação numérica com alta precisão, reduzindo o desempenho em operações aritméticas e matriciais. Algumas opções de refatoração de código para otimizar o desempenho do sistema são:

- Incluir a avaliação paralela de aplicações de combinações lineares de valores a isomorfismos. Dada a semântica da linguagem, é possível avaliar a aplicação de cada valor da combinação linear paralelamente, sincronizando o resultado de cada avaliação na construção do valor final.
- Paralelizar a aplicação das propriedades algébricas da linguagem. Considerando uma combinação da forma: $e_1 + e_2 + \dots + e_n$, a aplicação das propriedades algébricas em cada e_i pode ser realizada em paralelo, com os valores finais sincronizados por uma aplicação única das propriedades na combinação resultante.
- Verificação aprofundada da checagem de tipos, em especial da verificação da unitariedade dos isomorfismos. Essa verificação é implementada através da transformação de listas de amplitudes em matrizes via funções implementadas pelo módulo `Data.Matrix`. Uma verificação otimizada pode ser obtida através da manipulação direta das listas, evitando o custo com a construção das matrizes.
- Alteração da implementação numérica. O projeto atual utilizou a implementação numérica oferecida pelo tipo `CReal` devido a alta precisão, visando evitar preocupações de erros aritméticos na verificação semântica. Uma investigação cuidadosa das necessidades da linguagem, acompanhada de rotinas para correção de erros pode ser abordada para possibilitar a codificação de números complexos através de implementações com menor precisão, reduzindo o custo de memória associado.

REFERÊNCIAS BIBLIOGRÁFICAS

- AARONSON, S. **Quantum computing since Democritus**. [S.l.]: Cambridge University Press, 2013.
- ALLEN, C.; MORONUKI, J. **Haskell Programming from First Principles**. [S.l.: s.n.], 2017.
- ALTENKIRCH, T.; GRATTAJE, J. A functional quantum programming language. In: **IEEE. Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on**. [S.l.], 2005. p. 249–258.
- AMIN, N.; LABELLE, P. An elementary derivation of the deutsch-jozsa algorithm. **arXiv preprint arXiv:0810.2285**, 2008.
- BENNETT, C. H. Logical reversibility of computation. **IBM journal of Research and Development**, IBM, v. 17, n. 6, p. 525–532, 1973.
- CARDELLI, L. Type systems. **ACM Computing Surveys**, v. 28, n. 1, p. 263–264, 1996.
- CARDELLI, L.; WEGNER, P. On understanding types, data abstraction, and polymorphism. **ACM Computing Surveys (CSUR)**, ACM, v. 17, n. 4, p. 471–523, 1985.
- COUSINEAU, G.; MAUNY, M. **The functional approach to programming**. [S.l.]: Cambridge University Press, 1998.
- CURRY, H. B. et al. **Combinatory logic**. [S.l.]: North-Holland Amsterdam, 1958. v. 1.
- FRANK, M. P. Introduction to reversible computing: motivation, progress, and challenges. In: **ACM. Proceedings of the 2nd Conference on Computing Frontiers**. [S.l.], 2005. p. 385–390.
- GROVER, L. K. Quantum mechanics helps in searching for a needle in a haystack. **Physical review letters**, APS, v. 79, n. 2, p. 325, 1997.
- GUNTER, C. A. **Semantics of programming languages: structures and techniques**. [S.l.]: MIT press, 1992.
- HUDAK, P. Conception, evolution, and application of functional programming languages. **ACM Computing Surveys (CSUR)**, ACM, v. 21, n. 3, p. 359–411, 1989.
- HUDAK, P. et al. **The haskell school of music**. **Yale University**, 2008.
- HUGHES, J. Why functional programming matters. **The computer journal**, Oxford University Press, v. 32, n. 2, p. 98–107, 1989.
- JAMES, R. P.; SABRY, A. Information effects. In: **ACM. ACM SIGPLAN Notices**. [S.l.], 2012. v. 47, n. 1, p. 73–84.
- LANDAUER, R. Irreversibility and heat generation in the computing process. **IBM journal of research and development**, lbm, v. 5, n. 3, p. 183–191, 1961.
- MERMIN, N. D. From cbits to qbits: Teaching computer scientists quantum mechanics. **American Journal of Physics**, AAPT, v. 71, n. 1, p. 23–30, 2003.

NIELSEN, M. A.; CHUANG, I. **Quantum computation and quantum information**. [S.l.]: AAPT, 2002.

NIELSON, H. R.; NIELSON, F. Semantics with applications. **An Appetizer: Springer Verlag London Ltd**, Springer, 2007.

ÖMER, B. A procedural formalism for quantum computing. Citeseer, 1998.

O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. B. **Real world haskell: Code you can believe in**. [S.l.]: "O'Reilly Media, Inc.", 2008.

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

SABRY, A.; FELLEISEN, M. Reasoning about programs in continuation-passing style. **Lisp and symbolic computation**, Springer, v. 6, n. 3-4, p. 289–360, 1993.

SABRY, A.; VALIRON, B.; VIZZOTTO, J. K. **From Symmetric Pattern-Matching to Quantum Control**. [S.l.], 2017. 13 p.

SEBESTA, R. W. **Conceitos de linguagens de programação**. [S.l.]: Bookman Editora, 2009.

SELINGER, P. Towards a quantum programming language. **Mathematical Structures in Computer Science**, Cambridge University Press, v. 14, n. 4, p. 527–586, 2004.

SHANNON, C. E. A mathematical theory of communication. **ACM SIGMOBILE Mobile Computing and Communications Review**, ACM, v. 5, n. 1, p. 3–55, 2001.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. **SIAM review**, SIAM, v. 41, n. 2, p. 303–332, 1999.

THE UNIVERSITY OF GLASGOW. **Hackage package archive: Prelude**. 2010. Disponível em: <<https://hackage.haskell.org/package/base-4.10.1.0/docs/Prelude.html>>. Acesso em 12 ago. 2017.

THOMPSON, S. **Type theory and functional programming**. [S.l.]: Addison Wesley, 1991.

_____. **Haskell: the Craft of Functional Programming . International Computer Science Series**. [S.l.]: Addison-Wesley, March, 1999.

THOMSEN, M. K.; AXELSEN, H. B. Interpretation and programming of the reversible functional language rfun. In: **Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages**. New York, NY, USA: ACM, 2015. (IFL '15), p. 8:1–8:13. ISBN 978-1-4503-4273-5. Disponível em: <<http://doi.acm.org/10.1145/2897336.2897345>>.

TOFFOLI, T. Reversible computing. **Automata, languages and programming**, Springer, p. 632–644, 1980.

TONDER, A. V. A lambda calculus for quantum computation. **SIAM Journal on Computing**, SIAM, v. 33, n. 5, p. 1109–1135, 2004.

YING, M. **Foundations of Quantum Programming**. [S.l.]: Morgan Kaufmann, 2016.

YOKOYAMA, T.; AXELSEN, H. B.; GLÜCK, R. Principles of a reversible programming language. In: ACM. **Proceedings of the 5th conference on Computing frontiers**. [S.l.], 2008. p. 43–54.

_____. Towards a reversible functional language. In: SPRINGER. **International Workshop on Reversible Computation**. [S.l.], 2011. p. 14–29.

APÊNDICE A – IMPLEMENTAÇÃO DOS EXEMPLOS VIA SINTAXE ABSTRATA

Figura A.1 – Isomorfismos implementando Hadamard e o algoritmo de Deutsch.

```

1  hadIso :: (Iso,T) — Hadamard iso
2  hadIso = let a1 = (1/√2)::CReal
3              a2 = (-1/√2)::CReal —fixed precision numbers
4              alpha = (a1 :+ 0)
5              beta = (a2 :+ 0) — complex numbers
6              eTT = Val tt —ExtendedValue true
7              eFF = Val ff —ExtendedValue false
8              e1 = Combination (AlphaVal alpha eTT) (AlphaVal alpha eFF) — Clause 1
9              e2 = Combination (AlphaVal alpha eTT) (AlphaVal beta eFF) — Clause 2
10             had = Clauses [(tt,e1),(ff,e2)] —
11             isoType = Iso bool bool —Type of Had iso
12             in (had,isoType)
13
14  deutsch :: (Iso,T) — Deutsch's algorithm
15  deutsch = let v1 = PairV (Xval "x") (Xval "y")
16              e1 = LetE (Xprod "h1") (IsoVar "had") (Xprod "x") e2
17              e2 = LetE (Xprod "h2") (IsoVar "had") (Xprod "y") e3
18              e3 = LetE (Xprod "q") (IsoVar "Oracle") (PairP (Xprod "h1") (Xprod "h2"))
19                  e4
20              e4 = LetE (Xprod "out") (IsoVar "hadX") (Xprod "q") e5
21              e5 = AlphaVal (1:+0) (Val $ Xval "out")
22              clauses = Clauses [(v1,e1)]
23              iso = Lambda "had" (Lambda "Oracle" (Lambda "hadX" clauses))
24              ty = Iso (Prod bool bool) (Prod bool bool)
25              isoType = Comp (bool) (bool) $ Comp (Prod bool bool) (Prod bool bool) $
26                  Comp (Prod bool bool) (Prod bool bool) ty
27             in (iso,isoType)
28
29  oracle2 :: (Iso,T)
30  oracle2 = let (cnot,_) = simpleCnot
31              (mynot,_) = not1
32              v1 = PairV (Xval "x") (tt)
33              v2 = PairV (Xval "x") (ff)
34              c1 = Val $ PairV (Xval "x") (Xval "w")
35              c1' = Combination (AlphaVal (1:+0) c1) (AlphaVal (0:+0) c1)
36              c1'' = Combination (AlphaVal (0:+0) c1) (AlphaVal (1:+0) c1)
37              e1 = LetE (Xprod "w") (IsoVar "f") (Xprod "x") c1'
38              e2 = LetE (Xprod "w") mynot (Xprod "z") c1''
39              e2' = LetE (Xprod "z") (IsoVar "f") (Xprod "x") e2
40              clauses = Clauses [(v1,e1),(v2,e2')]
41              oracle2 = Lambda "f" clauses
42              ty = Iso (Prod bool bool) (Prod bool (Prod bool bool)) — Not the right
43                  type !!
44             in (oracle2,ty)

```

Fonte: Elaborado pelo autor, 2018.

Figura A.2 – Implementação do isomorfismo *next* para inteiros de 5 bits.

```

1 isoNext :: (Iso,T)
2 isoNext = let vList = [fl $ buildInt i 4 'v' | i <-
    [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] ]
3     eList = [Val $ fl $ buildInt i 4 'v' | i <-
    [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0] ]
4     cList = [buildOneZeroCombs eList i 0 | i <-
    [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] ]
5     clauses = Clauses (p vList cList)
6     bitsN = (Prod bool (Prod bool (Prod bool bool)))
7     ty = Iso bitsN bitsN
8     in (clauses, ty)
9
10 nextSigned :: (Iso,T)
11 nextSigned = let zero = fl $ buildInt 0 4 'v'
12     v1 = PairV ff zero
13     v2 = PairV ff (Xval "y")
14     v3 = PairV tt (Xval "y")
15     a1 = Val $ PairV (tt) zero
16     a2 = Val $ PairV (ff) (Xval "y'")
17     a3 = Val $ PairV (tt) (Xval "y'")
18     alist = [a1,a2,a3]
19     c1 = Combination (AlphaVal (1:+0) a1) (Combination (AlphaVal (0:+0) a1
    ) (AlphaVal (0:+0) a1)) —buildOneZeroCombs alist 0 0
20     c2 = buildOneZeroCombs alist 1 0
21     c3 = buildOneZeroCombs alist 2 0
22     e1 = c1
23     e2 = LetE (Xprod "y'") (IsoVar "prev") (Xprod "y") c2
24     e3 = LetE (Xprod "y'") (IsoVar "next") (Xprod "y") c3
25     clauses = Clauses [(v1,e1),(v2,e2),(v3,e3)]
26     iso = Lambda "next" (Lambda "prev" clauses)
27
28     bitsN = (Prod bool (Prod bool (Prod bool bool)))
29
30     fiveBits = Prod bool (Prod bool (Prod bool (Prod bool bool)))
31     ty = Comp bitsN bitsN (Comp bitsN bitsN $ Iso fiveBits fiveBits)
32     in (iso, ty)

```

Fonte: Elaborado pelo autor, 2018.

Figura A.3 – Definição do *quantum-walk* recursivo de Ying.

```

1  walkTIso :: (Iso,T) — An iso implementing a 1d walk W.
2  walkTIso = let v1 = PairV tt (Xval "n")
3                v2 = PairV ff (Xval "n")
4                a1 = Val $ PairV tt (Xval "n'")
5                a2 = Val $ PairV ff (Xval "n'")
6                c1 = buildOneZeroCombs [a1,a2] 0 0
7                c2 = buildOneZeroCombs [a1,a2] 1 0
8                e1 = LetE (Xprod "n'") (IsoVar "prevSigned") (Xprod "n") c1
9                e2 = LetE (Xprod "n'") (IsoVar "nextSigned") (Xprod "n") c2
10               clauses = Clauses [(v1,e1),(v2,e2)]
11               iso = Lambda "prevSigned" (Lambda "nextSigned" clauses)
12               fiveBits = Prod bool (Prod bool (Prod bool (Prod bool bool)))
13               walkBits = Prod bool fiveBits
14               ty = Iso walkBits walkBits
15               (_,prevTy) = prevSigned
16               (_,nextTy) = nextSigned
17               actualType = Comp fiveBits fiveBits (Comp fiveBits fiveBits ty)
18               in (iso,actualType)
19
20 nonCancellingYing722 :: (Iso,T)
21 nonCancellingYing722 = let (had,hTy) = hadIso
22                           v1 = InjL EmptyV
23                           h = Xval "h"
24                           y = Xval "y"
25                           t = Xval "t"
26                           p = Xval "p"
27                           v2 = InjR $ PairV h t
28                           pV1 = PairV v1 p
29                           pV2 = PairV v2 p
30                           c1 = Combination (AlphaVal (1:+0) (Val pV1)) $ AlphaVal
31                               (0:+0) (Val pV1)
32                           let1 = LetE (PairP (Xprod "h1") (Xprod "p1"))
33                               (IsoVar "singleStep") (PairP (Xprod "h") (Xprod "p"
34                               ))
35                               let2
36                               let2 = LetE (PairP (Xprod "h2") (Xprod "p2"))
37                               (IsoVar "singleStep") (PairP (Xprod "h1") (Xprod "
38                               p1"))
39                               let3
40                               let3 = LetE (PairP (Xprod "h3") (Xprod "p3")) — =
41                               (IsoVar "singleStep") (PairP (Xprod "h2") (Xprod "
42                               p2")) — in
43                               let4
44                               let4 = LetE (PairP (Xprod "h4") (Xprod "p4"))
45                               (IsoVar "singleStep") (PairP (Xprod "h3") (Xprod
46                               "p3"))
47                               let5
48                               let5 = LetE (PairP (Xprod "t1") (Xprod "p5"))
49                               (IsoVar "rec") (PairP (Xprod "t") (Xprod "p4"))
50                               c2
51                               newList = InjR $ PairV (Xval "h4") (Xval "t1")
52                               c2 = Combination (AlphaVal (0:+0) (Val pV1)) $ AlphaVal
53                               (1:+0) (Val $ PairV newList (Xval "p5"))
54                               clauses = Clauses [(pV1,c1),(pV2,let1)]
55                               lambda = (Lambda "singleStep" (Fixpoint "rec" clauses))
56                               fiveBits = Prod bool (Prod bool (Prod bool (Prod bool bool)
57                               ))
58                               ty = Comp bool bool (Iso (Prod (Rec bool) fiveBits) (Prod (
59                               Rec bool) fiveBits)) — Doesn't include the composition
60                               type of walkT, need to do this.
61                               in (lambda, ty)

```

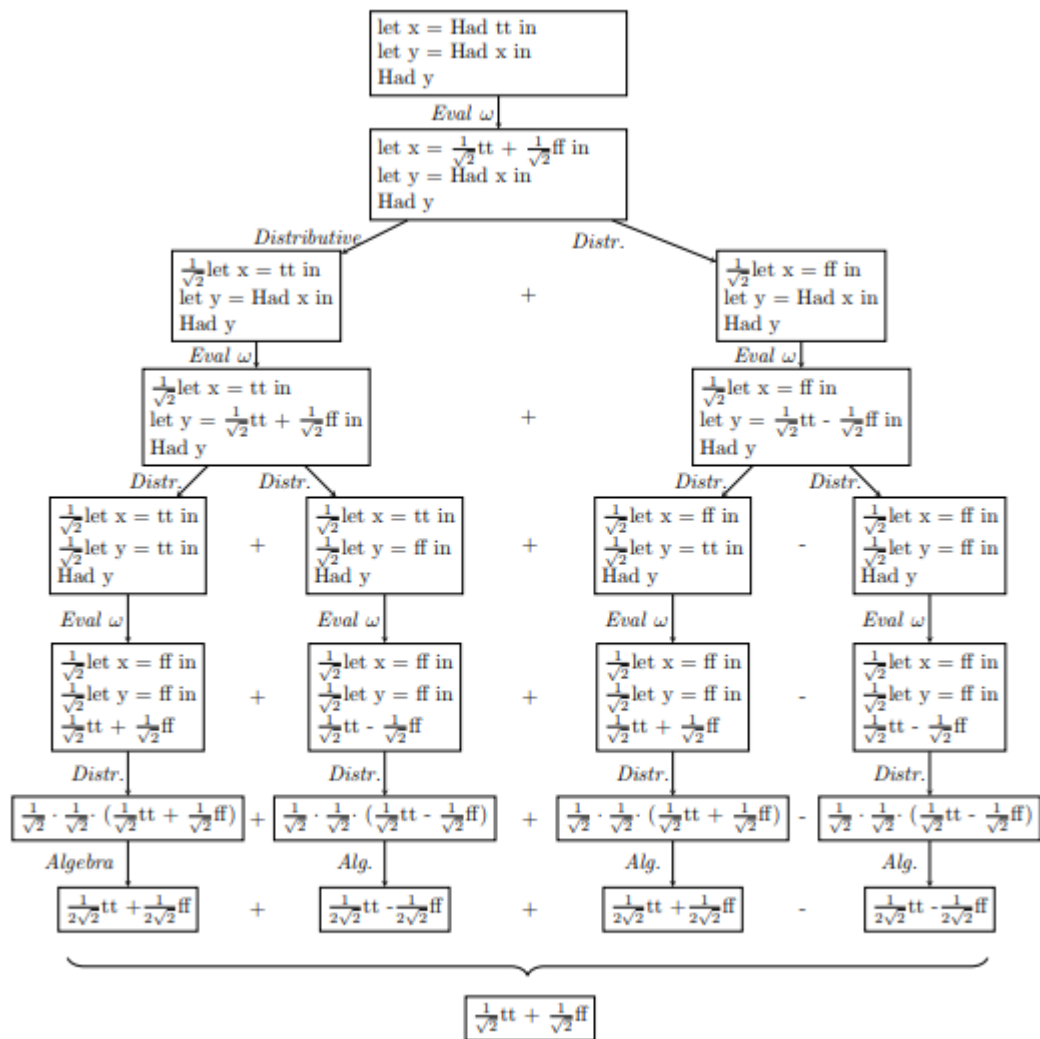
Fonte: Elaborado pelo autor, 2018.

APÊNDICE B – PROPRIEDADES DISTRIBUTIVAS EM EXPRESSÕES LET

Os diagramas incluídos neste apêndice ilustram as possíveis implementações da propriedade distributiva para expressões let. O primeiro diagrama (Figura B.1) ilustra a implementação atual do interpretador: as propriedades distributivas fazem com que a expressão let seja avaliada para cada valor da combinação linear, formando uma estrutura que remete a uma árvore, onde os ramos são caminhos distintos de avaliação que ocorrem sequencialmente. Nesse caso, a aplicação das propriedades algébricas e o cancelamento de amplitudes só pode ser realizado ao final da avaliação de todos os ramos da árvore.

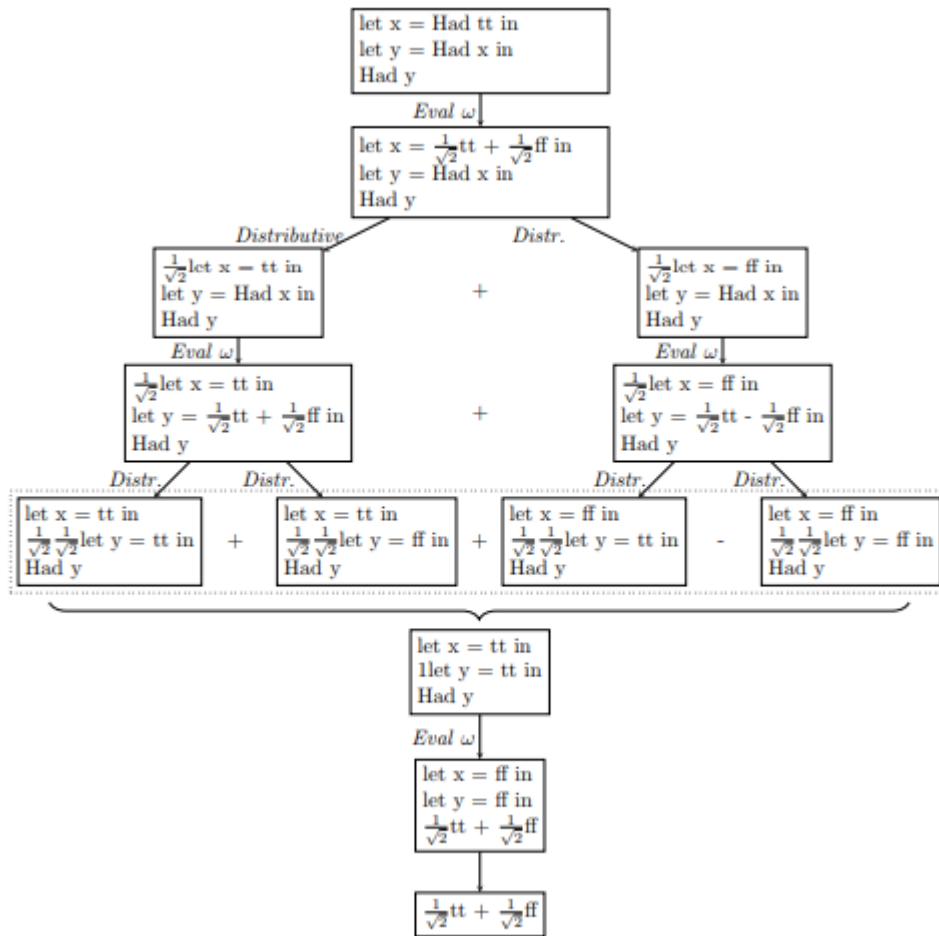
O diagrama da Figura B.2 apresenta uma avaliação alternativa, onde as amplitudes são consideradas em cada passo de avaliação da expressão let, realizando o cancelamento de caminhos de avaliação. A alternativa proposta reduz o número de avaliações necessárias, mas incorre em um custo de tomada de decisões para identificar momentos nos quais o cancelamento é possível.

Figura B.1 – Ilustração da implementação atual das propriedades distributivas.



Fonte: Elaborado pelo autor, 2018.

Figura B.2 – Ilustração de uma avaliação alternativa para as propriedades distributivas.



Fonte: Elaborado pelo autor, 2018.