Renato Pedrosa Vasconcelos

Computação Reversível: Fundamentos e Aplicações

Recife
2022

Computação Reversível: Fundamentos e Aplicações

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Universidade Federal Rural de Pernambuco – UFRPE
Departamento de Estatística e Informática
Curso de Bacharelado em Sistemas de Informação

Prof. Dr. Wilson Rosa de Oliveira Junior
Orientador
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Recife
2022

Computação Reversível: Fundamentos e Aplicações

Renato Pedrosa Vasconcelos [1], Wilson Rosa de Oliveira Junior [2]
[1]Departamento de Estatıstica e Informática – Universidade Federal Rural de Pernambuco (UFRPE)
Rua Dom Manuel de Medeiros, s/n, - CEP: 52171-900 – Recife – PE – Brasil
{renato.pedrosa@ufrpe.br, wilson.oliveirajr@ufrpe.br}

Recife
2022

Renato Pedrosa Vasconcelos

Computação Reversível: Fundamentos e Aplicações

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Aprovado em 17 de outubro de 2022

BANCA EXAMINADORA

Prof. Dr. Wilson Rosa de Oliveira Junior
Orientador
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Prof. Dra. Maigan Stefanne da silva Alcântara
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

V331c Pedrosa, Renato
Computação Reversível: Fundamentos e Aplicações: Fundamentos e Aplicações / Renato Pedrosa. 2022.
70 f. : il.

Orientador: Prof. Dr. Wilson Rosa de Oliveira Junior. Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, , Recife, 2023.

1. Computação Reversível. 2. Logicas Reversivel. 3. Linguagens de programação reversíveis.. I. Junior, Prof. Dr. Wilson Rosa de Oliveira, orient. II. Título

CDD

Ao grande arquiteto do universo Deus que me capacita diariamente.

Agradecimentos

Primeiramente agradeço a Deus por me conceder tanta paciência durante os anos na graduação, pois sem ele dificilmente eu terminaria o curso. À minha mãe, também pela paciência comigo. À minha família que me proporcionou um apoiou fundamental. A todos que, direta ou indiretamente contribuíram para que fosse possível a realização deste trabalho, não posso esquecer das outras universidades por onde andei. Aos professores e à secretaria do curso de Sistemas de Informação da UFRPE. Ao meu orientador, o professor Wilson Rosa de Oliveira Junior foi quem me apresentou o tema em 2020.

“Porque o Senhor dá a sabedoria, e da sua boca vêm a inteligência e o entendimento.”

(Provérbios 2:6)

Resumo

Motivada por suas aplicações promissoras, a lógica reversível recebeu atenção significativa. Como resultado, um progresso impressionante foi feito no desenvolvimento de abordagens de síntese, implementação de elementos sequenciais e linguagens de descrição de hardware. Nesta monografia, essas conquistas recentes são empregadas para à arquitetura PDP-8 e RISC em lógica reversível que possa executar programas de software escritos em linguagem de montagem. Os respectivos componentes combinacionais e sequenciais são projetados usando técnicas de design de última geração. Incentivando a importância do estudo da existência de autômatos logicamente reversíveis sugere que os computadores físicos podem se tornar termodinamicamente reversíveis e, portanto, capazes de dissipar uma quantidade arbitrariamente pequena de energia, por etapas se operados suficientemente devagar. Os computadores de hoje são baseados em dispositivos lógicos irreversíveis, que são conhecidos por serem fundamentalmente ineficientes em termos de energia há várias décadas. A ideia é entender os conceitos em lógica reversível em lógicas irreversíveis que possam executar programas de software escritos em linguagem montagem. Nos modelos tradicionais de computação, a pura reversibilidade parece diminuir a eficiência computacional geral. No entanto, os modelos tradicionais ignoram importantes restrições físicas no processamento de informações. Linguagens de programação reversíveis muito simples podem ser úteis para o estudo de transformações reversíveis. Este estudo buscara processar conceitos conhecimentos para desenvolver a melhoria na funcionalidade e no desempenho de um ambiente de programação reversível.

Palavras-chave: Computação Reversível, Logicas Reversivel,Linguagens de programação reversíveis.

Abstract

Motivated by its promising applications, reversible logic has received significant attention. As a result, impressive progress has been made in developing synthesis approaches, implementing sequential elements, and hardware description languages. In this monograph, these recent achievements are employed for PDP-8 and RISC architecture in reversible logic that can run software programs written in assembly language. The respective combinational and sequential components are designed using state-of-the-art design techniques. Encouraged by the importance of studying the existence of logically reversible automata, it suggests that physical computers can become thermodynamically reversible and therefore able to dissipate an arbitrarily small amount of energy in steps if operated slowly enough. Today's computers are based on irreversible logic devices, which have been known to be fundamentally energy-inefficient for several decades. The idea is to understand the concepts in reversible logic in irreversible logics that can run software programs written in assembly language. In traditional models of computation, sheer reversibility appears to decrease overall computational efficiency. However, traditional models ignore important physical constraints on information processing. Very simple reversible programming languages can be useful for studying reversible transformations. This study processes concepts and knowledge to develop the improvement in functionality and performance of a reversible programming environment.

Keywords: Reversible Computing, Reversible Logics, Reversible Programming Languages.

Lista de ilustrações

Figura 1 - O Código gerado no compilador Janus online código normal…..….....17

Figura 2 - O código há inversão do programa atualmente escrito na área de texto.....18

Figura 3 - A IDE editor compilador……..……..……..…….....18

Figura 4 - O código da linguagem de programação reversível R….....19

Figura 5 - Simulador do pendulum ISA…….....21

Figura 6 - O código do simulador do pendulum ISA (PISA)...…….....21

Figura 7 - A função FOR onde g e h são recursivos primitivos recursiva……..….....21

Figura 8 - O código da função FOR primitivo recursiva..……...…….....22

Figura 9 - A atualização logicamente reversível……...…..…….....25

Figura 10 - Os valores semânticos…….....……...…….....26

Figura 11- O Semântica operacional dos programas JANUS.....…..…..…..….....27

Figura 12 - Representa a gramática da linguagem JANUS…...…. …….….....30

Figura 13 - Mostra uma gramática formal descrevendo as regras de sintaxe R.....31

Figura 14 - Representa a R funções de saída …….....32

Figura 15 - Otimização do compilador…….....…….....36

Figura 16 - Mostra uma gramática formal..... …….....38

Figura 17 - Sintaxe de JANUS.…….....39

Figura 18 - dois códigos esquerdo normal e no lado direito o código reverso.....40

Figura 19 - Domínios de sintaxe e gramática ...…… ……,…… …….….....43

Figura 20 - Mostra o fluxo de controle estruturado reversível……..…….....44

Figura 21 - Regras de inversão para instruções PISA são auto inversas.....47

Figura 22 - Mostra alguns procedimentos da JANUS.....50

Figura 23 - Mostra código em C dois 47

Figura 24 - Mostra o fluxo de controle estruturado reversível.....48

Figura 25 - Condicional e Loop são reversíveis.....50

Figura 26 - Procedimento é reversível chamar/fechar.....51

Figura 27 - Mostra alguns procedimentos da JANUS.....53

Lista de tabelas

Tabela 1 – Na tabela abaixo comparação entre as linguagens SRL e Loop.....15

Tabela 2 - Porta reversível mais simples.....28

Erro! Fonte de referência não encontrada. 3 – As três classes de modelos de máquinas físicas que são comparadas.....44

Lista de abreviaturas e siglas

PISA	O microprocessador Pendulum ISA (PISA)
JANUS	A linguagem de programação reversível Janus
R	A linguagem de programação reversível R
SRL, ESRL e Loop	Linguagem de registradores
CNOT	Porta não contralada
CAD	Computer Aided Design
POO ou OOP	A programação orientada a objetos
MIT	Massachusetts Institute of Technology

Sumário

1. INTRODUÇÃO.....9

1.1 Justificativa.....11
1.2 Objetivo.....10
1.3 Objetivos Específicos.....11
1.4 Organização do trabalho.....11
2. TRABALHOS RELACIONADOS 12
2.1 O circuitos lógicos porta deToffoli 12
2.2 Informações logicamente reversível como a pagamento de bit.....12
2.3 Janus uma linguagem de programação imperativa reversível.....12
2.4 A linguagem de programação reversível R.....13
2.5 O microprocessador Pendulum (PISA) uma criada no MIT.....13
3. REFERENCIAL TEÓRICO.......13
3.1 As linguagens LOOP(N) e SRL(Z).....14
3.2 A linguagem de programação reversível Janus.....18
3.3 A linguagem de programação reversível R.....21
3.4 O conjunto de instruções do pêndulo arquitetura (PISA).....22
3.5 As funções computáveis FOR são recursivas primitivas.....25
4. MATERIAIS e MÉTODOS 26
4.1. Conjuntos de dados.....26
4.2 Dissipação de energia.....26
4.3 Fundamentos de linguagem reversível.....29
4.4 Atualizações reversíveis.....29
4.5 Portas reversíveis.....31
4.6 Janus.....35
4.7 A linguagem de registradores Loop e SRL.....37
5. METODOLOGIA 37
5.1 Analisador de sintaxe da linguagem JANUS.....38
5.2 R a linguagem de programação reversível.....39
5.3 Otimização de código de montagem de Pendulum.....41
5.4 O microprocessador Pendulum (PISA).....43
6. RESULTADOS e DISCUSSÃO.....44
6.1 Linguagem de progrmação R.....45
6.2 O microprocessador Pendulum ISA (PISA).....46
6.3 Janus.....55
7. COMPARAÇÃO dos TRABALHOS......56
8 . CONCLUSÕES e TRABALHOS FUTUROS......59
Referências......60

1.Introdução

Quando uma computação convencional da maquina de Turing foi realizada fisicamente, a destruição da informação tem um custo físico na forma de dissipação de calor. Por outro lado, se nenhum bit for apagado durante a computação, em teoria não há limite inferior de dissipação de calor para a computação reversível. Os estudos começaram a se desenvolver acerca da computação reversível a partir dos 60, com os circuitos lógicos composto pela a porta de Toffoli, TOFFOLI.**(1980)** e também a porta **CNOT.** A porta lógica de Toffoli é universal no sentido que circuitos reversíveis poderiam ser construídos a partir das portas de Toffoli. A computação quântica tem relação com portas lógicas reversíveis, portanto a porta de Toffoli também é um operador lógico. Enquanto as pesquisas se desenvolviam em circuitos reversíveis, em paralelo começavam os estudos em relação a linguagem de programação reversíveis. Muito progresso foi feito para alcançar a reversibilidade nos níveis de circuito e portas, mas não na linguagem de programação de alto nível. O ideal era criar uma linguagem de programação reversível de alto nível que seria capaz de ser compilada em uma linguagem montagem reversível de baixo nível sem sobrecargas significativas.

O princípio de Landauer, é considerado como o princípio básico da termodinâmica do processamento de informações, afirma que "qualquer manipulação logicamente irreversível de informações, como apagar um bit ou a combinação de dois caminhos de computacionais, deve ser acompanhado por um correspondente aumento de entropia em graus de liberdade não portadores de informação do aparato de processamento de informação ou seu ambiente" Bennett . (1973). Talvez isto seja um empecilho para o futuro da computação em si levando em conta o desenvolvimento de máquinas reversíveis que tenham a capacidade de diminuir o gasto energético com cálculos reversíveis.

Uma arquitetura é totalmente irreversível se, por exemplo, ela usa rotineiramente portas lógicas irreversíveis comuns, que devem produzir entropia toda vez que apagam um bit, de acordo com o princípio de Landauer Bennett1. (1973). Portanto, um computador deve dissipar pelo menos $kT\ em\ 2$ de energia (cerca de $3\ x\ 10^{-21}$ joule à temperatura ambiente) para cada bit de informação apagado ou jogado fora.

Um computador irreversível sempre pode ser reversível fazendo com que ele salve todas as informações que de outra forma jogaria fora, Por exemplo, uma máquina deterministica que pode receber uma fita extra (inicialmente em branco) na qual poderia

gravar cada operação como ela estava sendo executada, detalha o estado insuficientes, de que o estado anterior, seria determinado exclusivamente pelo estado atual do último registro na fita. Estudos afirmam que é possível projetar e fabricar um processador totalmente reversível utilizando recursos comparáveis a um microprocessador convencional, VIERI (1999).

Observações feitas ao longo dos anos em relação a computação clássica mostram que há alguns gargalos na computação, acreditava-se anteriormente, que não somente cálculos não reversíveis provocavam dissipação de color. A história HAULUND (2017), mostra que este problema é inerente ao modelo da arquitetura de John von Neumann. Este processo acontece no modelo clássico que é irreversível, porque as instruções precisam ser programadas, apagadas ou até mesmo modificadas.

As ferramentas, CAD; THOMSEN. (2012), CAD (do inglês Computer Aided Design), softwares que permitem que engenheiros e projetistas criem modelos realistas de peças e montagens existentes e a tecnologia de silício é suficiente para projetos de computadores reversíveis. O conteúdo da computação reversível é bem abrangente sobre circuitos reversíveis, mas o foco também é nas linguagens de programação reversíveis por exemplo, a linguagem de programação reversível Janus, proposta por Lutz e Derby em 1982 (LUTZ).

A computação clássica é determinística vem de forma avançada, ou seja, cada estado é seguido por um único. A computação clássica que também precisa ser retroativa e determinística: e cada estado tem um estado predecessor único. Portanto, o interesse dos estudiosos em relação ao paradigma reversível na computação convencional é bem mais abrangente do que isso, porque está ligado ao mecanismo de retrocesso onipresente. No entanto, o processo reversível é visto com característica rastreável. Ou seja, todo o percurso seguido de fluido de trabalho, portanto, quando o acontece o movimento de 1 até 2, o inverso pode acontecer no caso 2 até 1 indo no mesmo caminho. Este processo mostra que não existe dissipação de energia dentro do sistema. Na realidade depende muito dos materiais utilizados, porque na prática há perda de calor, portanto, de energia, mas a dissipação é bem menor do que a computação clássica.

O mundo vive um problema energético. O uso de combustíveis poluentes para gerar energia se tornou um grande problema. A matriz ENERGIA.(2020), energética mundial está num processo revolucionário na ampliação da participação das fontes renováveis na produção

de energia. Os computadores PC ou Portátil o ideal para um consumo regular seria importante sempre que for se ausentar por mais de meia hora, vale a pena desligar o computador. Os usuários pensam que o processo de ligar e desligar o computador consome mais energia do que continuar ligado. Isto não é verdade. Quando precisar sair do local de trabalho por mais de meia hora, é interessante desligar o computador. Um computador ligado durante 1 hora/dia consome 5,0 kwh/mês.

1.1 Justificativa

Há um grande esforço para minimização do consumo de energia dos microprocessadores modernos, a ponto de agora ser considerado uma restrição de design de primeira classe. No entanto, existe um limite inferior teórico para o nosso modelo de computação atual. Conhecido desde o início da década de 1960, o princípio de Landauer Landauer.(1961), sustenta que: o apagamento de informações em um sistema é sempre acompanhado por um aumento no consumo de energia. Eles colocam alguns limites no que pode ser calculado neste modelo reversível de computação. Uma razão pela qual as pessoas estão interessadas em computação reversível é por causa de sua eficiência energética teórica

A indústria de semicondutores está rapidamente se aproximando do limite de von Neumann-Landauer. A computação reversível pode ser uma solução viável, mas representa uma mudança significativa de paradigma em relação aos modelos irreversíveis de computação. A praticidade da computação reversível depende, entre outras coisas, da presença de linguagens de programação reversíveis que podem ser compiladas em código de montagem reversível de baixo nível sem sobrecarga significativa. Idealmente, essas linguagens devem fornecer as mesmas ferramentas e recursos para produzir modelos abstratos e interfaces disponíveis para linguagens irreversíveis modernas. A programação orientada a objetos (POO, ou OOP segundo as suas siglas em inglês) é um paradigma imensamente popular na indústria, mas a combinação de OOP e computação reversível são territórios totalmente desconhecidos. O trabalho apresentado nesta monografia é motivado pela escassez de linguagens de programação reversíveis, em particular, pela ausência de qualquer linguagem de programação orientada a objeto reversível.

1.2 Objetivo

O processo é reproduzir os resultados da Teoria da Computação Clássica (reversível) que relacionam as funções recursivas com LPs minimais, usando agora as correspondentes

reversíveis. Além disso, um dos nossos objetivos é comparar os estilos de programação de modelo computacional existentes que suporta o combinado com aqueles disponíveis dentro de máquinas de Turing reversíveis como em outras linguagens de programação reversíveis. Outra questão interessante é encontrar uma notação mais amigável para funções que têm várias saídas e, portanto, não se prestam facilmente a uma representação linear para composição.

1.3 Objetivos Específicos

- 1. Desenvolvemos um estudo do estado da arte sobre a Teoria da Computação Clássica (reversível) que relaciona funções recursivas com LPs minimais.
- 2. Entender o processo de entropia no processo de apagamento de bit que pode estar contida em qualquer sistema, em função do volume físico do sistema e da quantidade de energia que ele contém.
- 3. Apresenta e compara as linguagens de programação de alto nível Janus, a linguagem de programação reversível R e o microprocessador Pendulum ISA (PISA).

1.4 Organização do trabalho

Nesta seção apresenta a forma de organização do trabalho. A seção 1, apresenta a introdução sobre a problemática do tema, a sua justificativa e os objetivos do trabalho. Na próxima, a seção 2 apresenta o referencial teórico sobre este a monografia. A seção 3, apresenta os trabalhos relacionados com a problemática a respeito do tema proposto. A seção 4, detalha-se toda a matérias e métodos utilizada no experimento. Na seção 5, são apresentados os procedimentos a relacionados à metodologia aplicada. Na seção 6, os resultados da discussão do trabalho. Na seção 7, comparações dos trabalhos e analises.

Por fim, na 8, seção é apresenta a conclusão do estudo e são listados os possíveis trabalhos futuros.

2. Trabalhos Relacionados

2.1 Os circuitos lógicos da porta de Toffoli

Em TOFFOLI.(1980), a teoria da computação reversível é baseada em primitivas invertíveis e regras de composição que preservam a invertibilidade. Com essas restrições, ainda é possível lidar satisfatoriamente com os aspectos funcionais e estruturais dos processos computacionais; ao mesmo tempo, alcança-se uma correspondência mais estreita entre o comportamento dos sistemas computacionais abstratos e as leis da física microscópicas (que se presume serem estritamente reversíveis) subjacentes a qualquer implementação concreta de tais sistemas. De acordo com uma interpretação física, o resultado central é que *é idealmente possível construir circuitos sequenciais com dissipação de potência interna zero*.

2.2 Informações logicamente reversíveis como apagamento de bit

Em Landauer . (1961), diz na demonstração que são apenas as operações logicamente irreversíveis em um computador físico que necessariamente dissipam energia gerando uma quantidade correspondente de entropia para cada pedaço de informação que é irreversivelmente apagado; as operações logicamente reversíveis podem, em princípio, ser realizadas sem dissipação. No nível básico, no entanto, a matéria é governada pela mecânica clássica e pela mecânica quântica, que são reversíveis".

2.3 Janus uma linguagem de programação imperativa reversível

Em LUTZ Derby. (1986), foi criada a linguagem de programação reversível Janus. A linguagem teve o projeto base desenvolvido em cima da Prolog como linguagem de implementação, isso foi facilmente possível, apenas um subconjunto da linguagem foi implementado. A Janus é uma linguagem procedural com instruções de programa localmente invertíveis de acesso direto à semântica inversa. Existem 3 tipos de dados na Janus: inteiros simples, matrizes de inteiros de tamanho fixo e pilhas de inteiros de tamanho dinâmico. Variáveis inteiras e pilhas inteiras podem ser declaradas localmente ou estaticamente no escopo global, enquanto matriz de inteiros só podem ser declarados estaticamente.

2.4 A linguagem de programação reversível R

Em Frank.(1997), foi colocado a base para este projeto linguagem de programação reversível R e a arquitetura de processador reversível Pendulum, ambas originalmente desenvolvidas no Massachusetts Institute of Technology (MIT). Também foi desenvolvido um compilador para R que tem como alvo a arquitetura Pendulum. “Matt DeBergalis (MIT)” desenvolveu um simulador para a linguagem montagem Pendulum, conhecidos como “PendVM.” Mas também há modificação de memória em R feita pelas instruções de incremento, negação, troca e atualização. Essas instruções operam em localizações de memória que podem ser representadas por identificadores de variável, por expressões referentes a endereços de memória ou por expressões referentes a elementos específicos de uma matriz (com um sublinhado representando a indexação da matriz) .

2.5 O microprocessador Pendulum ISA (PISA)

Em VIERI .(1995), foi criada no Massachusetts Institute of Technology (MIT). A arquitetura Pendulum assemelha-se a uma mistura de PDP-8 e RISC e foi o primeiro processador programável reversível e conjunto de instruções. A PISA é uma linguagem de montagem do tipo MIPS que passou por várias encarnações. A versão apresentada nesta seção é conhecida como PISA montagem Language (PAL) e é compatível com a máquina virtual Pendulum.

3. Referencial teórico

A presente pesquisa possibilitará a compreensão do entendimento do problema fazendo observações no referencial teórico embasando-o, por meio de uma explicação sobre a computação reversível. O fornecimento de energia térmica para o material, potencializa o aumento da temperatura. Portanto sua temperatura deveria aumentar continuamente enquanto houver fornecimento de energia térmica para o objeto. A matéria é tudo que tem massa e também ocupa lugar no espaço. R. Landauer JANUS. (2020), CHOUDHURY .(2018), provou no artigo que são os processos logicamente irreversíveis em um computador que potencialmente dissipam energia produzindo uma quantidade correspondente de entropia para cada bit de informação que é irreversivelmente apagada. As operações logicamente reversíveis podem, em princípio, ser realizadas sem dissipação.

No entendimento das transformações reversíveis é quase tudo que pode voltar ao estado anterior da transformação. Já que no caso das transformações irreversíveis o processo de retorno não vai ocorrer. Landauer em princípio acreditou que a irreversibilidade lógica era necessária para a computação e, portanto, a computação reversível era impossível, mas Bennett provou de forma convincente do contrário. É importante atentar que não é só os cálculos não reversíveis que implicam em dissipação de calor Bennett1. (1973). Muitos dizem que este problema de dissipação de energia poderá ter influências negativas no futuro da computação porque a ideia é diminuir o consumo de energia LI_Ming_VITÁNYI .(1996). A ideia é a seguinte: com o processo e o avanço da computação fisicamente reversíveis poderá diminuir drasticamente a energia que atualmente se gasta na computação.

A dissipação VIERI (1999), de energia em microprocessadores modernos está rapidamente se tornando uma preocupação primordial do projeto. Microprocessadores contendo alguns milhões de transistores e dissipando dezenas de watts são comuns, limitando sua utilidade em aplicações portáteis e dificultando a remoção de calor em estruturas densas. A literatura apresenta o “Pendulum”, uma arquitetura de computador logicamente reversível que pode operar sem dissipar energia. O novo aspecto do processador reversível Pendulum é que toda a computação é reversível.

– Portanto, podemos definir que o PROCESSO IRREVERSÍVEL é aquele em que um sistema depois de atingido o estado final de equilíbrio, não retorna ao estado anterior ou até mesmo a quaisquer transformações sem ação de agentes externos

– Portanto, podemos definir que o PROCESSO REVERSÍVEL é aquele em tem o estado de ida e volta em ambos movimentos passando por todas as etapas, sem que isso provoque modificações permanentes ao meio externo.

3.1 As linguagens LOOP(N) e SRL(Z)

Caracterização das funções recursivas primitivas por uma linguagem de registradores. Uma subclasse bem conhecida e muito importante de funções recursivas (totais) é a classe de funções “recursivas primitivas” MATOS. (2014). Conforme demonstrado por AR Meyer e DM Ritchie, essa classe de funções também pode ser caracterizada por uma linguagem de registradores específica (não reversível), chamada “LOOP”. Isso significa que todo programa escrito em LOOP (junto com a especificação dos registradores de entrada e saída) define uma

função recursiva primitiva e, reciprocamente, que todo recursivo primitivo é calculado por algum programa em LOOP.

A linguagem SRL e as transformações reversíveis. Baseado no trabalho de MATOS2. (2020), detalha a características da linguagem de registradores e reversível denominada de SRL (Simple Reversible Language), que determina um conjunto de transformações (bijeções) e de tuplas. A linguagem LOOP identifica uma subclasse de programas que existem dentro das linguagens de programação WHILE e que correspondem à classe de funções recursivas primitivas, crucial na teoria da recursão. A diferença distintiva entre linguagens SRL e LOOP, ou WHILE, é que os registradores de SRL inteiros positivos e negativos (como linguagens de programação padrão) .

Na tabela 1 abaixo faremos uma breve comparação entre as linguagens SRL com a linguagem Loop (que caracteriza a classe de funções recursivas primitivas). Na linguagem Loop, após a execução da instrução “for”, a variável loop é setada em 0.

Linguagem:	Laço	SRL	ESRL
Total	Sim	Sim	Sim
Reversível	Não	Sim	Sim
Registradores de conteúdo	Um inteiro não negativo	Um inteiro (possivelmente negativo)	Um inteiro (possivelmente negativo)
Entrada	Uma tupla de registradores	uMa tupla de todos os registradores	Uma tupla de todos os registradores
Resultado	Um registrador	Uma tupla de todos os registradores	Uma tupla de todos os registradores
Instruções	inc, dec, for (*)	inc, dec, for	inc, dec, for swap

Tabela 1 - Na tabela acima faremos uma breve comparação entre as linguagens SRL com a linguagem LOOP. Fonte: MATOS. (2014)

Para cada programa P de SRL, podemos construir o programa P^{-1} que reverte o comportamento de P de forma efetiva. Ou seja, executar P^{-1} logo após P é equivalente à identidade. Obviamente, incremento e decremento são inversos mútuos. Por outro lado, para $R(P)$ itera n vezes a execução de P, sempre $que\ n \geq 0$, itera n vezes a execução da inversa de P sempre $que\ n \leq 0$; então, ele pode ser usado para se inverter. Apesar do conjunto de instruções de SRL ser bastante limitado, sua semântica operacional é inesperadamente complexa. Lendo os textos sobre linguagens reversíveis ou modelos de computação por exemplo, em que existe um algoritmo que tem a função de inverter cada programa (ou autômato circuitos) P, a ideia é criar o programa inverso P^{-1} que satisfaz

$$P;\ P^{-1} \equiv P^{-1};\ P \equiv \varepsilon$$

Onde ε define o programa de identidade, e " ≡ " define equivalência de programa, e ";" define a operação de concatenação (sequenciamento de programa), e "ε" Como definição do programa "ε" é total, todo programa P também deve ser total. O inverso de um programa deve ser muito fácil de definir e deve usar exatamente os mesmos registradores; nenhuma memória adicional deve ser necessária para executar o programa na direção inversa; isso não acontece com algumas "reversibilizações" do modelo computacional, como, por exemplo, em Bennett1. (1973).

Definição Dois programas P e Q são equivalentes, e escrevemos $P \equiv Q$, se eles induzem a mesma permutação $f: Z^{\infty} \rightarrow Z^{\infty}$, isto é, se $[[P]] = [[Q]]$. O programa vazio é denotado por ε, enquanto a permutação identidade $\mathrm{Z}^{\infty} \rightarrow \mathrm{Z}^{\infty}$ são denotadas por ι. Aqui definimos simultaneamente a sintaxe, o inverso e o significado de um programa.

- Instrução inc x. O inverso é $\mathrm{dec\ x}^{-1}$ = dec x. O significado é: incrementa em 1 o valor contido em x.
- Instrução dec x. O inverso é $\mathrm{dec\ x}^{-1}$ = inc x. O significado é: decrementar em 1 o valor contido em x.
- *Instrução "for x(P)" onde P nunca modifica x. O valor de x não é alterado pela execução desta instrução. O inverso é* $(for\ x(P))^{-1}$ *= for x* (P^{-1})*. O significado é: executar x vezes*

o programa P; se x for negativo, isso deve ser interpretado como "execute -x vezes o programa (P^{-1})*.*

Troca de instruções $(x;\ y)$. O inverso é $(swap(x;\ y))^1 = swap(x;\ y)$ (a mesma instrução). O significado é: trocar os valores armazenados nos registradores x e y.

$y = x\ mod\ m\ para\ m \geq 2\ fixo$ MATOS2. (2014).

Suponha que y, z e w tenham valores iniciais $0, 0\ e\ 1$, respectivamente. Considere o programa $P = swap(z\ ;\ w);\ swap(y;\ z)$ com saída $y + 2z$; executando x vezes, obtemos sucessivamente

X	y	z	w	y + 2z
0	0	0	1	0
1	1	0	0	1
2	0	1	0	2
3	0	0	1	0
4	1	0	0	1
5	0	1	0	2
6	0	0	1	0

Vemos que o programa $for\ x(P)$ implementa a função $y = x\ mod\ 3$. Isso é facilmente gerado para qualquer $m \geq 2$.

Propriedade 1 Seja P qualquer programa em SRL e seja P^{-1} seu inverso formal (ou sintático), conforme mostrado acima. Então MATOS2. (2020),

$$[[P;\ P^{-1}]] = [[P^{-1};\ P]] = \text{ł}$$

Prova. Use indução estrutural em P.

Transformações reversíveis: definição indutiva. Da mesma forma que acontece com as funções recursivas primitivas, também é possível definir indutivamente a classe de transformações SRL, sem mencionar nenhuma linguagem de programação em particular, como (aproximadamente) o menor conjunto de transformações que inclui as seguintes operações

1) incrementar um registrador em 1;

2) decrementar um registrador em 1;

3) compor duas transformações SRL;

4) composição paramétrica, em que o valor de uma variável x determina o número de vezes que uma determinada transformação SRL P é composta consigo mesma,

dizer $\overbrace{P; P; \ldots P;}^{v[x]\ P\text{'s}}$ onde $v[x]$ é o valor contido no registrador x. A transformação P não pode mencionar x. Como $v[x]$ pode ser um inteiro negativo, damos uma definição apropriada para “execute $v[x]$ vezes a transformação P" quando $v[x] < 0$.

Deve-se enfatizar que a definição indutiva e a definição baseada na linguagem são de fato bastante semelhantes. E isso é verdade para ambos os conceitos: funções recursivas primitivas e transformações SRL reversíveis. Por exemplo MATOS2.(2020), calcular o valor de uma função $f(x; y)$ usando a definição de recursão primitiva , é essencialmente idêntico à execução do programa a Loop com registradores de entrada x e y.

No entanto, pode acontecer que $[[P;\ Q]] = \wr$ e Q, não é o inverso formal de P.

Por exemplo, P = “inc x; inc y; dec x" e Q = “dec y".

3.2 A linguagem de programação reversível Janus

Janus é uma linguagem procedural com instruções de programa localmente invertíveis e acesso direto à semântica inversa. Existem 3 tipos de dados no Janus: inteiros simples, matrizes de inteiros de tamanho fixo e pilhas de inteiros de tamanho dinâmico. Variáveis inteiras e pilhas inteiras podem ser declaradas localmente ou estaticamente no escopo global, enquanto matrizes inteiras só podem ser declaradas estaticamente. Um programa Janus consiste em um procedimento principal seguido por qualquer número de procedimentos secundários. O procedimento principal atua como o ponto de partida do programa e é precedido por comandos de variáveis estáticas, que servem como saída do programa após o término.

```
[n, n_1 n_2] armazenamentos globais (três variáveis inteiras, inicialmente zero)
        procedure   fib
        if n=0   them     x_1 +=1                    reversível
        teste                x_2+=1                  procedimento
                          else n -= 1
                          call      fib
                          x_1 +=x_2
                          x_1 ⇔ x_2          valores de troca de   x_1, x_2
        fi  [x_1 = x_2]
```

Figura 1 - Fonte: (SIMULADOR_JANUS)

```
/* Baseado em David Gries: A ciência da programação, 1987. pp.270 */ /*
Permutação para código * Dado um matriz X que contém uma permutação dos
inteiros 0, 1, ..., * tamanho(X)-1, ele calcula o matriz X' onde X'[i] é o número de X[j <
i] que * são menores que X[i].*/
                procedure perm_to_code(int x[])
                        local int k = size(x)
                        from k = size(x) loop
                                        k -= 1
                                        local int j = 0
                                from j = 0 loop
                                        if x[j] > x[k] then
                                        x[j] -= 1
                                        fi x[j] >= x[k]
                                        j += 1
                                until j = k
                                delocal int j = k
                until k = 0
                delocal int k = 0
        procedure main()
                int x[6]
                x[0] += 2
                x[1] += 0
                x[2] += 3
                x[3] += 1
                x[4] += 5
                x[5] += 4
                show(x)
                call perm_to_code(x)
```

Figura 2 - O código gerado no compilador Janus online, código normal.

Fonte: (SIMULADOR_JANUS)

```
procedure perm_to_code(int x[])
                local int k = 0
                 from k = 0 loop
        local int j = k
        from j = k loop
                j -= 1
                if x[j] >= x[k] then
                 x[j] += 1
```

```
                fi x[j] > x[k]
        until j = 0
        delocal int j = 0
        k += 1
        until k = size(x)
         delocal int k = size(x)

procedure main()
                int x[6 ]
                x[0] += 2
                x[1] += 0
                 x[2] += 3
                 x[3] += 1
                x[4] += 5
                 x[5] += 4
                 show(x)

                 call perm_to_code(x)
```

Figura - 3 O houver a inversão do programa atualmente escrito na área de texto.

Fonte: (SIMULADOR_JANUS)

- Tipos escalares e de matrizes, valores inteiros
- Operadores de controle estruturado (IF, LOOP)
- Procedimentos simples (correspondem a LOOPS)
- – Sem valor de retorno, efeitos colaterais no armazenamento global

```
/* Permutation to code
 * Given an array X that contains a permutation of the integers 0, 1, ...,
 * size(X)-1, it computes the array X' where X'[i] is the number of X[j < i]
 * are smaller than X[i].
 */
procedure perm_to_code(int x[])
    local int k = size(x)
    from k = size(x) loop
        k -= 1
        local int j = 0
        from j = 0 loop
            if x[j] > x[k] then
                x[j] -= 1
            fi x[j] >= x[k]
            j += 1
        until j = k
        delocal int j = k
    until k = 0
    delocal int k = 0

procedure main()
    int x[6]

    x[0] += 2
    x[1] += 0
    x[2] += 3
    x[3] += 1
    x[4] += 5
    x[5] += 4
    show(x)
    call perm_to_code(x)
```

Figura - 4 Editado do compilador Janus. Fonte: (SIMULADOR_JANUS)

3.3 A linguagem de programação reversível R.

A linguagem imperativa reversível desenvolvida no Massachusetts Institute of Technology (MIT) CHOUDHURY (2018). R é como C, recurso notável ausente do R é o suporte a operações de dados não inteiros e de ponto flutuante. Como a arquitetura atual do Pendulum não contém hardware de ponto flutuante, as operações aritméticas de ponto flutuante teriam que ser feitas em software. Uma abordagem possível para desenvolver uma biblioteca de ponto flutuante para R seria traduzir uma biblioteca de uma implementação C de código aberto. A linguagem suporta matrizes simples do tipo C, laços for, instruções "if" e sub-rotinas recursivas com argumentos. A reversibilidade da execução de programas R é garantida pela reversibilidade da versão assumida do conjunto de instruções Pendulum, desde

que o programa não use a instrução em linguagem “montagem” EMIT (que explicitamente permite que as informações sejam removidas irrecuperavelmente do processador).
Exemplo de programa R para calcular o enésimo par de Fibonacci.

```
(defsub fib (x1 x2 n)
                (if (n = 0) then
                    (x1 += 1)
                    (x2 += 1) )
                (if (n != 0) then
                    (n -= 1)
                (call fib x1 x2 n)
                (x1 += x2)
                (x1 <-> x2)
        (n += 1))); restaurar valor de n para condicional
        (defword x1 0)
        (defword x2 0)
        (defword n 4)
        defmain fibprog (call fib x1 x2 n))
```

Exemplo de programa R para calcular o enésimo par de Fibonacci, adaptado do programa de exemplo.

Figura 5 - O código da linguagem de programação reversível R.

Fonte: CLARK. (2001), HAULUND (2017).

3.4 PISA

O microprocessador Pendulum (PISA) é um pouco complexa faremos a descrição detalhada do conjunto de instruções em linguagem de montagem para o microprocessador Pendulum reversível CHOUDHURY.(2018). Em um processador convencional, as regras que governam o fluxo de controle são bastante simples: após cada instrução, adicione 1 ao contador de programa. No caso de um salto, é definido o contador de programa para o

endereço do rótulo onde salta. Em um processador reversível como o Pendulum, essas regras são muito mais complicadas, pois simplesmente sobrescrever o conteúdo do contador do programa constituiria uma perda de informação que quebra a reversibilidade. O conjunto de instruções do PISA pode ser dividido em três categorias: operações aritméticas/lógicas reversíveis, instruções de desvio comuns e instruções especiais. O processador Pendulum usa três registradores de propósito especial para lógica de fluxo de controle:

1. O contador de programa (PC) armazena o endereço da instrução atual 2.
2. O registrador de ramificação (BR) para armazenar deslocamentos de salto 3.
3. O bit de direção (DIR) para acompanhar a direção de execução

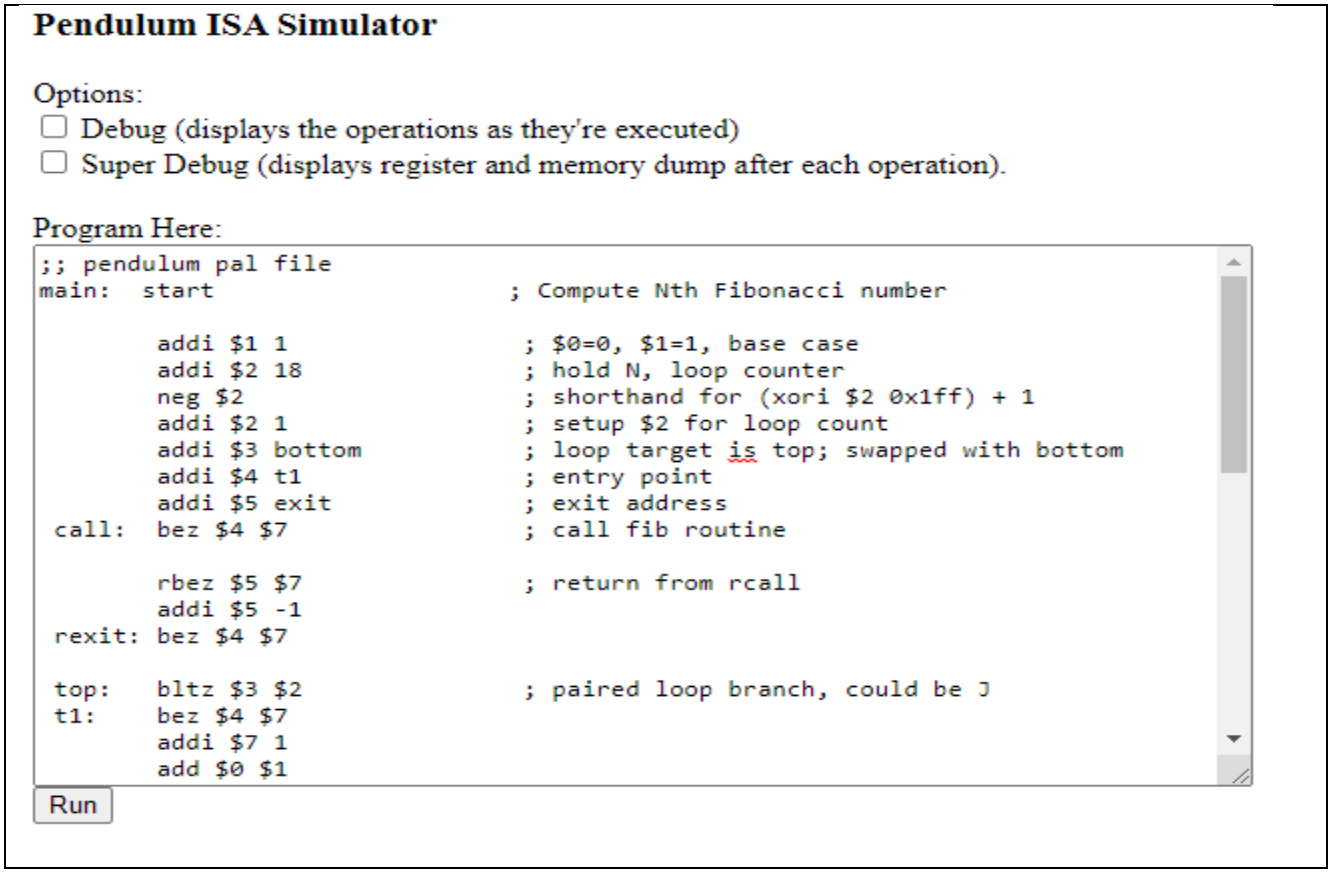


Figura 6 - Simulador do pendulum ISA. Fonte: (SIMULADOR_ISA)

```
;; Call Fall:
10: ADDI $4 1000; h += 1000
11: ADDI $5 5         ; tend += 5
12: BRA 16            ; call
;; Uncall Fall:
8: ADDI $6 40         ; v += 40
19: ADDI $7 4
20: ADDI $5 4
21: RBRA 7
;; Sub routine  Fall:
27: BRA 9
28: SWAPBR $8      ; br <=> rtn
29: NEG $8                  ;rtn=-rtn
30: BGTZ $7 5         ; t > 0?
31: ADDI $6 10          ;v += 10
32: ADDI $4 5         ; h += 5
33: SUB $4 $6         ; h -= v
34: ADDI $7 1         ; t += 1
35: BNE $7 $5 -5   ; t != tend?
```

Figura 7 - O código do simulador do pendulum ISA (PISA). Fonte: (SIMULADOR_ISA)

3.5 As funções computáveis FOR são recursivas primitivas

As funções computáveis FOR são recursivas primitivas correspondem a uma categoria de funções conhecidas como recursivas primitivas. Uma função é recursiva primitiva se for (FOR- computáveis). As funções computáveis FOR são recursivas primitivas aparentam ser relativamente simples para gerar números grandes e listas. As funções computáveis FOR são

recursivas primitivas contém apenas comandos muito básicos, mas eles podem ser combinados para implementar um grande número de algoritmos, KARPER.(2014). A impressão que fica é que qualquer computação pode ser definida dessa forma, no entanto, infelizmente, esse não é o caso. Primeiramente, a interpretação do programa, MATOS2. (2020).

1. Uma função sucessor S^i $(x_1,...., x_n) = x_1 + 1$
2. Uma função projeção $P^i(x_1,..., x_n) = x_1$
3. Na forma de recursão primitiva:

$$f(n, x_1, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_n), & \text{if } n = 0 \\ h(f(n-1, x_1, \ldots, x_n), x_1, \ldots, x_n), & \text{else} \end{cases}$$

Figura 8 - função FOR onde g e h são recursivos primitivos. Fonte: KARPER. (2014).

Dar para perceber que a funções computáveis FOR são recursivas primitivas conseguem computar qualquer função primitiva. A função sucessora e as projeções são bem triviais, para rodar uma função primitiva recursiva, temos os programas [G] = g e [H] = h pela hipótese de indução.

Código em a linguagem recursiva FOR computáveis.

```
F read NX {
        N:= hd NX
        Xs:= tl NX
        Repetitions:= [unary](N)
        Result:= [G](Xs)
        Counter:= 0
                for I in Repetitions {
                Result:= [H](Result.Xs)
                }
} write Result
```

Figura 9 - O código da função FOR primitivo recursiva. Fonte: (KARPER. (2014).

4. Materiais e Métodos

4.1. Conjunto de dados

. Os materiais pesquisados são artigos científicos relacionados a computação reversível, a qual é uma área de pesquisa caracterizada por possuir apenas modelos computacionais que são determinísticos tanto para frente quanto para trás e, simuladores para computação reversíveis.

4.2 Dissipação de Energia

O que determina a temperatura de um material é a quantidade de energia térmica que o material possui. A chama de uma vela é uma fonte contínua de energia térmica enquanto a vela estiver acesa. Então, se colocamos um objeto próximo à chama ou na chama da vela, este objeto está recebendo energia térmica continuamente. É por isso que a barra de ferro colocada na chama da vela não se aquece continuamente, pois a energia térmica se dissipa pelo ambiente que possui uma temperatura menor que o ferro aquecido. A reversibilidade requer que cada estado de um sistema de computação determine o próximo estado e o estado anterior.

A conexão entre a entropia no sentido da ciência da informação e a entropia no sentido da termodinâmica é demonstrada e foi discutida por Szilard SZILARD .(1964). Isso implica que nenhuma informação pode ser apagada na transição de um estado para o outro. Há muitas vantagens em termos computacionais, dentre elas, a eliminação da necessidade de sobrescrever informações. Observe que para cada vez que um dado é sobrescrito, esse apagamento é dissipado como forma de calor e, por este motivo, a computação clássica tende a gerar muito mais calor do que a computação reversível a termodinâmica, Termodinâmica; Landauer. (1961), mostrou que o apagamento de bit requer dissipação de K_B^T em 2 unidades de energia, onde K_B é a constante de Boltzmann.

Os limites máximos da dissipação de energia pela computação serão expressos em número de bits apagados irreversivelmente. Por isso, consideramos a compactação de registradores. O coletor de lixo, o custo é menor se compactamos o lixo antes de jogá-lo fora. A compactação final que pode ser efetivamente explorada é expressa em termos de complexidade de “Kolmogorov” CHOUDHURY.(2018). Seja $l(p)$ o comprimento da string binária p. Seja S o número de elementos do conjunto S. Damos algumas definições e propriedades básicas da complexidade de Kolmogorov. Para todos os detalhes e atribuições nos referimos a LI_Ming_VITÁNYI. (1997). Lá também se encontram as noções básicas da

teoria da computabilidade e das máquinas de Turing. A propriedade “simetria de informação” . Ele refina uma versão anterior em ZVONKIN.(1970), relacionada à complexidade original de Kolmogorov.

Definição que uma função de valor real $f(x,y)$ sobre strings ou números naturais x,y é semicomputável superior se o conjunto de triplos

$$\{\,(x,\,y,\,d) : f(x,\,y) < d,\, com\ d\ racional\,\}$$

é recursivamente enumerável. Uma função f é semicomputável inferior se $-f$ é semicomputável superior.

As funções (x,y,z), etc. são definidas com a ajuda de x,y em qualquer uma das formas usuais. Introduzimos a notação

$$K(x,y) \;=\; K(x,y), K(x|y,z) \;=\; K(x|y,z),$$

etc. A complexidade de Kolmogorov tem a seguinte propriedade de adição:

$$K(x,y) \mathrel{+}= \; K(x) \;+\; K(y|x,K(x)).$$

Assim, estendemos nossa definição inicial de complexidade de Kolmogorov considerando programas de computador com uma interface de entrada-saída muito simples: os programas recebem um fluxo de bits que, durante a execução, podem ler um bit de cada vez. Não há marcadores finais no fluxo de bits, de modo que, se um programa p parar na entrada y e produzir x, ele também irá parar em qualquer entrada yz, onde z é uma continuação de y, e ainda produzirá x. Escrevemos $p(y) = x$ se, na entrada y, p imprimir x e depois parar.

Os computadores fizeram melhorias constantes e dramáticas, tudo graças à Lei de Moore - o aumento exponencial ao longo do tempo no número de transistores que podem ser fabricados em um circuito integrado de um determinado tamanho. A Lei de Moore deveu seu sucesso ao fato de que, à medida que os transistores foram diminuindo, eles se tornaram simultaneamente mais baratos, mais rápidos e mais eficientes em termos de energiaA lei de Moore para dimensões L, W e t tensão limite Vt dissipação de calor Q, Reversible. (2022).

Tecnologia (µm)	L (µm)	W (µm)	t (µm)	V_t (V)	Q (fJ)
2.4	2.4	2.4	42.5	0.9	38
0.8	0.8	2.0	15.5	0.75	2.0
0.35	0.35	0.5	7.4	0.6	0.30

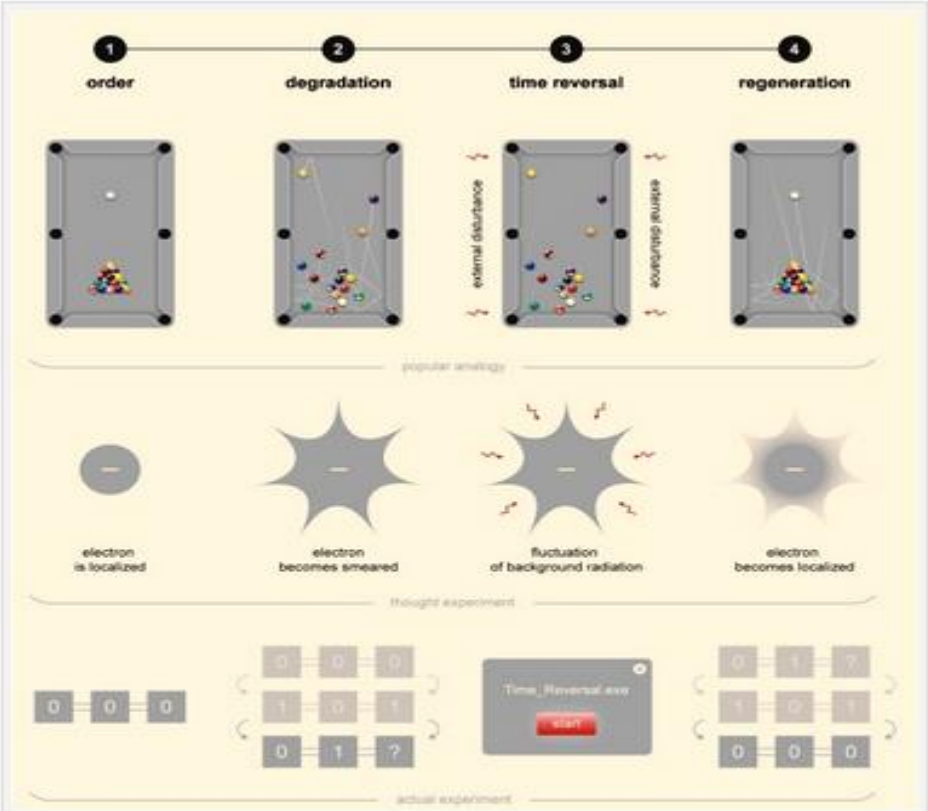
Dissipação de energia por etapa computacional:

$$Q \approx CV_t^2,$$

Onde

$$C \approx \epsilon_0 \epsilon \frac{W L}{t}$$

Comparamos com o quantum de Landauer. (1961), $kT\ log(2)\ \approx 3\ z$J = 0,000 003 $f\ J$.



Os quatro estágios do experimento real em um computador quântico espelham os estágios do experimento mental envolvendo um elétron no espaço e a analogia imaginária com as bolas de bilhar. Cada um dos três sistemas inicialmente evolui da ordem para o caos, mas então uma perturbação externa perfeitamente cronometrada reverte esse processo. – Imagem: tsarcyanide/MIPT

Estudo realizado por uma equipe multinacional faz volta no tempo usando computador quântico **BILHAR**. (2022).

4.3 Fundamentos de uma linguagem reversível

De acordo com Yokohama. (2008), se destacam os fundamentos de uma linguagem reversível por meio de atualizações reversíveis, de determinismo reverso, tendo evidências do processo inverso e a sua eficiência reversível. Com isto temos o interesse em linguagens de programação onde cada passo de execução é reversível. Procuramos observar as propriedades fundamentais que todas as linguagens reversíveis devem satisfazer inerentemente, independentemente do paradigma da linguagem (como funcional ou imperativo) e do nível da linguagem (código de máquina, alto nível). Por uma questão de concretude e para facilitar a comparação entre, as terminologias de programação de estilo imperativo em todo o artigo. Uma solução trivial para tornar um programa reversível é criar um rastro de sua execução. Isso é conhecido como incorporação de Landauer.(1961) e é um exemplo de adição de dados de lixo a um programa para garantir a reversibilidade.

4.4 Atualizações reversíveis

É fato que as funções presentes nas linguagens de programação precisam ser reversíveis a computação atômica reversível, a atualização reversível. Em linguagens reversíveis, tais etapas de computação devem ser compostas apenas de acordo com regras que não prejudiquem a reversibilidade. Circuitos lógicos booleanos, máquinas de acesso aleatório). O apagamento de informações (manifestado, por exemplo, pela incapacidade de recuperar as entradas da saída de nand-gates) torna uma computação irreversível e a maquinaria física de computação em que ela é executada está sujeita ao princípio de Landauer. (1961); a informação apagada deve ser dissipada como calor, que está diretamente relacionado

As condições para uma atualização reversível de um estado de computação são as seguintes.

Definição 1 (1º argumento de função injetiva): Uma função parcial $\odot:(A \times B) \rightarrow C$, escrita como um operador binário fixo, é injetiva no primeiro argumento se, e somente se, $\forall a, a' \in A, \forall b \in B$, $a \odot b$ e $a' \odot b$ satisfazem à definição abaixo:

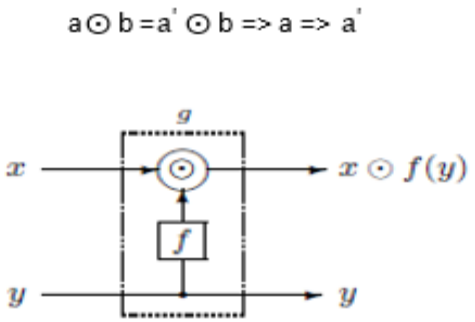


Figura 10 - Atualização logicamente reversível. Fonte: TOFFOLI. (1980)

E para qualquer operador ⊙ injetivo no primeiro argumento, existe um operador ∅:(A × B) →A tal que ∀a ∈ A, ∀b ∈ B:

((a ⊙ b) ∅ b) = a.

Por exemplo, $n + c = m + c$ implica $n = me\,(n + c) - c = n$ para quaisquer inteiros n, m, c. Ou bit a bit exclusivo, xor, tem a propriedade útil de que ((a xor b) xor b) = a. Por outro lado, $n * 0 = m * 0$ não implica $n = m$, e assim o operador de multiplicação * não é injetivo em seu primeiro argumento. Observe que $(a \odot b)\, \emptyset\, b = a$ não implica $(c\, \emptyset\, b) \odot b = c$, então operadores ⊙ e ∅ não são necessariamente trocáveis. Cada operador injetivo também é injetivo em seu primeiro argumento, mas o inverso não é válido

Definição 2 (atualização reversível): Seja uma função f: D → B, um operador ⊙:(A × B) →C, injetivas no primeiro argumento da função parcial g: $(A \times D) \rightarrow (C \times D)$. Uma atualização reversível será equivalente a:

$$g\,(x,) = (x \odot f(\gamma),\ \gamma)$$

A ideia aqui é que a função g deve ser injetiva, no entanto, o artigo diz que a função f não há necessidade de ser injetavas. Ao salvarmos o valor de y, podemos aplicar qualquer função f e recuperarmos quaisquer valores iniciais, finais ou intermediários. Uma atualização reversível g é ilustrada na Figura (10). Logo compreendemos que o processo de atualização reversível se dar por meio do operador inverso ∅, apresentado anteriormente, como:

$$g^{-1}\,(x, y) = (x\, \emptyset\, f(y), y)$$

Com isso nos leva a crer, na grande maioria das vezes, devemos reescrever programas irreversíveis através de atualização reversíveis; portanto fazendo com este seja representado em um programa reversível. Constatou-se ainda que isso significa que normalmente o processo permanece utilizando os mesmos recursos (memória, uso de CPU, espaço e tempo de execução), visto que a função inversa também precisará ser escrita e executada para reverter o processo.

$$g's\ (x,y)\ =\ (x\ +\ f\ (y),y),$$

e seu inverso

$$g^{-1}(x,y)\ =\ (x\ -\ f(y),y).$$

Uma atualização reversível com (XOR) é auto inversa para qualquer f:

$$g's\ (x,y) = g^{-1}(x,y) =\ (x\ xor\ f\ (y),y).$$

4.5 Portas Reversíveis

Modelos de computação convencionais, como máquinas de Turing e máquinas de acesso aleatório (RAMs), destroem informações em cada etapa computacional. Uma porta lógica que implementa uma função não-inversível (não bijetiva) é dita irreversível. Portanto, todas as portas lógicas que possuam mais bits de entrada do que de saída são irreversíveis. Esse processo ocorre nas portas padrões, entre outras, que transformam duas ou mais entradas em uma única saída. Que o uso das portas lógicas irreversíveis necessariamente produz dissipação de calor TOFFOLI. (1980), independente da tecnologia utilizada.

Por outro lado Bennett. (2007), mostrou como usar portas lógicas reversíveis para reduzir ou mesmo eliminar a dissipação de calor em um circuito. Isso acontece, por exemplo, com as portas AND, OR e XOR padrões, entre outras, que transformam duas ou mais entradas em uma única saída. Das portas clássicas convencionais de até 2 bits, apenas a identidade I é reversível. Portas irreversíveis podem ser transformadas em reversíveis, com alguns ajustes e incluindo bits extras

1. O número de linhas de entrada é igual a o número de linhas de saída (n × n)
2. Sua função booleana $B^n \rightarrow\ B^n$ é bijetiva

Tabela verdade de três portas lógicas irreversíveis

(a) porta XOR

(b) porta OR

(c) E porta AND.

AB	P
0 0	0
0 1	1
1 0	1
1 1	0

(a)

$P = A \oplus B$

AB	P
0 0	1
0 1	0
1 0	0
1 1	0

(b)

$P = \overline{A + B}$

AB	P
0 0	0
0 1	0
1 0	0
1 1	1

(c)

$P = A\ B$

Tabela verdade de uma porta lógica reversível IWAMA.(2002).

AB	PQ
0 0	0 0
0 1	1 0
1 0	1 1
1 1	0 1

$P = A \oplus B$
$Q = A$

Figure 1: Not gate

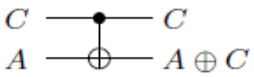


Figure 2: Feynman gate, Feyn.

- A porta Not (Figura. 1); a única porta da lógica convencional que é reversível.
- A *porta Feynman* (Feyn, Figura. 2), ou porta não controlada, nega a entrada *A* se o controle *C* for verdadeiro.

Figura 11 Fonte: TOFFOLI. (1980).

Tabela verdade do somador completo

(a) irreversível

(b) reversível

ABC_{in}	$C_{out}S$
0 0 0	0 0
0 0 1	0 1
0 1 0	0 1
0 1 1	1 0
1 0 0	0 1
1 0 1	1 0
1 1 0	1 0
1 1 1	1 1

(a)

A	B	C_{in}	P	C_{out}	S	G_1	G_2
0	**0**	**0**	0	**0**	**0**	0	0
0	0	0	1	1	0	0	0
0	**0**	**1**	0	**0**	**1**	0	0
0	0	1	1	1	1	0	0
0	**1**	**0**	0	**0**	**1**	0	1
0	1	0	1	1	1	0	1
0	**1**	**1**	0	**1**	**0**	0	1
0	1	1	1	0	0	0	1
1	**0**	**0**	0	**0**	**1**	1	1
1	0	0	1	1	1	1	1
1	**0**	**1**	0	**1**	**0**	1	1
1	0	1	1	0	0	1	1
1	**1**	**0**	0	**1**	**0**	1	0
1	1	0	1	0	0	1	0
1	**1**	**1**	0	**1**	**1**	1	0
1	1	1	1	0	1	1	0

(b)

Assim: um bit de entrada extra: P pré-ajustado
e dois bits de saída extras: garbages $G1$ *e* $G2$

Tabela verdade da função booleana $f(A, B, C)$
(a) irreversível
(b) reversível

ABC	f
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

(a)

A	B	C	P	G_1	G_2	G_3	$f \oplus P$
0	**0**	**0**	0	0	0	0	**0**
0	0	0	1	0	0	0	1
0	**0**	**1**	0	0	0	1	**0**
0	0	1	1	0	0	1	1
0	**1**	**0**	0	0	1	0	**0**
0	1	0	1	0	1	0	1
0	**1**	**1**	0	0	1	1	**1**
0	1	1	1	0	1	1	0
1	**0**	**0**	0	1	0	0	**0**
1	0	0	1	1	0	0	1
1	**0**	**1**	0	1	0	1	**1**
1	0	1	1	1	0	1	0
1	**1**	**0**	0	1	1	0	**1**
1	1	0	1	1	1	0	0
1	**1**	**1**	0	1	1	1	**1**
1	1	1	1	1	1	1	0

(b)

Assim: um bit de entrada extra: P pré-ajustado
e MUITOS bits de saída extras: garbages G_i
1º Argumento injetivo

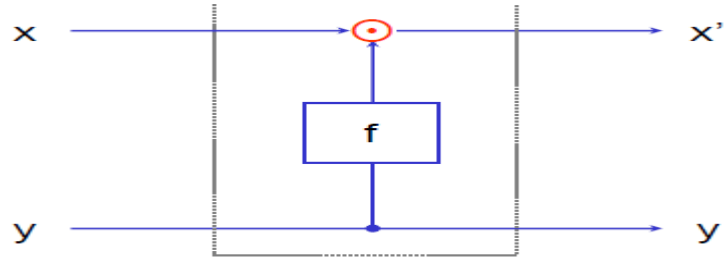
If um operador $\odot$ for injetivo em seu **em seu 1° argumento**,
$a \odot \boldsymbol{c} = b \odot \boldsymbol{c} \Rightarrow a = b$ (if $a \odot \boldsymbol{c}$ *e* $b \odot \boldsymbol{c}$ definido),
então existe um operador $\odot$ existe para reconstruir o 1º argumento:
$(a \odot \boldsymbol{c}) \odot \boldsymbol{c} = a$

Exemplo:
$n + \mathbf{3} = m + \mathbf{3} => n = m\, e\, (n + \mathbf{3}) - \mathbf{3} = n$
Não é Exemplo**:**
$n * \mathbf{0} = m * \mathbf{0} \neq> n = m$

Dado (⊙, ⊖) e uma função parcial f, então a função
$g\ (x, y) =^{def}\ (x \odot \boldsymbol{f}(\boldsymbol{y}), y)$
é uma atualização reversível de x, e existe
$g^{-1}(x, y) =^{\text{def}}$ (x ⊖ $\boldsymbol{f}$ **(y)**, y)

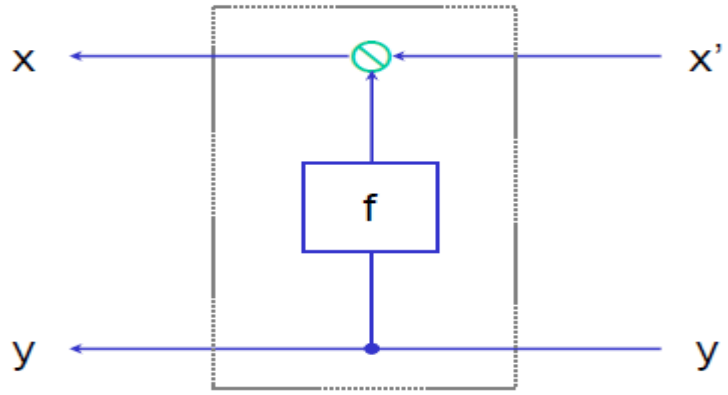
Exemplo:
$g\ (x, y) = (x + \boldsymbol{f}(\boldsymbol{y}), y)$ … atualização reversível de x
$g'^{S}(x, y) = (x - \boldsymbol{f}(\boldsymbol{y}), y)$ … função inversa

Frente



⊙ ...injetivo no argumento x

Para trás



⊖... injetivo no argumento x

Adição é irreversível
Por exemplo **2 + 3 → 5**
1 + 4 → 5
? + ? ← 5
Solução:

$$+_n(A, B) \mapsto (A, B + A \bmod 2^n)$$

- Atualizações reversíveis
- Mantém um operante

4.6 Janus

As linguagens clássicas, embora bem didáticas, são apenas determinísticas, tem processo de sentido único (por exemplo, C, Java e Python) e, no entanto, não possuem uma semântica inversa eficiente. A semântica dos programas Janus é especificada pelas regras mostradas na Figura (11). A semântica operacional tem três significados principais: a avaliação de expressões, a execução de instruções e a execução de programas CHOUDHURY.(2018). Definido o significado da expressão σ que é um armazenamento, e a expressão e v um valor. Definimos que no armazenamento a expressão σ, é avaliada como valor v. A avaliação das expressões não causa efeitos colaterais em um armazenamento. Algumas definições são (outras são semelhantes):

$$\sigma \vdash \text{expr e} \Rightarrow v$$

$v \in Vals = Z_{32} \cup Stack\ Z$

$l \in Lvals = \{a, b, \ldots, a[0], a[1], \ldots, b[0], \ldots\}$

$\sigma \in Stores = Lvals \rightharpoonup Vals$

$\Gamma \in Pmaps = PIds \rightharpoonup Procs$

Figura 12 - valores semânticos. Fonte: CHOUDHURY. (2018)

$$\sigma\ \Gamma \vdash \text{stmt s} \Rightarrow \sigma'$$

O significado de uma condicional é definido pelas regras If True e If False, e qual regra se aplica depende do valor.**Figura** (24)

Seja Z_{32} o conjunto de inteiros com sinal de 32 bits. Um valor v é um inteiro $\in Z_{32}$ ou uma pilha de inteiros $\in StacK_Z$. As pilhas de números inteiros são definidas indutivamente por

$$StacK_Z = \{nil\}\ U\ \{hd :: tl \mid hd \in Z_{32}\ \Lambda\ tl \in StacK_Z\}$$

A execução sai do laço se o teste e_2 da **Figura** (14) for verdadeiro seguindo a regra Laço Base, caso contrário o laço continua pela regra Laço Rec. As regras para loops dadas aqui são funcionalmente equivalentes a formalização em PAOLINI.(2016). Uma call de

procedimento executa o corpo do procedimento no armazenamento atual, onde os parâmetros formais x_1 , . . .,x_n que aparecem no corpo são substituídos pelos parâmetros reais y_1, . . ., y_n. Usamos o modo de passagem de parâmetro por referência. A regra Call relaciona um armazenamento de entrada σ com um armazenamento de saída σ´ após a execução do corpo do procedimento.

Por outro lado, um procedimento "uncall" relaciona σ e σ´ com a opção armazenamentos positivos de uma "call": o armazenamento da entrada σ de uma "call" é a saída armazenamento de um "uncall", e vice-versa. Assim, uma "uncall" efetivamente inverte a direção de execução para o corpo do procedimento. Este importante mecanismo de linguagens reversíveis, orientado pelo conceito de troca do armazenamento de entrada e saída para construções inversas, é uma técnica semântica. Usamos a mesma técnica para definir um pop como o inverso de um push "regras Push e Pop".

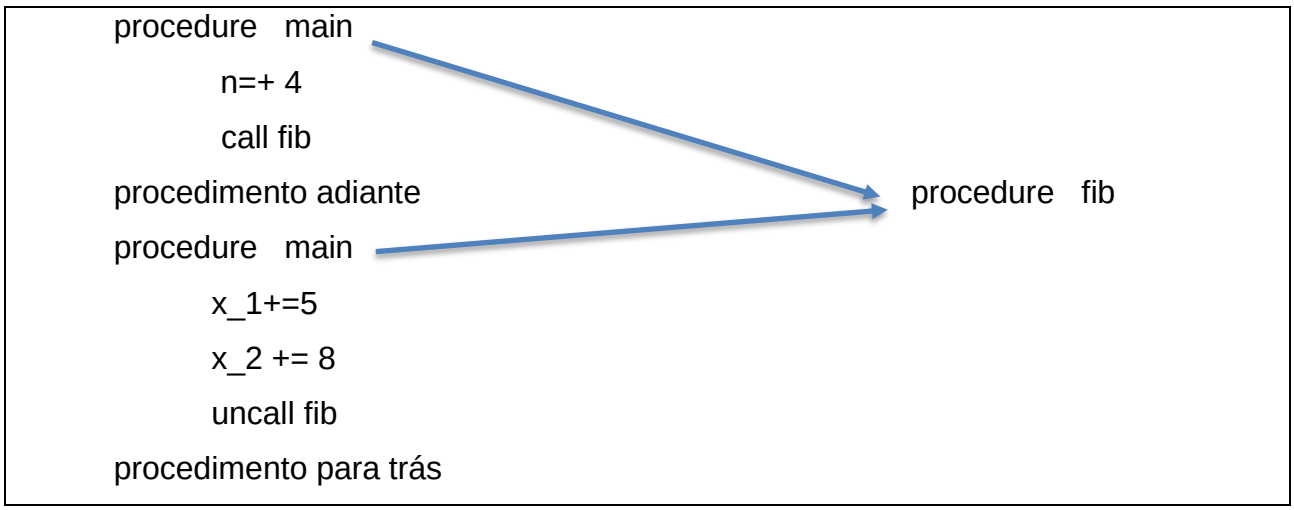


Figura 13- Computação para frente e para trás. Fonte: (SIMULADOR_JANUS,CLARK).

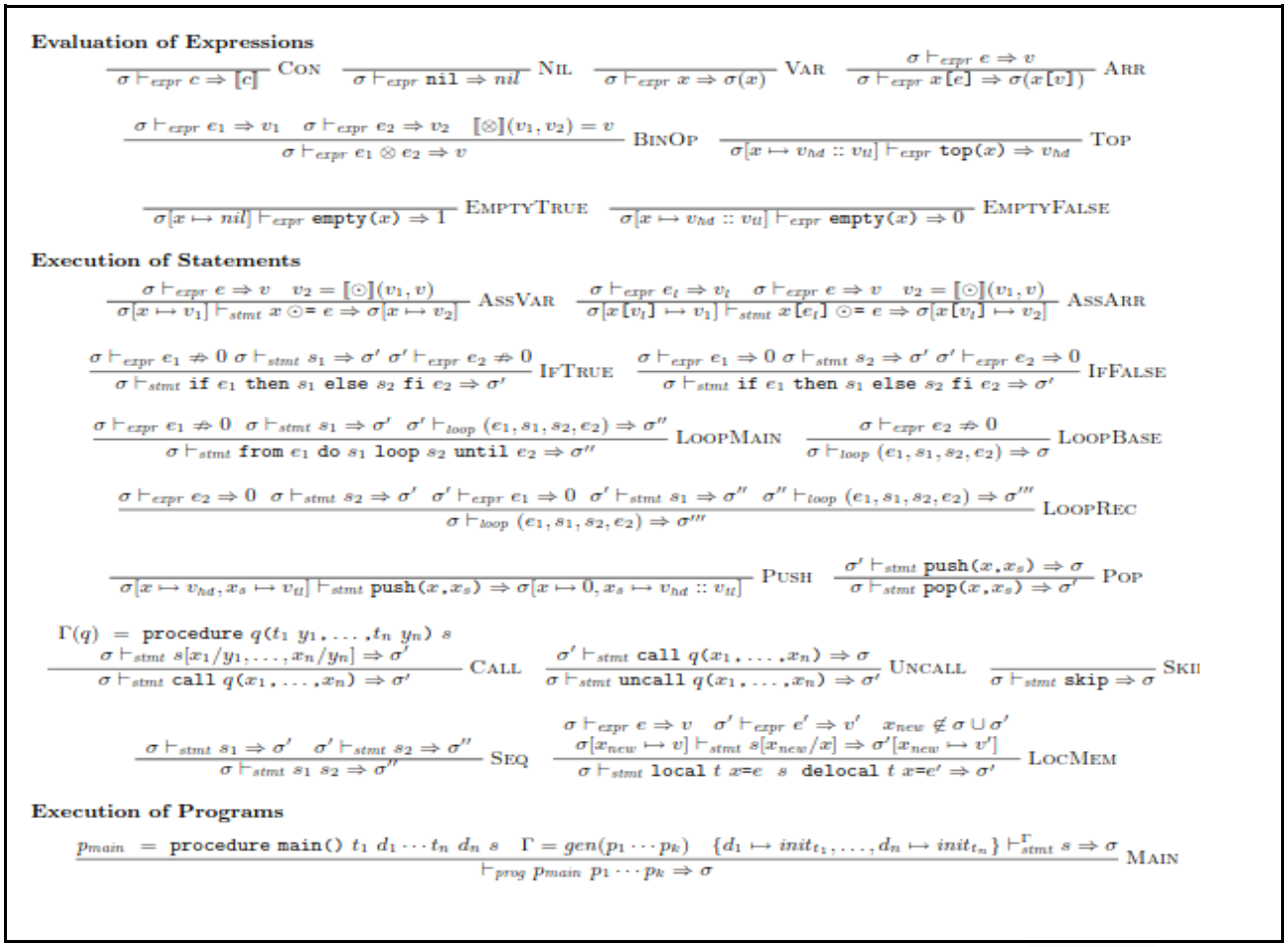


Figura 14- Semântica operacional dos programas Janus. Fonte PAOLINI.(2016)

4.7 A linguagens de registradores Loop e SRL

Cada programa usa apenas um pré-determinado número de registradores, de modo que, usando o modelo, podemos dizer que, independentemente dos valores das entradas, cada programa usa uma quantidade fixa de memória, PAOLINI. (2016), MATOS.(2014). Isso não acontece, por exemplo, com máquinas de Turing, onde o número de células da fita usadas em um cálculo depende em geral da entrada; para fazer cálculos sem ininterrupção, esse número também não pode ser limitado.

5 Metodologia

A metodologia descreve os procedimentos das pesquisas realizadas com o intuito de atingir os objetivos do trabalho, descrevendo como foi o processo de desenvolvimento para se chegar ao embasamento previsto do estudo da computação reversível.

- A linguagem de programação reversível Janus
- A linguagem de programação reversível R

- O microprocessador Pendulum e o Pendulum ISA (PISA)

5.1 Analisador de sintaxe da linguagem JANUS

Definição 1 (A sintaxe de Janus). A gramática que gera expressões, programas e declarações do dialeto de Janus que focamos é:

e::= n | x | e + e | e − e | e ∗ e | e/e | e% e | e <= e | e ! = e | e == e

P::= (procedure id s)(procedure id s)∗

s::= x += e | x −= e | call id | uncall id | skip | s s

| if e then s else s fi e | from e do s loop s until e

Uma expressão "e" é um numeral "n" '(tacitamente confundido com um número natural em N) uma variável x ou a aplicação de um operador binário a sub-expressões.

[Terminais entre aspas ou minúsculas.

{ }* indica zero ou mais repetições. [ ] indica zero ou uma repetição.

```
Program ::=          { ident [ '[' num ']' ] }*
                   { 'PROCEDURE' ident Statements }*
Statements ::=       I fstmt   Statements
                   | Dostmt Statements
                   | Callstmt Statements
                   | Readstmt Statements
                   | Writestmt Statements
                   | Lvalstmt Statements
                   | null
Ifstmt ::=           'IF'        Expression
                   [ 'THEN'   Statements]
                   [ 'ELSE'    Statements]
                   'FI'          Expression
Dostmt ::=           'FROM'    Expression
                   ['DO'      Statements ]
                   ['LOP'     Statements]
                   'UNTIL'    Expression
Callstmt ::=         'CALL'   ident
                   | 'UNCALL' ident
Readstmt ::='        READ' ident
Writestmt ::=        'WRITE' ident
Lvalstmt ::=         Lvalue Modstmt
                   | Lvalue Swapstmt
```

```
Modstmt ::=          '+=' Expression
          | '-=' Expression
          | '!=' Expression
Swapstmt ::=         ':' Lvalue
Expression ::=       Minexp
          | Minexp binop Expression
Minexp ::=           '(' Expression ')'
          | '-' Expression
          | '~' Expression
          | Lvalue
          | Constant
Lvalue ::=           ident
          | ident '[' Expression ']'
Constant ::=         num
```

Figura 15 - Representa a gramática da linguagem Janus. Fonte: (SIMULADOR_JANUS).

Para cada (ident ['[' num ']']) encontrado, o nó a raiz do analisador de (procedimento parseprog) cria uma instância da classe (dotaarray), que contém uma matriz para os resultados da correlação do armazenamento PAOLINI. (2016). Uma condição de erro pode ser resolvida revertendo a direção de execução do programa no ponto da tentativa de operação singular, mas como as condições de erro são consideradas anormais e normalmente não ocorrem em programas “em funcionamento”, o sistema de tempo de execução responde simplesmente interromper a execução do programa. Isso permite examinar o estado no momento da condição de erro. Subscritos fora do intervalo são considerados uma condição de erro. Para cada (‘PROCEDURE' ident Statements) encontrado, o (rootnode) cria uma instância de Classe (Statements,) que analisa as instruções. Um ponteiro para esta classe é colocado na coluna de procedimento da entrada da tabela de símbolos para (ident).

5.2 A linguagem de programação reversível R

A linguagem de programação reversível R, é uma linguagem de programação multiparadigma orientada a objetos, voltada à manipulação, análise e visualização de dados, não são a mesma coisa. A linguagem imperativa reversível R foi desenvolvida no laboratório do Massachusetts Institute of Technology (MIT) em 1997. A sintaxe é muito similar à de C e de LISP onde o código tem expressões S aninhadas, tem matrizes semelhantes como as de

C e aritmética de ponteiro. R é uma linguagem compilada, com o compilador disponível visando o conjunto de instruções reversíveis Pendulum, CLARK. (2001).

Gramática R	
prog ::= s*	(programa)
s ::= (defmain progname s*)	(rotina principal)
| (defsub subname (name*) s*)	(sub-rotina)
| (defword name $\underline{n}$)	(variável global)
| (defarray name $\underline{n}$ *)	(matriz global)
| (call subname e *)	(chamar sub-rotina)
| (rcall subname e*)	(sub-rotina de chamada reversa)
| (if e then s*)	(condicional)
| (for name = e to e s*)	(laço)
| (let (name <- e) s*)	(vinculação de variável)
| (printword e) | (println)	(resultado)
| (loc ++) | (- loc)	(incrementar/negativo)
| (loc <-> loc) | (loc $\odot$ e)	(trocar/atualizar)
loc ::= name | (* e) | (e _ e)	(localização)
e ::= loc | (e $\otimes$ e) | $\underline{n}$)	(expressão)
$\odot$ :: = += | −= |$^{=}$| ≤< | ≥>	(operador de atualização)
$\otimes$ :: = + | − | & | ≪ | ≫ | ∗ − /	(operador de expressão)
| = | < | > | <= | >= |!=	(operador relacional)

Figura 16 - Mostra uma gramática formal descrevendo as regras de sintaxe.

Fonte: CHOUDHURY. (2018), HAULUND (2017).

Atualmente, cada instrução R é compilado individualmente desta a forma de código fonte fica na forma de código de montagem Pendulum (função principal estado). A rotina principal pode invocar sub-rotinas que são definidas com a instrução (função sub). Além disso, um programa pode fazer uso de variáveis e matrizes do escopo global, definidos com a declaração (função palavras) e (função matriz).” Esses quatro tipos de instruções podem aparecer em qualquer lugar em um programa, mas só têm um efeito real quando aparecem como instruções de nível superior, CHOUDHURY. (2018). As instruções (ligar e ligue) são

usadas para invocar uma sub-rotina em qualquer direção de execução e correspondem às instruções (ligar e ligue) de Janus.

Portanto, ele não pode realizar otimizações nas instruções. Parâmetros vinculados a uma expressão ou constante devem manter seu valor em todo o corpo da sub-rotina para evitar comportamentos indefinidos ou irreversíveis CLARK. (2001). Para reduzir esses tipos de ineficiências, um novo compilador agora executa uma varredura de otimização do código da linguagem de montagem após a conclusão do processo de compilação normal. O compilador é capaz de reconhecer e eliminar de uma única computação os recálculos desnecessários de valores.

O fragmento R e seu código compilado (antes da otimização) na **Figura 15** mostram o tipo de otimização realizado pela varredura pós-compilação. Todas as instruções em negrito são removidas pelo otimizador. A instrução “se” é usada para execução condicional. É um requisito que o valor da expressão condicional seja o mesmo antes e depois da execução da instrução condicional, caso contrário, pode ocorrer um comportamento indefinido ou irreversível

(defword x 16) (defstring h "hello") (defstring w " world") (defmain main (print h) (println w) (print "Base 10: ") (printword x) (println) (print "Base 16: ") (println x :base 16))	hello world Base 10: 16 Base 16: 10

Figura 17 - Representação da função R de saída. Fonte: CLARK .(2001).

5.3 Otimização do código da linguagem de montagem Pendulum

Atualmente, o compilador compila cada instrução “R”, individualmente desde a forma de código fonte até a forma de código de montagem Pendulum CLARK.(2001). Portanto, ele não pode realizar otimizações nas instruções. Por exemplo, um valor intermediário usado por duas instruções consecutivas será calculado duas vezes. Além disso, para garantir a

reversibilidade, o valor calculado deve ser decrementado após cada uso, levando à seguinte sequência geral de eventos:

1. Cálculo do valor intermediário 2.
2. Primeiro uso do valor intermediário 3.
3. Decrementado do valor intermediário 4.
4. Cálculo do valor intermediário 5.
5. Segundo uso do valor intermediário 6.
6. Decrementado do valor intermediário 2.

```
(if (x) then                                    ADDI$  3X
   (print 1)                                    EXCH $4 $3
else                                            ADD $2  $4
   (print 0)                                    EXCH $4 $3
)                                               ADDI $3 -X
                                    _IFTOP:     BEQ $2 $0 _IFBOT
                                                OUTPUT $3
                                                ADDI $3 1
                                                OUTPUT $3
                                                ADDI $3 -1
                                     _IFBOT:    BEQ $2 $0 _IFTOP
                                                ADDI $3 X
                                                EXCH $4 $3
                                                SUB $2 $4
                                                EXCH $4 $3
                                                ADDI $3 -X
                                                ADDI $3 X
                                                EXCH $4 $3
                                                ADD $2 $4
                                                EXCH $4 $3
                                                ADDI $3 -X
                                    _ELSETOP:   BNE $2 $0 _ELSEBOT
                                                OUTPUT $3
```

```
                    OUTPUT $3
_ELSEBOT:    BNE $2 $0 _ELSETOP
             ADDI $3 X
             EXCH $4 $3
             SUB $2 $4
             EXCH $4 $3
             ADDI $3 -X
```

Figura 18 - Otimização do compilador R. Fonte: CLARK. (2001).

5.4 O microprocessador Pendulum e o Pendulum ISA (PISA)

A versão original da arquitetura do conjunto de instruções Pendulum (PISA) tinha algumas desvantagens. Havia uma falta de controle de software sobre dados lixo, o que significava que a arquitetura não era capaz de realizar todo o potencial de custo assintótico da SCRL. Essa arquitetura Pendulum é muito similar ao de PDP-8 e o RISC e foi o primeiro processador realmente programável reversivelmente de conjunto de instruções. Foi criada no Massachusetts Institute of Technology (MIT) por Carlin James Vieri CHOUDHURY.(2018). A PISA é uma linguagem de montagem do tipo MISP a qual foi várias vezes aperfeiçoada. Além disso, o conjunto de instruções não garantia total reversibilidade independentemente da correção do programa, o que impossibilitou algumas das possíveis aplicações alternativas para reversibilidade, como depuração bidirecional.

No caso de um salto, defina o contador de programa para o endereço do rótulo que tinham saltado. Em um processador reversível como o Pendulum, essas regras são muito mais complicadas, pois simplesmente sobrescrever o conteúdo do contador do programa constituiria uma perda de informação que quebra a reversibilidade. O processador Pendulum usa três registradores de propósito especial para lógica de fluxo de controle:

1. O contador de programa (PC) para armazenar o endereço da instrução atual
2. O registrador de ramificação (BR) para armazenar deslocamentos de salto
3. O bit de direção (DIR) para acompanhar a direção de execução

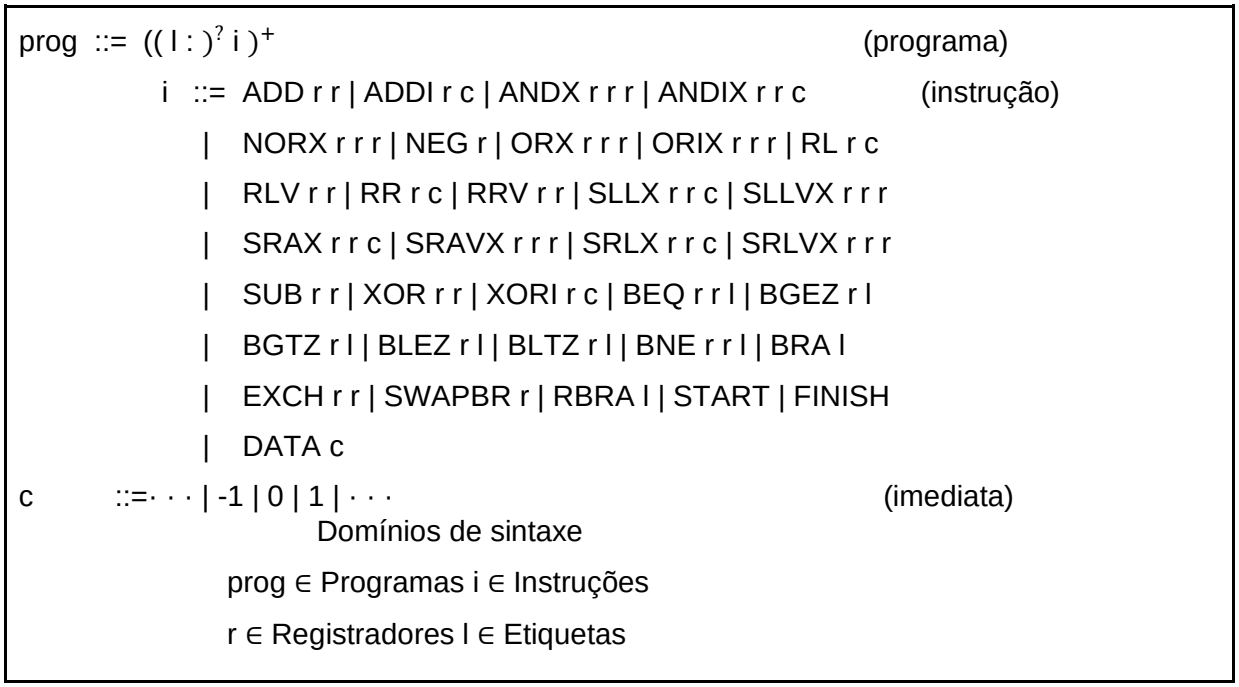


Figura 19 - Domínios de sintaxe e gramática EBNF para PISA. Fonte: CLARK. (2001).

6 Resultados e Discussão

Além do armazenamento estático acessível por meio de inteiros de “matrizes” comandos, desta versão da Janus permitem armazenamentos alocados dinamicamente na forma de pilhas de inteiros e variáveis locais. As instruções de modificação de pilha, push e pop, são usadas para manipular pilhas de inteiros da maneira usual, a única diferença é que empurrar uma variável para uma pilha limpa o conteúdo da variável com zero, enquanto o pop de um valor em uma variável pressupõe que a variável é limpa com zero. Isso significa que push e pop são inversões um do outro.

Uma atualização da variável reversível da Janus funciona atualizando uma variável no escopo atual de forma que o armazenamento original permaneça acessível por decrementação subsequente. Somente atualizações injetivas em seu primeiro argumento e com inversos definidos com precisão são permitidas e é um requisito que a expressão que está sendo atualizada não dependa de forma alguma do valor da variável que está sendo atualizada (para evitar perda de informação). Para garantir que tal atualização não ocorra, não é permitido que o identificador de variável do lado esquerdo da atualização ocorra em qualquer lugar do lado direito. Isso também exige uma restrição adicional: dois identificadores não podem se referir ao mesmo local na memória no mesmo escopo (uma situação conhecida como aliasing), pois isso faria pois isso seria uma maneira de contornar o requisito mencionado acima.

6.1 R

Na figura 16 a gramática formal descreve as regras de sintaxe R. Programas na linguagem reversível R simbolizam qualquer número de instruções, no entanto precisa conter exatamente uma rotina principal, pré-definida com a instrução (defmain). Para melhor o desempenho foi aprimorado a linguagem R, compilador e emulador PendVM. A expressão de condição pode ser uma combinação booleana de comparações relacionais. Os operadores booleanos suportados são AND (&&), OR (| |) e NOT (!), CHOUDHURY.(2018), KARPER. (2014). Os operadores relacionais suportados são =! =, <, >, <=, >=. Os parênteses devem ser incluídos de forma que cada operador tenha exatamente uma expressão entre parênteses em cada lado (exceto para NOT, que deve ter exatamente uma expressão à direita). A expressão inteira também deve estar entre parênteses. Abaixo é um exemplo válido:

$$(((x\ <\ y)\ \&\&\ (y!\ =\ (z+1)))\ ||\ (!\ z))$$

Na figura 17 há uma nova função de saída chamada “print” que é uma função de saída de uso geral. Ele pode ser usado para gerar o conteúdo de um registrador, uma variável de memória estática, uma variável de memória dinâmica, uma variável de “string” estática ou uma “string” literal. Ele também suporta opções para especificar parâmetros da representação de saída. A função (println) foi aprimorada para aceitar todos os mesmos tipos de dados e opções como (print). Quando (println) é usado sem nenhum argumento, ele gera apenas uma representação de nova linha como na versão original. A função (printword) ainda está disponível para compatibilidade com programas existentes. As funções (print) e (println) produzirão qualquer tipo de dados e permitirão que o programador especifique opções para transmitir ao ambiente de tempo de execução como os dados devem ser exibidos. Para variáveis inteiras, as opções de exibição consistem em complemento a dois com sinal, complemento a dois sem sinal, formato decimal e hexadecimal. O programa R e sua saída na Figura 5

i	i^{-1}
ADD r1 r2	**SUB r1 r2**
SUB r1 r2	**ADD** r1 r2
A**DDI** r c	**ADDI** r −c
RL r c	**RR** r c
RR r c	**RL** r c
RLV r1 r2	**RRV** r1 r2
RRV r1 r2	**RLV** r1 r2

Figura 20 - Regras de inversão para instruções PISA, todas as outras instruções são auto inversas. Fonte: CHOUDHURY.(2018).

6.2 O microprocessador Pendulum ISA (PISA)

Um programa PISA é uma lista de instruções de máquina no estilo RISC (possivelmente rotuladas), veja a figura 21 para um trecho representativo. Uma instrução é uma instrução de dados, ou uma instrução de ramificação, ou uma instrução especial usada no controle do programa CHOUDHURY.(2018), KARPER. (2014). Todas as instruções PISA têm inversas que também são instruções PISA únicas. O inverso de ADD é SUB, o inverso de ANDX é ele mesmo, etc.

Símbolo	Nome da classe de arquitetura	Entropia gerada por operação
FIA	Arquitetura totalmente irreversível	Θ (1)
TPRA	Arquitetura proporcionalmente reversível	Θ (1/top)
BRA	Arquitetura balisticamente reversível	0

Tabela 3 modelos de máquinas.

As três classes de modelos de máquinas físicas que são comparadas na tabela acima. A diferença que define entre eles está em como a entropia média gerada por operação computacional é dimensionada em relação ao período de tempo no qual a operação é executada.

$prog$	::= $p_{main}\ p^*$	(programa)
t	::= **int | stack**	(tipo de dados)
p_{main}	::= **procedure main () (int** x**([** $\underline{n}$ **])**$^?$ | **stack** $x\ )^* s$	(procedimento principal)
p	::= **procedure** $q(t\ x, \ldots, t\ x)\ \ s$	(definição do procedimento)
s	::= $x\ \odot = e$ | $x[\ e\ ] \odot = e$	(atribuição)
	| $\boldsymbol{if}\ e\ \boldsymbol{them}\ \ s\ \ \boldsymbol{else}\ s\ fi\ e$	(condicional)
	| $from\ \ e$ **do** s loop s until e	(laço)
	| $\boldsymbol{push}\ (x, x)$ | $pop(x, x)$	(modificação de pilha)
	| $local\ \ t\ x\ = e\ s\ \ delocal\ \ t\ x\ = e$	(bloco de variável local)
	| $call$ q $(x, \ldots, x)$ | $unicall$ q$(x, \ldots, x)$	(invocação de procedimento)
	| $skip$ |$s\ s$	(sequência de instruções)
e	:: = n | x | $x[e]$ | $e \otimes e$ | empty(x) | top(x) | nil	(expressão)
$\odot$	:: =+ | - | ^	(operador)
$\otimes$	:: = e | $x[\ e\ ]$ (main procedure) (procedure definition) (loop) | * | / | % | & | | | && | || | < | > | = | != | <= | >=	(operador)

Domínios de sintaxe

prog ∈ Progs s ∈ Stms d ∈ Vdecs ∈ ModOps

p ∈ Procs e ∈ Exps t ∈ Types ⊗ ∈ Ops

q ∈ PIds x ∈ Vars c ∈ Cons

Figura 21 - Sintaxe de Janus. Fonte: CHOUDHURY. (2018).

```
/** Calculando a raiz quadrada de um inteiro. * Versão adaptada do exemplo de
"Janus: uma linguagem reversível no tempo"
por * C. Lutz e H. Derby.
 */
// Calculates floor(sqrt(num))
procedure root(int num, int root)                    procedure root (int num, int
root)
      local int bit = 1                                                num += root * root
      from bit = 1 loop/ encontrar parque de bola exponencial....      local int bit = 1
      call doublebit (bit).....                                  ....  from bit = 1 do
      until (bit * bit) > num                                  ..if root / bit % 2 != 0 then
      from (bit * bit) > num do                                              root -= bit
uncall doublebit (bit)                                  fi (root + bit) * (root + bit) <= num

     if ((root + bit) * (root + bit)) <= num then.....            uncall doublebit(bit)
      root += bit.....                                            until bit * bit > nu
      fi (root / bit) % 2! = 0                                  from bit * bit > num loop
      until bit = 1                                               call doublebit(bit)
      delocal int bit = 1                                 .....          until bit = 1
      num -= root * root                                          delocal int bit = 1
procedure doublebit(int bit)                             procedure doublebit(int bit)
      local int z = bit                .....                      local int z = bit / 2
      bit += z                                 .....                          bit -= z
      delocal int z = bit / 2                                .....    delocal int z = bit
procedure main()                                              .....procedure main()
      int num                                                  .....int num
      int root                                                 .....int root
      num += 66
      call root(num, root)      num += 66                        call root(num, root)
      /*
      root += 25
```

```
        uncall root (num, root)
        */
saida:
num = 2
root =
```

Figura 22 - mostrando dois códigos esquerdo normal e no lado direito o código reverso. Fonte: (SIMULADOR_JANUS).

```
    /* Codigo  Janus  traduzido para linguagem C */
#include <stdio.h>      /* printf */
#include <assert.h>
#include <math.h>
void root_forward (int & num, int &root);
void root_reverse (int & num,  int &root);
void doublebit_forward (int &bit);
void doublebit_reverse (int &bit);
void root_forward (int &num, int &root) {
    int bit = 1;
    assert (bit == 1);
    while (!(bit * bit > num)) {
        doublebit_forward(bit);
        assert(!(bit == 1));
    }
    assert(bit * bit > num);
    doublebit_reverse(bit);
    if ((root + bit) * (root + bit) <= num) {
        root += bit;
        assert(root / bit % 2 != 0);
    }
    while (!(bit == 1)) {
        assert (!(bit * bit > num));
        doublebit_reverse(bit);
        if ((root + bit) * (root + bit) <= num) {
            root += bit;
            assert (root / bit % 2 != 0);
        }
    }
    Assert (bit == 1);
    num -= root * root;
```

```
}
void root_reverse (int & num, int &root) {
    num += root * root;
    int bit = 1;
    assert (bit == 1);
    if (root / bit % 2 != 0) {
        root -= bit;
        assert ((root + bit) * (root + bit) <= num);
    }
    doublebit_forward(bit);
    while (!(bit * bit > num)) {
        assert (! (bit == 1));
        if (root / bit % 2! = 0) {
            root -= bit;
            assert ((root + bit) * (root + bit) <= num);
        }
        doublebit_forward(bit);
    }
    Assert (bit * bit > num);
    while (! (bit == 1)) {
        doublebit_reverse(bit);
        assert (! (bit * bit > num));
    }
    assert (bit == 1);
}
void doublebit_forward (int &bit) {
    int z = bit;
    bit += z;
    assert (z == bit / 2);
}
void doublebit_reverse (int &bit) {
```

```
    int z = bit / 2;
    bit -= z;
    assert (z == bit);
}
int main () {
    int num = 0;
    int root = 0;

    num += 66;
    root_forward (num, root);
    return 1;
}
```

Figura 23 - é código em C dois primeiros. Fonte: (SIMULADOR_JANUS).

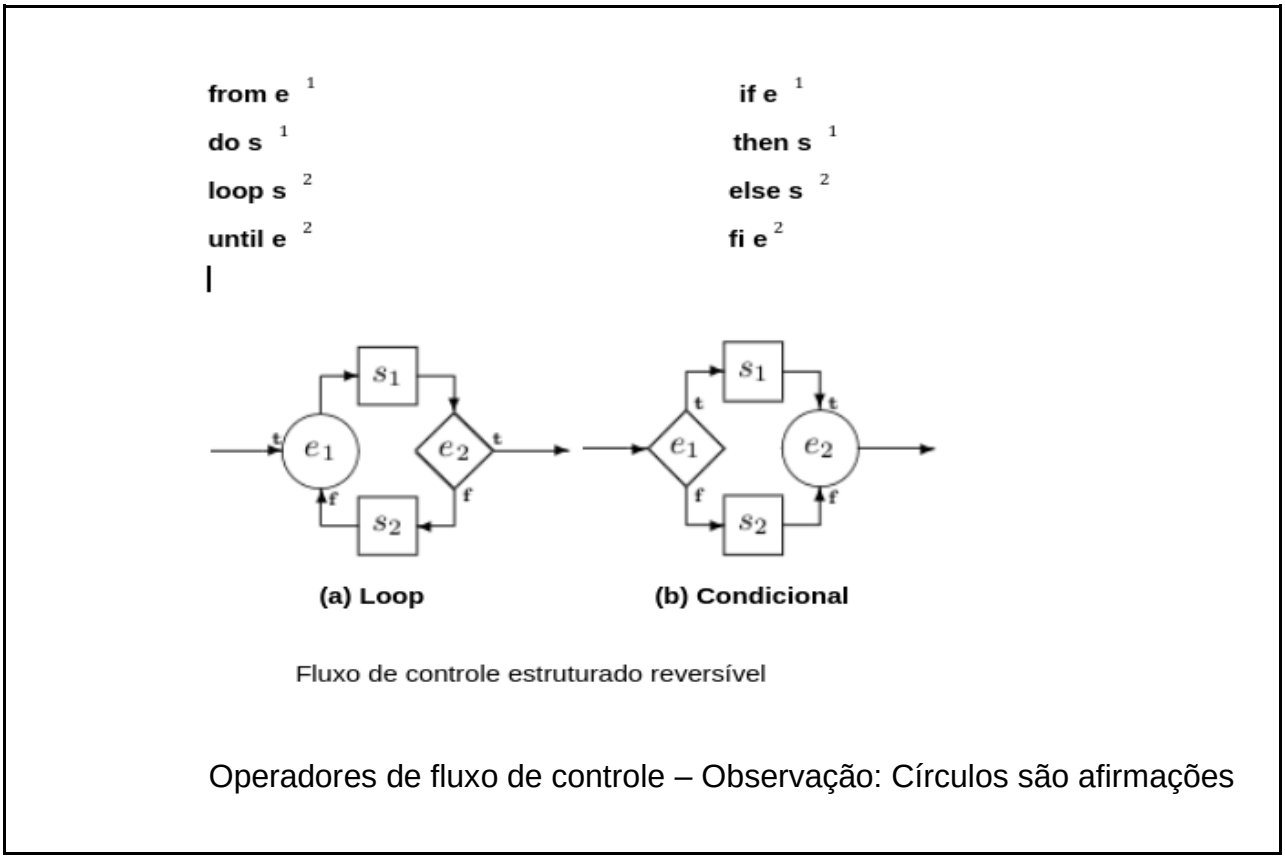


Figura - 24 - mostra o fluxo de controle estruturado reversível. Fonte: CLARK. (2001).

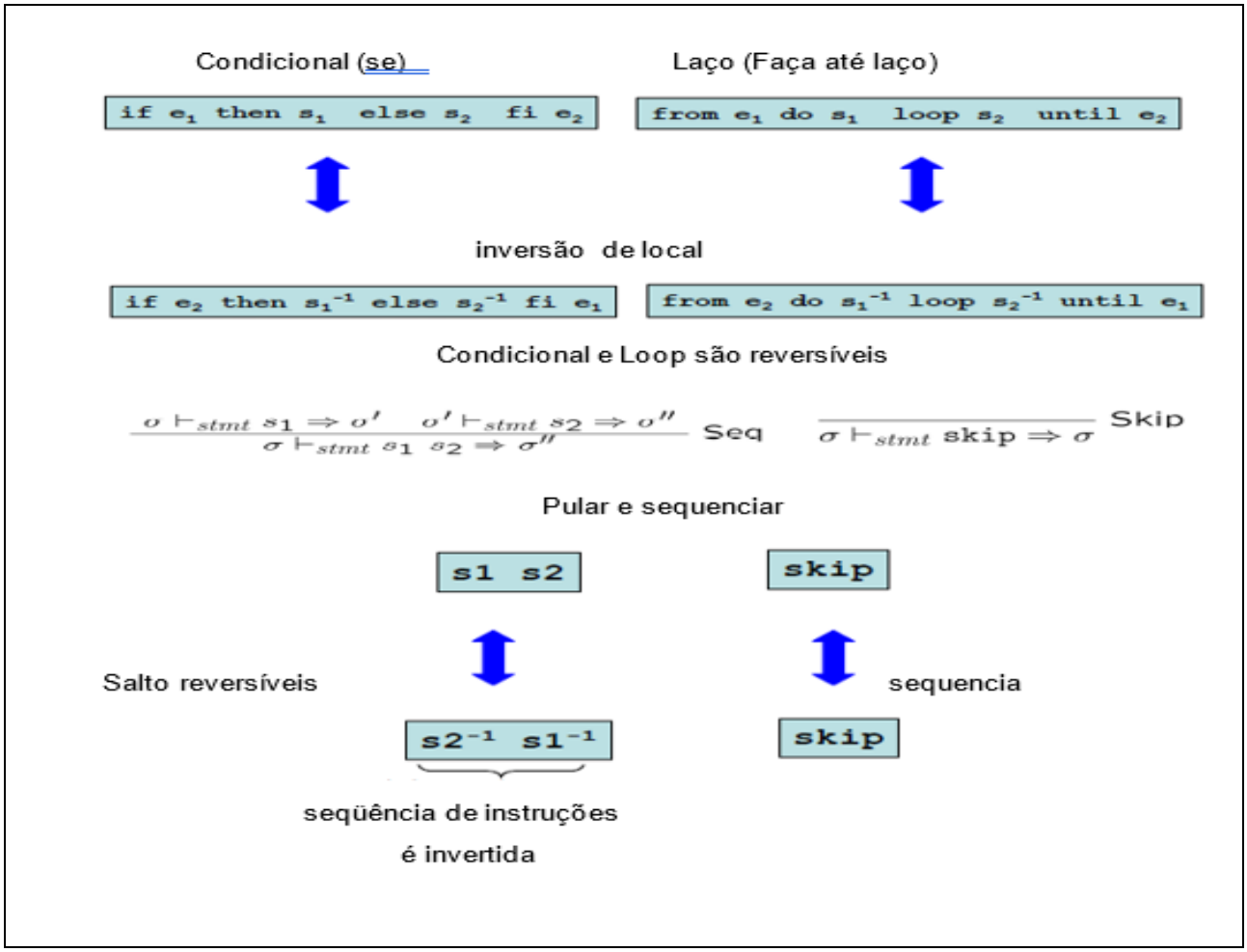


Figura 25 é extensão da Figura – 24 Condicional e Loop são reversíveis.

Fonte: CLARK (2001), CHOUDHURY (2018)

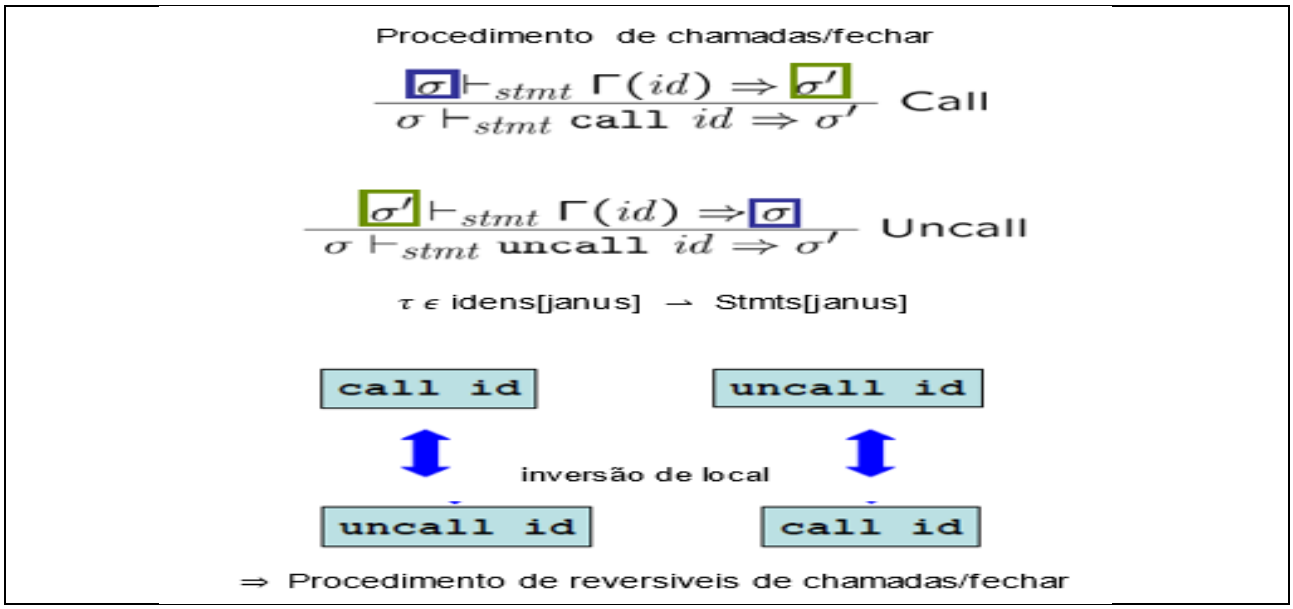


Figura 26 extensões da figura 25. procedimento de chamar/fechar.

Fonte: CLARK .(2001), CHOUDHURY.(2018)

6.3 Janus

A semântica da notação de comandos da Janus em termos de relações funcionais injetivas, ou seja, funções injetivas parciais. Cada construção da linguagem Janus é apresentada como uma composição adequada de alguns combinadores categóricos apresentados na seção anterior.

O objetivo é duplo: visamos impor a reversibilidade (já que a interpretação é obtida pela composição de algumas funções reversíveis básicas) e visamos evidenciar uma conexão entre Janus é um framework de linguagens categóricas reversíveis, fornecemos uma semântica de notação de comandos da Janus em termos de relações funcionais injetivas, ou seja, funções injetivas parciais. Cada construção da linguagem Janus é interpretada como uma composição adequada de alguns combinadores categóricos apresentados.

Como citada em Choudhury CHOUDHURY.(2018), JANUS.(2022). A interpretação de números de estados é dada pela função identidade. Denotamos com Σ o conjunto de todos os estados. Por simplicidade, paramos para anotar estados com o programa envolvido (está implícito no contexto), mas ainda assumimos que o estado é uma função de todas as variáveis envolvidas para números naturais. Uma expressão e da linguagem é interpretada em uma relação funcional $[e]$ de estados para N da seguinte maneira.

$[n]$ = {(σ, n) | σ ∈ Σ}

$[y]$ = {(σ, σ (y)) | σ ∈ Σ}

$[e_1 \odot\ e_2]$ ={(σ, n) | $(\sigma, n_1)\ \in [e_1], (\sigma, n_2)\ \in [e_2], n\ =\ n_1\ \ [\odot]\ n_2$}

Observe que a relação está sendo a interpretação de uma expressão que pode não ser injetiva utilizando conceitos das linguagens de programação de comandos, os programas construídos em tais linguagens têm os preceitos de gerar programas e concluí-los. Nessas linguagens de comandos inclui-se predicados, quantificadores, conectivos lógicos e regras de inferência que, como veremos, fazem parte do Cálculo de Predicados. Predicados Figura 24(a): um teste na entrada "e_1" e uma asserção na saída "e_2" da condicional. O predicado e_2 deve ser verdadeiro quando o fluxo de controle atinge a asserção ao longo da borda verdadeira "rotulada t" e falso quando o fluxo de controle atinge a asserção ao longo da aresta falsa "rotulada f"; caso contrário, a operação é indefinida "parada anormal". Os comandos ""e_1" e "e_2" são os ramos then e else, respectivamente.

A asserção "marcada com um círculo no diagrama para distingui-la de um teste" torna a condicional determinística inversa; na direção inversa, uma asserção atua como um teste e

um teste como uma asserção. Asserções são uma parte operacional de um programa da mesma forma que os testes. Um laço reversível tem dois predicados Figura 24(b): uma afirmação na entrada ““e_1” e um teste na saída do laço “e^2”. Inicialmente, a asserção ““e_1” deve ser verdadeira e então ““e_1” é executado. O “laço” termina se o teste “e_2”” for verdadeiro; caso contrário, “e_2””, é executado, após o qual ““e_1”, deve ser falso.

A afirmação é apenas inicialmente verdadeira. O “laço” é repetido enquanto a afirmação e o teste forem falsos e termina quando o teste for verdadeiro. Isso torna o laço para trás determinístico. Por brevidade, permitimos a omissão de ramificações de salto nas construções de fluxo de controle. Por exemplo, de ““e_1” dos s até “e_2”” é açúcar sintático para de ““e_1” dos s pular até “e_2””.

Uma instrução condicional da Janus tem uma condição de ramificação e uma asserção de saída, ambas expressões. A condição de ramificação determina qual ramificação da condicional é executada, enquanto a asserção de saída é usada para unir reversivelmente os dois caminhos de computação. Se a condição de ramificação for avaliada como verdadeira, a ramificação será executada sobre a qual a asserção de saída também deverá ser avaliada como verdadeira CHOUDHURY.(2018). Se a condição de ramificação for avaliada como falsa, a ramificação “else” é executada após o que a asserção de saída deve ser avaliada como falsa.

Se a asserção de saída não corresponder à condição de ramificação, a instrução será indefinida. Inicialmente, a asserção de entrada deve ser avaliada como verdadeira, após o que a instrução do é executada. Se a condição de saída for verdadeira, o laço termina, caso contrário, a instrução de loop é executada sobre a qual a asserção de entrada deve agora ser avaliada como falsa. Quando executado em sentido inverso, a condição de saída serve como asserção de entrada e vice-versa. Na Figura 24 mostra um fluxograma ilustrando a mecânica dos laços reversíveis.

7 Comparação dos trabalhos

Faremos uma análise comparativa dos trabalhos recentes. Entendemos que há diferenças importantes entre circuitos reversíveis e computadores de propósito especial versáteis, por um lado, e computadores universais reversíveis, por outro lado. Diante disso, enquanto ainda não se projetar um modelo reversível de propósito especial para cada circuito irreversível particular usando portas universais reversíveis, tal método não produz um

compilador irreversível para reversível que possa executar qualquer programa irreversível em uma arquitetura de computador reversível universal fixa, como estamos interessados aqui. Mas há o modelo de máquina de Turing com as operações elementares construída no formato quádruplo “p, s, a, q”, significando que se o controle finito está no estado “p” e “a” máquina varre o símbolo da fita s, então a máquina executa a ação “a” e subsequentemente o controle finito entra no estado “q”. Em referência a outros estudos sobre desenvolvemos uma teoria matemática para o número inevitável de operações de bits irreversíveis em uma computação reversível.

Na computação adiabática muitos esquemas físicos atualmente propostos que implementam computação adiabática reduzem a irreversibilidade usando tempos de computação mais longos LI_Ming_VITÁNYI.(1996). Este tipo de computação não dissipa energia, a única dissipação de energia ocorre puxando a tensão para cima e para baixo: quanto mais lento for, menos energia é dissipada. Cálculos podem ser executados logicamente de forma reversível, ao custo de eventualmente encher a memória com informações indesejadas de LI_Ming_VITÁNYI.(1996) lixo. Isso significa que computadores reversíveis com memórias limitadas requerem, a longo prazo, operações de bits irreversíveis, por exemplo, para apagar registros irreversivelmente para criar espaço de memória livre poderia citar várias linguagens de programação reversíveis, no entanto só vai ser citada Janus que representa as demais. Janus é um compilador e interpretador para a linguagem Janus reversível no tempo.

O compilador Janus é escrito em SLIMEULA e compila o código em uma estrutura de classe SLIMEULA interna que pode ser interpretada diretamente. “SLIMEULA” significa SIMULA rodando em um DECSYSTEM-20. Um operador de fluxo de controle reversível (condicional, loop), uma operação de pilha (push, pop), um bloco de variável local, uma invocação de procedimento (call, uncall), um salto ou uma sequência de instruções Ele não será mantido e não pretende ser robusto. O compilador consiste em quatro partes principais: Um analisador léxico que tokeniza o fluxo de entrada e gera uma tabela de símbolos; um analisador descendente recursivo feito do Código de Inicialização das Classes SLIMEULA; um interpretador que consiste no procedimento 'exec' comum a todas as Classes criadas na análise; e o scanner de comando de tempo de execução.

var[index]
Imprime o valor de var [index]
Var
Imprime todos os elementos da matriz var
var=n
Sets var [0] to n
var[index]=n
Sets var[index] to n
CALL name
Chama o nome do procedimento.
UNCALL name
ancela o nome do procedimento
SYMBOLS
Tabela de tipos de todos os símbolos e seus atributos.
TRACE
Ativa o recurso de rastreamento. Lista as instruções à medida que são executadas, juntamente com os valores de quaisquer variáveis que são modificadas.
UNTRACE
Desativa o recurso de rastreamento.
RESET
Redefine todas as variáveis para zero.
RESET var
Redefine todos os elementos da matriz var para zero.

Figura -27 mostra alguns procedimentos da Janus.

Fonte: CLARK .(2001), (SIMULADOR_JANUS).

A simulação reversível em T, os passos de uma computação irreversível de x a $f(x)$ calcula reversivelmente da entrada x na saída, $(f(x))$ em tempo $T' = O(T)$.No entanto, como esta simulação reversível em algum instante de tempo tem que registrar todo o histórico da computação irreversível, seu uso de espaço aumenta linearmente com o número de passos simulados. Ou seja, se a computação irreversível simulada usa espaço S, então para alguma

constante C >1 a simulação usa tempo $T' \approx C + CT$ e espaço $S' \approx C + C(S + T)$. Isso pode ser uma quantidade inaceitável de espaço para muitos cálculos úteis na prática.

8 Conclusões e Trabalhos Futuros

O foco desta monografia é avaliar aspectos do tema computação reversível e linguagens reversíveis seus fundamentos e aplicações. A investigação o sobre tema descortina quão é interessante o estudo. Mas a bem da verdade é que a lei de Moore diz que o poder computacional praticamente dobrou a cada 18 meses. Este aumento de potência deve-se principalmente à contínua miniaturização dos elementos dos quais computadores são feitos. Como o aumento linear da frequência de clock é cada vez maior, consequentemente há a dissipação de energia. É importante frisar que as leis da física não impedem criar tecnologias como computador logicamente semelhante para operar fisicamente em um ambiente sem dissipação Bennett1. (1973), BENNET.(1982).

No entanto há uma diferença decisiva entre circuitos reversíveis e computadores reversíveis de uso especial. Embora se possa projetar uma versão reversível de propósito especial para cada circuito irreversível particular usando portas universais reversíveis. Agora considere o formato especial (determinístico) das máquinas de Turing usando quádruplas. Uma máquina de Turing determinística é definida como uma máquina de Turing com quádruplas dos quais dois não se sobrepõem em domínio Bennett1.(1973).

As funções recursivas primitivas (PR) formam uma grande e importante subclasse enumerável das funções recursivas (total computável). Sua vastidão é óbvia pelo fato de que toda função recursiva total cuja complexidade de tempo é limitada por uma função recursiva primitiva é ela mesma recursiva primitiva, enquanto sua importância é bem expressa pelo Teorema da Forma Normal de Kleene BENNET. (2009), e pelo fato de que todo conjunto recursivamente enumerável pode ser enumerado por uma função recursiva primitiva.

Muitas linguagens de programação reversíveis essenciais aparecem na literatura por exemplo SRL a foi concebido destilando o núcleo reversível da linguagem LOOP. A Primeira, permite programar apenas procedimentos totais. Em segundo lugar, também é um núcleo (reversível) de uma linguagem de programação imperativa padrão . Quase todas as linguagens de programação reversíveis são concebidas para serem completas, de modo que o primeiro recurso distingue o SRL delas. Consideremos a segunda característica. Janus foi a

primeira linguagem de programação reversível destilada de uma linguagem de programação estruturada imperativa.

Foi feita uma tentativa de tradução correta e eficiente para compilar a linguagem de programação reversível de alto nível Janus para a linguagem de máquina reversível de baixo nível PISA. Programas alvo produzidos usando nesta tradução conservam tanto a semântica (correção) quanto as complexidades de espaço / tempo (eficiência) dos programas fonte. A compilação reflete isso traduzindo cada instrução individual no programa de origem de forma limpa. Isso tem o efeito de que em nenhum ponto da execução de qualquer programa traduzido nós acumulamos mais do que uma quantidade constante de dados de lixo temporários. O processo não foi o esperado, mas ainda não paramos porque aqui.

Referências

CHOUDHURY Vikraman. Reversible Programming Languages. 2018

JANUS time-reversible computing programming language). Janus. 04 Jan 2022. Disponível em**:** https://en.wikipedia.org/wiki/Janus_(time-reversible_computing_programming_language) Accessed:2022-04-10.

TOFFOLI Tommaso. Reversible computing. In: International colloquium on automata, languages, and programming. Springer, Berlin, Heidelberg, 1980. p. 632-644.

PAOLINI Luca; PICCOLO, Mauro; ROVERSI, Luca. On a class of reversible primitive recursive functions and its turing-complete extensions. New Generation Computing, v. 36, n. 3, p. 233-256, 2018.

MATOS MATOS, A. B. Register reversible languages (work in progress). Technical report, LIACC, 2014.

MATOS2 MATOS, Armando B.; PAOLINI, Luca; ROVERSI, Luca. On the expressivity of total reversible programming languages. In: International Conference on Reversible Computation. Springer, Cham, 2020. p. 128-143

.Landauer Reversible computing. WIKIPEDIA. 04 Jan 2022. Disponível em**:** https://en.wikipedia.org/wiki/Reversible_computing . Acessado 04 Jan 2022.

_SPECTRUM MICAHELP. FRANK. The FUTURE OF COMPUTING DEPENDS ON MARKINGIT REVERSIBLE. 25 agosto 2017 Disponível em: https://spectrum.ieee.org/the-future-of-computing-depends-on-making-it-reversible . Acessado 04 Out 2022.

LUKAC LUKAC, Martin et al. Building a Completely Reversible Computer. arXiv preprint

arXiv:1702.08715, 2017.

KARPER, Aaron. A Programming Language Oriented Approach to Computability. arXiv preprint arXiv:1402.2949, 2014.

HAULUND HAULUND, Tue. Design and implementation of a reversible object-oriented programming language. arXiv preprint arXiv:1707.07845, 2017.

Bennett Charles H. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's demon. IBM Research Division, Yorktown Heights, NY 10598, USA, 2007.

Bennett1 Charles H. Bennett. Logical reversibility of computation. IBM Journal of Research and Development, 6:525{532, 1973.

SZILARD Leo Szilard. On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings. *Zeitschrift far Physik,* 53:840-852, 1929. English translation in *Behavioral Science,* 9:301-310,1964

Termodinâmica Reversible process (thermodynamics).2022**.** WIKIPEDIA. 04 Out 2022. Disponível em**:** https://en.wikipedia.org/wiki/Reversible_process_(thermodynamics) Acessdo:2022- 04-10.

Kolmogorov Complexidade de Kolmogorov. 2020. WIKIPEDIA. 04 Out 2022. Disponível em**:** https://pt.wikipedia.org/wiki/Complexidade_de_Kolmogorov Acessado: 2022-04-10.

AXELSEN AXELSEN, Holger Bock. Clean translation of an imperative reversible programming language. In: *International Conference on Compiler Construction*. Springer, Berlin, Heidelberg, 2011. p. 144-163

SIMULADOR_ISA Pendulum ISA Simulator. W3C. 04 Out 2022. Disponível em**:** https://www.cise.ufl.edu/research/revcomp/users/adickins/phpisa/phpisa.html. Acessado: 2022-04-10.

_SIMULADOR_JANUS About the Janus Playground**.** Janus 04 Out 2022. Disponível em**:** http://topps.diku.dk/pirc/janus-playground/#examples/factor . Acessado: 2022-04-10.

CLARK CLARK, Christopher R.; MICHAEL, D.; FRANK, P. Improving the reversible programming language r and its supporting tools. Senior Project, 2001.

BENNETT BENNETT, Charles H. et al. Information distance. IEEE Transactions on information theory, v. 44, n. 4, p. 1407-1423, 1998.

LI_Ming_VITÁNYI LI, Ming; VITÁNYI, Paul. Reversibility and adiabatic computation: trading time and space for energy. *Proceedings of the Royal Society of London. Series A:*

Mathematical, Physical and Engineering Sciences, 1996, 452.1947: 769-789.

LUTZ LUTZ, Christopher; DERBY, Howard. Janus: A time-reversible language. A letter to R. Landauer. [1986-04-01]. http://www. else. ufl. edu/mpf/rc/janus, html, 1986.

Frank Michael P Frank. The r programming language and compiler. Technical report, MIT Reversible Computing Project Memo, 1997.

VIERI Carlin James Vieri. Pendulum–a reversible computer architecture. PhD thesis, Massachusetts Institute of Technology, 1995

CAD Computer-aided design. WIKIPEDIA. 04 Out 2022. Disponível em**:** https://en.wikipedia.org/wiki/Computer-aided_design. Acessado: 2022-04-10.

MEYAR MEYER, Albert R.; RITCHIE, Dennis M. The complexity of loop programs. In: Proceedings of the 1967 22nd national conference. 1967. p. 465-469

Acessado: 2022-04-10

ENERGIA Consumo de energia: quanto gasta um computador que fica sempre ligado.2020. idealista/News. 11 maio 2022. Disponível em**:** https://www.idealista.pt/news/financas/lar/2022/05/11/52219-consumo-de-energia-quanto-gasta-um-computador-que-fica-sempre ligado#:~:text=O%20c%C3%A1lculo%20%C3%A9%20o%20seguinte,%3D%201%2C56%20kWh%2Fdia. Acessado: 2022-04-10

THOMSEN THOMSEN, Michael Kirkedal. Design of Reversible Logic Circuits using Standard Cells. University of Copenhagen, Copenhagen, 2012.

BENNET BENNETT, Charles H. The thermodynamics of computation—a review. International Journal of Theoretical Physics, v. 21, n. 12, p. 905-940, 1982

.KLEENE Stephan Cole Kleene. Introduction to Metamathematics. North-Holland, 1952. Reprinted by Ishi press, 2009.

ZVONKIN, Alexander K.; LEVIN, Leonid A. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. Russian Mathematical Surveys, v. 25, n. 6, p. 83, 1970.

Acessado: 2022-04-10

.Reversible Reversible computing. WIKIPEDIA. 04 Out 2022. Disponível em: https://en.wikipedia.org/wiki/Reversible_computing. Acessado: 2022-04-10

.BILHAR Experimento volta no tempo usando computador quântico**.** BAINNIAC. 04 Out 2022. Disponível em: http://www.brainniac.ufv.br/index.php/experimento-volta-no-tempo-usando-computador-

quantico/. Acessado: 2022-04-10

IWAMA IWAMA, Kazuo; KAMBAYASHI, Yahiko; YAMASHITA, Shigeru. Transformation rules for designing CNOT-based quantum circuits. In: Proceedings of the 39th annual Design Automation Conference. 2002. p. 419-424.