UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Thiago Pacheco Rocha

Higher-order Virtual Machine 3: Uma reconstrução racional

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Thiago Pacheco Rocha

Higher-order Virtual Machine 3: Uma reconstrução racional

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Alexsandro Santos Soares

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4144 - http://www.portal.facom.ufu.br/ facom@ufu.br

ATA DE DEFESA - GRADUAÇÃO

Curso de Graduação em:	Ciência da Computação				
Defesa de:	GBC084 Projeto de Graduação				
Data:	23/09/2025	Hora de início:	9:35	Hora de encerramento:	10:50
Matrícula do Discente:	12121BCC033				
Nome do Discente:	Thiago Pacheco Rocha				
Título do Trabalho:	Higher-order Virtual Machine 3: uma reconstrução racional				
A carga horária curricular foi cumprida integralmente?	(x) Sim ( ) Não				

Reuniu-se na sala 1B126, Campus Santa Mônica, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Curso de Graduação em 09/2025, assim composta: Professores: Carlos Roberto Lopes - FACOM/UFU; Maria Adriana Vidigal de Lima - FACOM/UFU; Alexsandro Santos Soares - FACOM/UFU, orientador do candidato.

Iniciando os trabalhos, o presidente da mesa, Dr. Alexsandro Santos Soares, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao discente a palavra, para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado com nota 100

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Alexsandro Santos Soares**, **Professor(a) do Magistério Superior**, em 23/09/2025, às 11:15, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Maria Adriana Vidigal de Lima**, **Professor(a) do Magistério Superior**, em 23/09/2025, às 11:31, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Carlos Roberto Lopes**, **Professor(a) do Magistério Superior**, em 23/09/2025, às 13:44, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **6701694** e o código CRC **E9C2B905**.

Referência: Processo nº 23117.066326/2025-54 SEI nº 6701694

Agradecimentos

Agradeço à minha família por todo o apoio, confiança e paciência infinita durante a elaboração deste trabalho.

Sou grato ao meu orientador, Professor Alexsandro, que me motivou desde o início a explorar esta área incrível das redes de interação, sobre a qual eu não possuía conhecimento prévio.

Agradeço sinceramente a todos os meus amigos. Sem o seu apoio, eu não teria encontrado forças para chegar até aqui.

Resumo

A *Higher-order Virtual Machine 3* (HVM3) é um compilador que gera código de alta performance baseado em redes de interação, que se destaca por seu potencial de execução massivamente paralela na sua versão de avaliação ansiosa (*strict*) e pela redução ótima de termos do cálculo-λ na versão de avaliação preguiçosa (*lazy*). Apesar das suas propriedades promissoras, a complexidade de seu funcionamento e a carência de documentação didática representam uma barreira para novos pesquisadores e desenvolvedores. Este trabalho se propõe a preencher essa lacuna, descrevendo de forma incremental e detalhada a sintaxe, a semântica e o funcionamento da HVM3, com foco na sua versão *Lazy*. Partindo dos fundamentos teóricos do cálculo-λ e dos combinadores de interação, a máquina virtual é reconstruída passo a passo através de uma série de submáquinas, cada uma adicionando novas funcionalidades à anterior. Inicia-se com o modelo de memória, evoluindo para a implementação dos combinadores de interação, e subsequentemente, incorporando operações aritméticas, referências, casamento de padrões para tipos de dados algébricos (ADTs), mecanismos de priorização de redução, etc. O resultado é uma documentação que serve como um guia para a compreensão da HVM3, tornando este poderoso modelo de computação mais acessível e fomentando futuras pesquisas na área.

Palavras-chave: Redes de interação, Combinadores de interação, Cálculo de interação, Avaliação ótima, HVM, Cálculo lambda

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

HVM	Higher-order Virtual Machine
HOC	Higher Order Company
IC	Interaction Calculus (Cálculo de interação)
LSP	Language Server Protocol
ADT	Algebraic Data Type (Tipo de dado algébrico)

Sumário

1 Introdução

1.1 Contextualização

O cálculo lambda (cálculo-λ), introduzido por Alonzo Church em 1932, estabeleceu um modelo fundamental de computação baseado puramente em funções. Este formalismo forneceu uma base teórica sólida para a programação funcional e a compreensão da computabilidade. Apesar de sua expressividade, o cálculo-λ apresenta desafios inerentes a sua implementação eficiente e paralelização devido à natureza global das substituições de variáveis.

Em contraste, as redes de interação, propostas por Yves Lafont em 1989, apresentam um modelo de computação distribuída fundamentado em reescrita de grafos, com regras de interação locais e atômicas. Os combinadores de interação, sendo um sistema específico das redes de interação que utiliza um conjunto mínimo de células (construtores, duplicadores e apagadores), constituem um modelo universal de computação distribuída. A relevância contemporânea dos combinadores de interação reside em suas propriedades desejáveis para a computação moderna, como a confluência forte, que permite execução massivamente paralela sem a necessidade extensiva de sincronização global.

O Cálculo de Interação (IC), inspirado no cálculo-λ, surge como um sistema de reescrita mínimo com potencial para maior eficiência e expressividade, constituindo uma base teórica para máquinas virtuais de alta performance. A capacidade de traduzir o cálculo-λ para o cálculo de interação e vice-versa destaca a conexão teórica entre esses modelos e a praticidade do cálculo de interação como um alvo de compilação eficiente.

1.2 Problema

Apesar da elegância teórica e das propriedades promissoras dos combinadores de interação e do Cálculo de Interação para a construção de máquinas virtuais eficientes e altamente paralelizáveis, como as Higher-order Virtual Machines (HVMs), há uma carência de material didático e descritivo que detalhe a semântica de seu funcionamento e as otimizações algorítmicas empregadas em suas implementações, dificultando tanto a compreensão por parte de interessados quanto o desenvolvimento de novas implementações ou extensões.

O processo de tradução de conceitos de linguagens de alto nível para as

instruções de uma máquina virtual (como a HVM), passando pelo cálculo-λ e, posteriormente pela representação em combinadores de interação, não é trivial e requer compreensão profunda das regras de interação e da estrutura das redes. Assim, faz-se necessário um trabalho que descreva de forma didática e incremental o funcionamento da HVM, a fim de tornar este modelo de computação mais acessível e fomentar futuras pesquisas e desenvolvimentos na área.

1.3 Hipótese

É possível facilitar a compreensão do runtime da HVM3 reconstruindo-a incrementalmente a partir de máquinas virtuais menores que, sucessivamente, subsumam as instruções.

1.4 Objetivos

O presente trabalho objetiva descrever didaticamente a sintaxe e semântica da linguagem da Higher-order Virtual Machine 3 (HVM3), particularmente sua versão de avaliação preguiçosa (*Lazy*), assim como explicar as adições feitas pela Higher Order Company (HOC), empresa responsável pelo desenvolvimento e implementação da HVM3, aos combinadores de interação para tornar a implementação desse modelo eficiente na prática, introduzindo-as conforme necessário ao longo da definição de submáquinas, que poderão executar um subconjunto de instruções da HVM3.

1.5 Justificativa

Não há, até onde o autor pôde verificar, uma descrição teórica e didática da semântica e algoritmos da HVM3. No momento da escrita deste trabalho, a HVM3 está sob desenvolvimento intenso e desenvolvedores alheios interessados em aprender o funcionamento da sua linguagem de programação ainda dependem fortemente da análise direta do código fonte da máquina virtual. Além disso, o conhecimento necessário para entender sua semântica pode ser estranho a quem não é familiarizado com redes de interação ou programação funcional.

1.6 Organização do trabalho

Para iniciar o estudo do funcionamento da HVM3, é necessário que o leitor esteja familiarizado com os conceitos de cálculo-λ, redes de interação, com-

binadores de interação e cálculo de interação. Esses conceitos são apresentados sequencialmente ao longo do capítulo 2.

No capítulo 3, a seção 3.2 apresenta uma submáquina que contempla o modelo de memória, mas essa primeira iteração ainda não é capaz de avaliar nenhuma rede de combinadores de interação. Apenas a partir da seção 3.3, torna-se possível avaliar qualquer rede representável pelo cálculo de interação. Neste ponto, para aqueles que já desejarem implementar um protótipo de submáquina baseando-se nos conceitos apresentados, recomenda-se consultar diretamente a seção 3.10, que demonstra como avaliar completamente uma rede. Embora esta seção utilize conceitos de todas as submáquinas a serem apresentadas, é possível deduzir como realizar o processo de redução completo para redes que utilizem apenas a submáquina 2. As demais seções incorporam novos termos e interações à máquina, em uma ordem que respeita a dependência de funcionalidades entre esses componentes.

O capítulo 4 apresenta as conclusões do autor, destacando os resultados alcançados e sugerindo propostas para trabalhos futuros.

Grande parte da notação de grafos do cálculo de interação, particularmente as células (termos) adicionados pela HVM3, foi criada pelo próprio autor e de forma alguma representa um posicionamento oficial da HOC ou de Yves Lafont. Durante o desenvolvimento, alguns desenvolvedores da HOC forneceram esclarecimentos sobre determinadas funcionalidades da HVM3; contudo, isso não implica que este trabalho seja endossado por eles ou pela HOC.

2 Revisão Bibliográfica

2.1 Cálculo lambda

O cálculo lambda (cálculo-λ), publicado por Church (1932), teve seu início em 1928, quando já existiam outras notações para a abstração de funções; porém, Church foi o primeiro a estabelecer regras formais de conversão para esta notação e a analisar com profundidade as consequências disso.

O cálculo-λ é muito importante para a HVM, pois a tradução de expressões lambda (expressões-λ) para os combinadores de interação integra o processo de compilação de linguagens de alto nível para as instruções da HVM. É mais prático representar elementos de uma linguagem de programação através da notação do cálculo-λ e depois convertê-los para combinadores de interação do que tentar fazer uma compilação direta.

Esta seção apresenta uma revisão breve dos conceitos essenciais do cálculo-λ para a compreensão de sua correspondência com os combinadores de interação. Para aprofundamento e definições formais dos conceitos apresentados a seguir, consulte o livro de Hindley e Seldin (2008).

2.1.1 Notação

O cálculo-λ é um modelo de computação em que as únicas abstrações são funções e suas aplicações, não há formas primitivas para números, booleanos, condicionais ou quaisquer outros elementos comumente disponíveis em linguagens de programação modernas; todas essas estruturas podem ser representadas por meio de funções, conforme será demonstrado posteriormente.

A notação do cálculo-λ caracteriza-se por funções que sempre recebem um único argumento, representadas conforme segue:

$$\lambda a.M \tag{2.1}$$

Em que a é o único argumento e M é o corpo da função. A função identidade $f(x) = x$ pode, por exemplo, ser representada da seguinte forma:

$$\lambda x.x \tag{2.2}$$

E a função $f(x) = x(x)$, isto é, a função que recebe outra função como argumento e aplica-a a si mesma, pode ser representada como:

$$\lambda x.x\,x \tag{2.3}$$

Note que a aplicação de uma função ocorre quando duas variáveis são seguidas uma da outra ($x\,x$), separadas por espaço em vez de parênteses ($x(x)$).

Para representar funções de múltiplos argumentos, pode-se utilizar uma função que recebe o primeiro argumento e retorna outra função que recebe o segundo. Por exemplo:

$$\lambda x.\lambda y.x\,y \tag{2.4}$$

Igualmente:

$$\lambda x.\,(\lambda y.x\,y) \tag{2.5}$$

Por conveniência, é possível representar equivalentemente todos os argumentos de funções aninhadas antes do ponto final da primeira função. Assim, a última função vista também pode ser escrita como:

$$\lambda x\,y.x\,y \tag{2.6}$$

Para fazer a aplicação da definição de uma função a argumentos, utilizam-se parênteses:

$$(\lambda x\,y.x\,y)\,a\,b \tag{2.7}$$

O processo de substituição de variáveis durante a aplicação de funções denomina-se redução beta (redução-β).

$$\begin{aligned}(\lambda x\,y.x\,y)\,a\,b &\to_\beta (\lambda y.a\,y)\,b \\ &\to_\beta (a\,b)\end{aligned} \tag{2.8}$$

As variáveis têm escopo limitado a sua respectiva função. Durante a avaliação de um termo, pode-se renomeá-las para nomes únicos globais, processo denominado equivalência alfa (equivalência-α).

$$\lambda x.(\lambda x.x)\,x =_\alpha \lambda x.(\lambda y.y)\,x \tag{2.9}$$

A conversão eta (conversão-η) é uma transformação que não altera o significado da expressão. Se tivermos uma função que não faz mais nada além de aplicar uma outra função f ao seu argumento, podemos reduzir essa expressão a apenas f, como demonstrado a seguir:

$$\lambda x.f\,x \leftrightarrow_\eta f \tag{2.10}$$

Isso acontece, pois aplicar essa função ou aplicar f resulta sempre no mesmo resultado. Veja o exemplo na sequência:

$$(\lambda x.f\,x)\;a = f\,a \tag{2.11}$$

Podemos dar nomes às funções, mas isso não significa que recursão é permitida.

$$minhaFunc \equiv \lambda x.x\,x\,x \tag{2.12}$$

No exemplo 2.12, $minhaFunc$ não poderia aparecer no corpo da função. Funções nomeadas são uma extensão do cálculo-λ, utilizada neste trabalho para fins didáticos, não integrando a notação original.

Tendo apresentado a notação básica do cálculo-λ, prosseguiremos definindo tipos de dados, operadores e estruturas que programadores usualmente esperam encontrar em qualquer linguagem de programação.

2.1.2 Definições de estruturas e conceitos no cálculo-λ

2.1.2.1 Números

Ao final do seu artigo de 1933, Church introduziu a representação dos números naturais no cálculo-λ, que ficaram conhecidos como números de Church (definidos na expressão 2.13). Esta foi uma das primeiras demonstrações de que toda função computável, incluindo seus valores de entrada (números, conjuntos, etc.), pode ser representada exclusivamente através da notação funcional do cálculo-λ.

$$0 \equiv \lambda f\,x.x,\ Succ \equiv \lambda n\,f\,x.f\ (n\,f\,x)\,,\ n =_\beta \lambda f\,x.\underbrace{f\,(\ldots(f}_{n\text{ vezes}}\,x)\ldots) \tag{2.13}$$

Um número n é representado pela aplicação da função f ao argumento x n vezes.

$$\begin{aligned} 3 &\equiv \lambda f\,x.f\,(f\ (f\,x)) \\ 4 &\equiv \lambda f\,x.f\,(f\ (f\ (f\,x))) \end{aligned} \tag{2.14}$$

Em 2.13, definimos a função $Succ$. Para exemplificar seu funcionamento, vejamos como aplicá-la ao número 2. Expandiremos a definição de 2 posteriormente para evitar confusão.

$$\begin{aligned} 2 &\equiv \lambda f\,x.f\ (f\,x) \\ Succ(2) &= (\lambda n\,f\,x.f\ (n\,f\,x))\,2 \\ &\to_\beta \lambda f\,x.f\ (2\,f\,x) \\ &= \lambda f\,x\,.f\ \left(\overbrace{(\lambda f\,x.f\ (f\,x))}^{2}\ f\,x\right) \\ &\to_\beta^+ \lambda f\,x.f\ (f\ (f\,x)) \\ 3 &\equiv \lambda f\,x.f\ (f\ (f\,x)) \end{aligned} \tag{2.15}$$

A definição dos números pode levantar a questão sobre a sua forma. Na prática, a representação de um número é arbitrária, e não há uma maneira "definitiva" de abordá-la. Todavia, para que uma definição seja útil, é crucial que operações numéricas compatíveis com ela também sejam estabelecidas. Os números de Church são amplamente conhecidos e utilizados porque Church foi o primeiro a definir diversas operações aritméticas e lógicas que funcionam com esta representação. Existem outras formas úteis de representar números, como a proposta por Böhm e Gross (1966), porém estas não serão abordadas neste trabalho.

2.1.2.2 Booleanos

Para definir booleanos, precisamos pensar na sua utilidade prática. Quando um booleano é utilizado, em algum momento ele selecionará um valor em uma estrutura condicional, como um *if then else*.

```
IF [condição] THEN [resultado 1] ELSE [resultado 2]
```

Nesse caso, se a condição é verdadeira, selecionamos o "resultado 1", e, se é falsa, o "resultado 2". Esta lógica pode ser representada através de funções lambda, conforme segue:

$$\begin{aligned} True &\equiv \lambda xy.x \\ False &\equiv \lambda xy.y \end{aligned} \tag{2.16}$$

Como exemplo, vamos representar a função Not, que simplesmente é a negação na lógica.

```
Not = IF condição THEN False ELSE True
```

$$Not \equiv \lambda c.c\ False\ True \tag{2.17}$$

Em que c é uma condição ($True$ ou $False$).

Qualquer estrutura computacional pode ser representada no cálculo-λ, sendo apresentadas progressivamente ao longo do presente trabalho, conforme necessário.

2.2 Redes de interação

As redes de interação, criadas por Lafont (1989), são um modelo de computação notável. Elas se destacam pois máquinas de Turing, cálculo-λ, autômatos celulares e outros sistemas de reescrita de termos podem ser vistos como instâncias especiais de redes de interação. Essas instâncias de modelos computacionais

podem ser implementadas nas redes de interação, mantendo suas propriedades de sequencialidade e paralelismo. Porém, ao representar uma rede de interação em outros modelos, nem sempre essas características são preservadas.

Redes de interação são essencialmente constituídas por 2 elementos:

- **Células**
  Uma célula α possui uma *porta principal* e $n \geq 0$ *portas auxiliares* (ou *aridade*). α pode ser representada de diferentes formas, geralmente círculos, mas nos combinadores de interação que veremos posteriormente, é representada por um triângulo, em que a porta principal está em um dos vértices e as portas auxiliares estão distribuídas ao longo da aresta oposta. Quando a aridade é zero, α é representada por um círculo. As portas são numeradas no sentido anti-horário, sendo a porta principal sempre designada porta 0. É interessante que a porta principal esteja destacada de alguma forma. Ao longo deste capítulo, células terão suas portas principais destacadas com um círculo amarelo, exceto quando fizerem parte do sistema dos combinadores de interação. Veja a Figura 1.

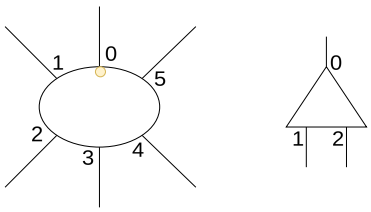

Figura 1 – Células com 5 e 2 portas auxiliares. A célula à direita é um exemplo de célula dos combinadores de interação

- **Fios**
  Um fio liga 2 portas, estejam elas na mesma célula ou não. Caso um fio esteja ligado a apenas uma célula, a única porta em que ele está conectado denomina-se *porta livre*. Veja a Figura 2.

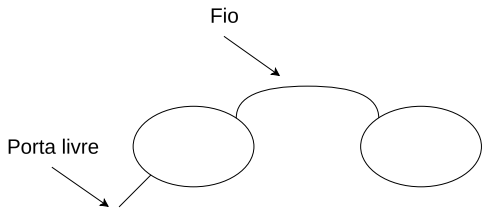

Figura 2 – Células ligadas por fios e uma porta livre

Ao conectar células por fios através de suas portas, formamos uma *rede*. Uma rede não necessariamente tem todas as suas células conectadas.

Quando duas células estão conectadas entre si através de suas portas principais, formam um *par ativo*, ou *expressão de redução* (*redex*). Os *redexes* são importantes pois são alvo de *regras de interação*.

Figura 3 – Células ligadas através de suas portas principais, formando um redex

As regras de interação mostram como reescrever uma rede dado um redex. À esquerda de uma seta ($\rightarrow$), tem-se a rede original; à direita, como deve ser reescrita. Note que as regras são simétricas no sentido de que α_1 interagir com α_2 é o mesmo que a_2 interagir com α_1.

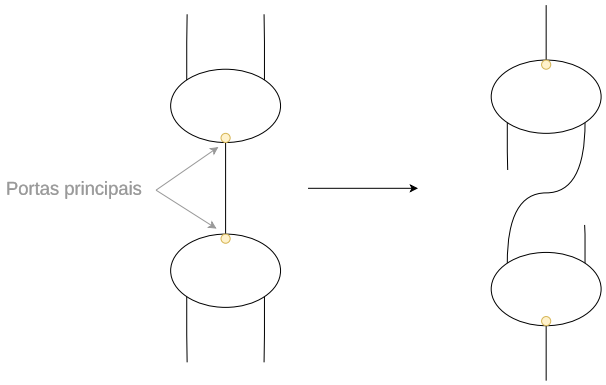

Figura 4 – Regra de interação. Portas principais destacadas com círculo amarelo.

Um *sistema de interação* é um conjunto de regras que podem ser aplicadas sem ambiguidades a uma rede de interação; ou seja, para cada redex há apenas uma regra de interação candidata a ser aplicada. Como as regras de interação são simétricas, um sistema de interação com m células requer no máximo $\frac{m(m+1)}{2}$ regras, que podem ser representadas em uma matriz simétrica parcialmente definida.

$$M = \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 1 & 2 & 3 & 4 & 5 \\ B & 2 & 6 & 7 & 8 & 9 \\ C & 3 & 7 & 10 & 11 & 12 \\ D & 4 & 8 & 11 & 13 & 14 \\ E & 5 & 9 & 12 & 14 & 15 \end{array} \tag{2.18}$$

Na matriz em 2.18, $M_{i,j}$ indica que quando a célula i interagir com a célula j, a regra identificada por r deve ser aplicada, tal que r é o valor armazenado na matriz.

2.2.0.1 Construindo um sistema de interação

Redes de interação podem ser concebidas como uma linguagem de programação. Vejamos um exemplo de implementação de uma lista encadeada.

Uma das abordagens mais comuns para implementar uma lista encadeada envolve uma estrutura com dois ponteiros: um apontando para a cabeça da lista, e outro que pode apontar para a cauda da lista ou para Nil, que representa o fim da lista. Em Haskell, podemos fazer isso da seguinte forma:

```haskell
-- Definição do tipo da lista encadeada
data Lista elemento = Nil | Cons elemento (Lista elemento)
```

Para acrescentar uma lista à outra, basta que troquemos Nil de uma delas pelo início da outra lista. Podemos fazer isso percorrendo recursivamente a cauda da lista até chegar ao Nil, e então substituí-lo.

```haskell
-- Concatena duas listas
append :: Lista a -> Lista a -> Lista a
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Para representar isso em redes de interação, precisaremos de duas células para definir a estrutura da lista: uma que representa Nil, com apenas uma porta para se conectar à cauda de outras listas; e outra que representa Cons, com uma porta principal para interagir com operadores e outras duas portas para apontar para um elemento e para a cauda.

A operação Append será igualmente representada por uma célula com 3 portas: duas delas apontando para as listas que queremos unir e uma terceira ligando a lista resultante ao resto da rede. Com isso, podemos definir as regras de interação de Append semelhantemente à definição em Haskell. O caso recursivo do append pode ser observado na Figura 5 e, o caso base na Figura 6. Assim, juntar duas listas é apenas uma questão de uni-las através de uma célula Append e seguir as regras de interação, conforme exemplificado nas Figuras 7, 8 e 9.

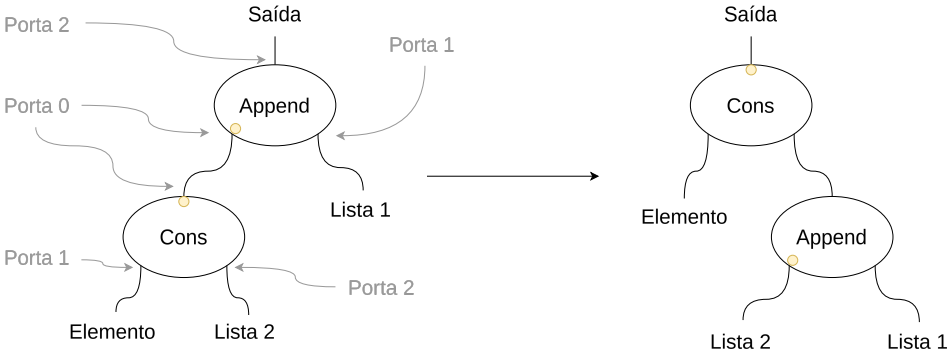

Figura 5 – Regra de interação da operação Append em uma lista encadeada (caso recursivo)

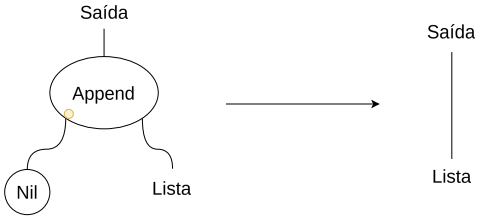

Figura 6 – Regra de interação da operação Append em uma lista encadeada (caso base)

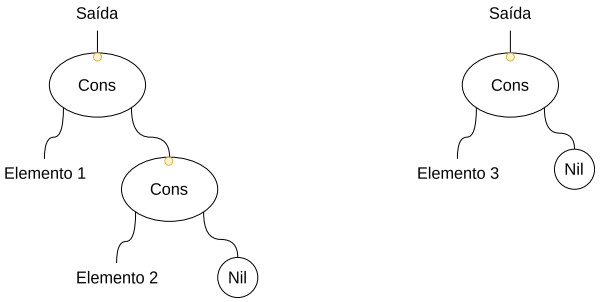

Figura 7 – Listas a serem unidas

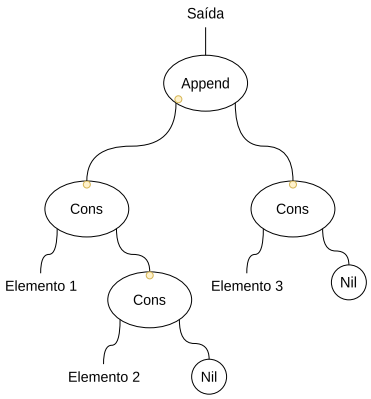

Figura 8 – Redex formado através de célula Append

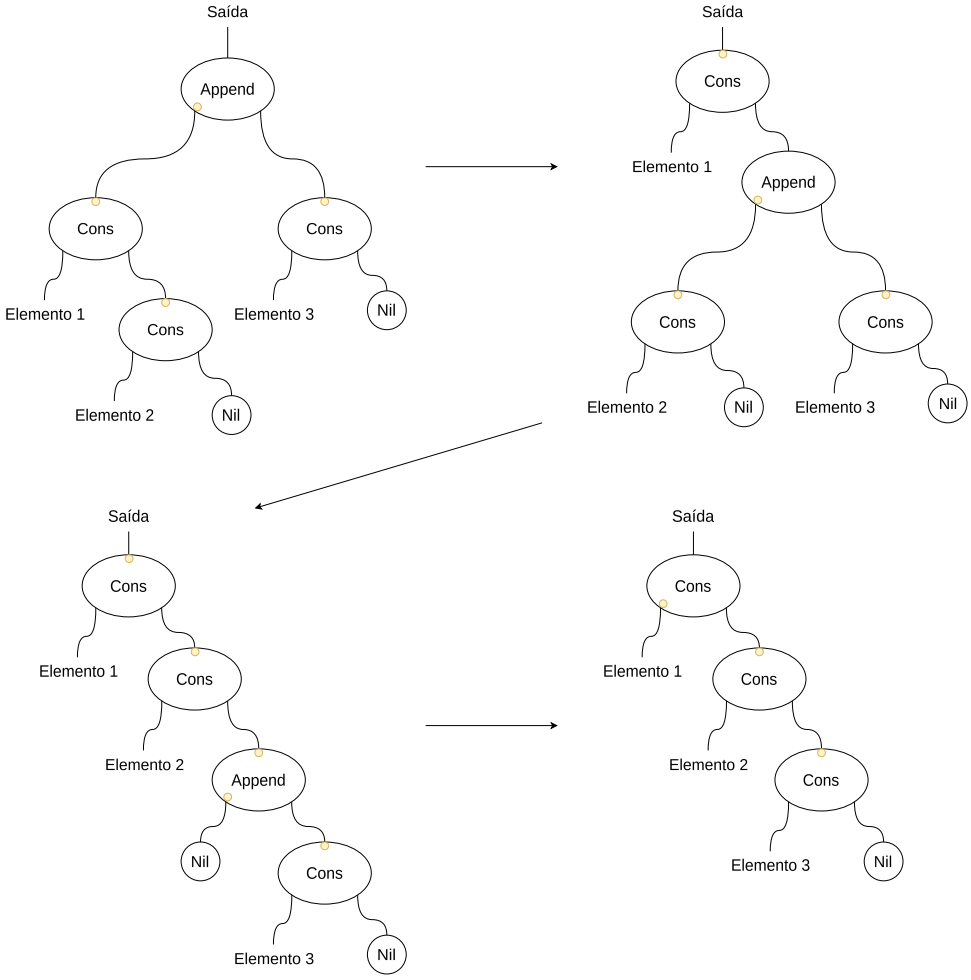

Figura 9 – Redução da rede da Figura 8 até que ambas as listas sejam unidas em uma única

2.2.1 Confluência forte

A confluência forte é uma propriedade que torna a execução massivamente paralela da HVM possível. Se temos uma rede μ com dois ou mais redexes, temos uma escolha a fazer: qual redex reduzir primeiro?

A confluência forte (Figura 10) estabelece que, se μ pode ser reduzido para v_1 ou v_2 em um único passo, ambos v_1 e v_2 podem ser reduzidos em um passo para uma mesma rede ε. Em outras palavras, não importa a ordem em que os redexes são executados, o resultado sempre **pode** ser o mesmo. Além disso, é uma certeza que o resultado será o mesmo caso a redução continue até uma rede irredutível (ver Figura 11). Uma consequência disso é que redes de combinadores de interação possuem no máximo uma forma normal. Diferente do cálculo-λ, não precisamos de uma estratégia de redução para normalizar uma expressão com o menor esforço possível pois, no contexto das redes de interação, qualquer ordem de redução exige a mesma quantidade de trabalho.

Pensando na implementação da HVM, cada redex pode ser executado em um fio de execução diferente com pouquíssima necessidade de sincronização entre eles.

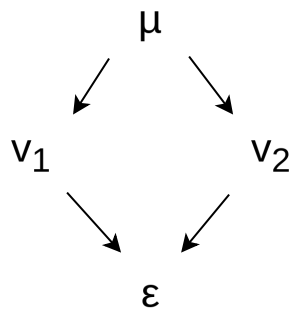

Figura 10 – Confluência forte

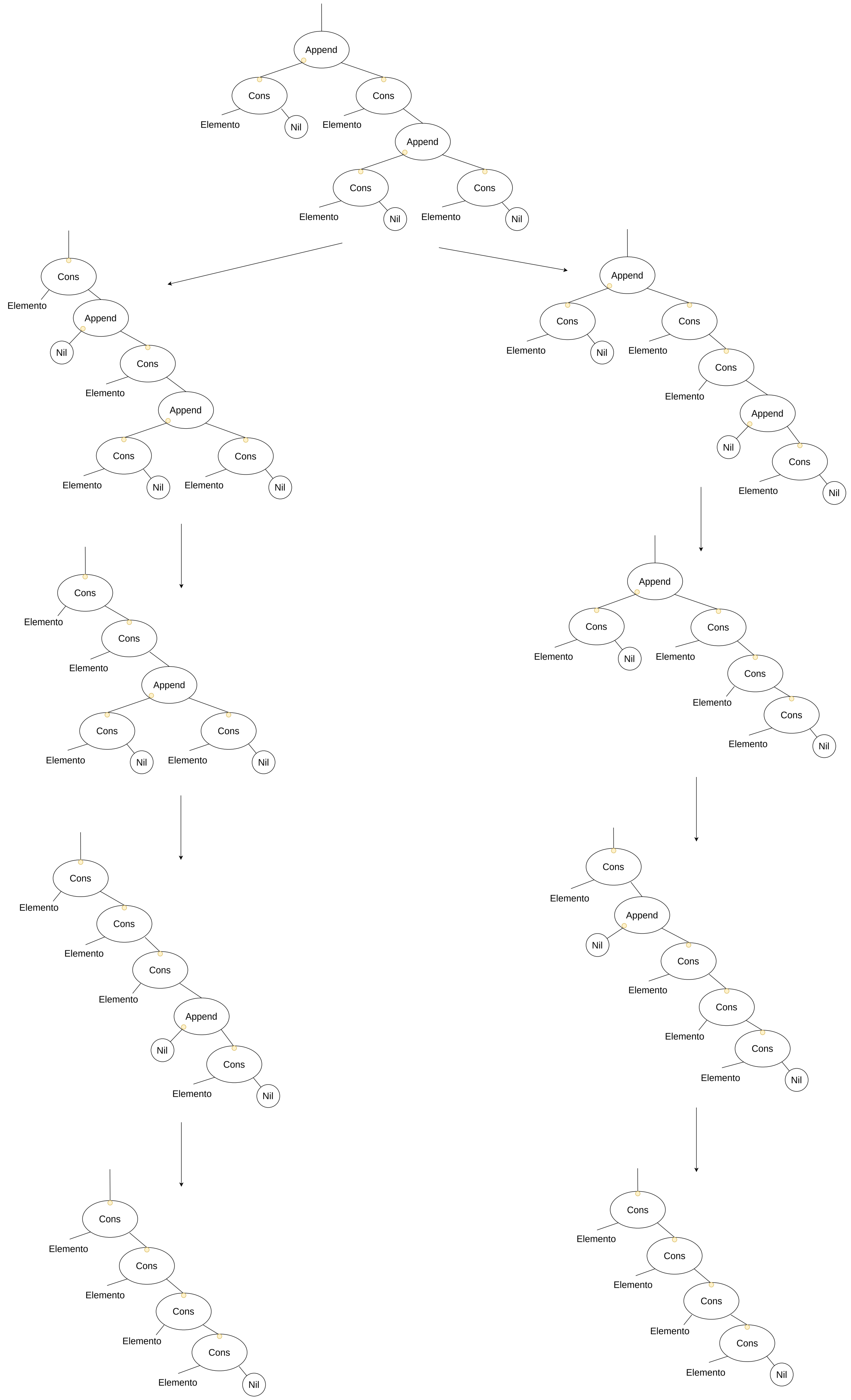

Figura 11 – Exemplo de uma rede reduzindo através de sequências diferentes para o mesmo resultado, na mesma quantidade de passos

2.3 Combinadores de interação

Os combinadores de interação, também propostos por Lafont (1997), são um sistema de interação específico das Redes de Interação que formam um modelo universal de computação distribuída, isto é, qualquer outra rede de interação pode ser representada utilizando apenas as células e regras desse sistema, conforme explicado na página 82 do mesmo artigo que os introduziu.

Os combinadores de interação compreendem essencialmente três tipos de célula:

Construtores (γ) Têm aridade 2, representados por um triângulo branco.

Duplicadores (δ) Têm aridade 2, representados por um triângulo preto.

Apagadores (ε) Têm aridade 1, representados por um círculo.

As regras de interação na Figura 12 são classificadas de duas maneiras: se ambas as células são do mesmo tipo, há uma interação de *aniquilação*, isto é, células serão apagadas; caso contrário, há uma *comutação*, isto é, células serão criadas.

Essas células e regras de interação são bastante abstratas, e não é intuitivo tentar entendê-las isoladamente. Portanto, seus significados e a interpretação das interações serão explicados juntamente com a correspondência desse sistema com o cálculo de interação ao longo das seções 2.3.1 e 3.3.1.

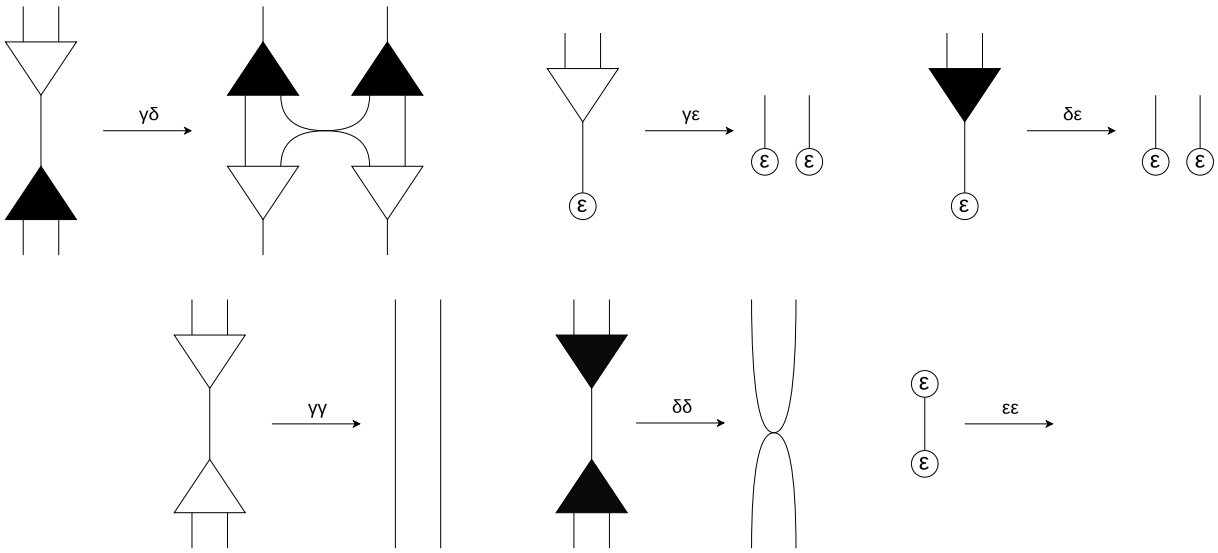

Figura 12 – Regras de interação dos combinadores de interação

2.3.1 Cálculo de interação

A HVM é uma intérprete de cálculo de interação (IC), um sistema de reescrita mínimo inspirado pelo cálculo-λ, criado pelo próprio Taelin (2025), mas com algumas diferenças que o tornam inerentemente mais eficiente. Particularmente:

- **Variáveis são afins:** Elas podem ser lidas no máximo uma vez.
- **Variáveis são globais:** Elas podem ocorrer em qualquer lugar do programa, sem restrição de escopo.

Embora toda expressão do cálculo lambda seja diretamente representável nos combinadores de interação, nem toda rede de combinadores de interação admite representação direta na notação do cálculo-λ. Em contraste, o cálculo de interação faz uma correspondência direta entre cada tipo de célula e os termos da sua própria notação, levando também em consideração o "contexto" das células (ou polaridade, que será explicado na seção 2.3.1.1), o que o permite representar qualquer rede de combinadores de interação.

Assim, estabelece-se a possibilidade de tradução bidirecional: tanto de IC para combinadores de interação quanto de combinadores de interação para IC. Porém, esse processo é bem mais simples se usarmos a variante "Combinadores de interação simétricos", descrita por Mazza (2007). A única diferença entre os combinadores de interação e os combinadores de interação simétricos é a regra de interação γγ, que Mazza renomeia como "ζζ" para acentuar a diferença. Sua nova definição é ilustrada na Figura 13.

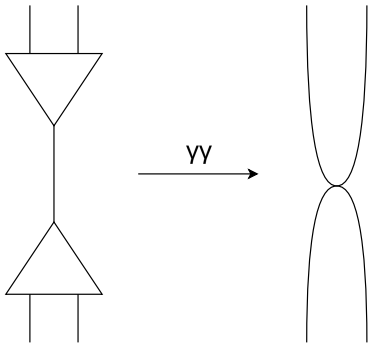

Figura 13 – Regra γγ (ou ζζ) para combinadores de interação simétricos

A partir deste ponto, sempre que os combinadores de interação forem citados, estaremos, na verdade, nos referindo à sua variante simétrica. Assim, podemos traduzir o cálculo-λ conforme a Figura 14.

Funções lambda serão transformadas em um γ, em que a porta 1 representa o argumento que a função recebe e, a porta 2, o corpo. Na Figura 15 é evi-

denciada a função identidade, que liga suas portas 1 e 2 diretamente, já que o argumento é retornado assim como foi recebido. A porta principal de uma lambda pode se conectar à porta principal de um δ para que a função inteira seja duplicada, ou conectar-se à porta principal de uma célula de aplicação, também representada por γ, para formar um redex equivalente à aplicação de uma função a um argumento.

Uma célula de aplicação é ligada a um argumento na sua porta 1, que será aplicado a uma função lambda conectada a sua porta principal. O resultado da computação será ligado à porta 2 através da interação γγ. Observe na Figura 16 que o argumento "1" é ligado a "saída" após a interação da aplicação com a função identidade.

Caso um argumento seja utilizado mais de uma vez no corpo de uma função, é necessário duplicá-lo com um δ, conforme ilustrado na Figura 17.

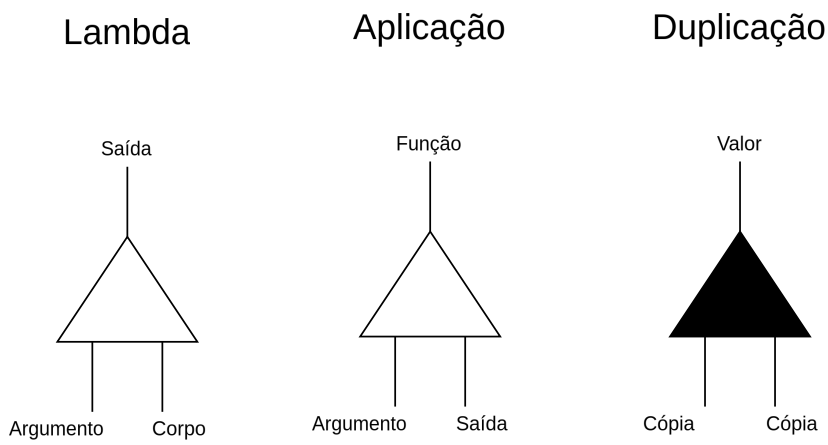

Figura 14 – Correspondência entre cálculo-λ e combinadores de interação

Com essa abordagem, podemos utilizar as regras de interação para computar qualquer expressão do cálculo-λ com a seguinte restrição: funções que duplicam seus argumentos não podem ser duplicadas. Por exemplo, a expressão "$(\lambda x.add\, x\, x)\, 2$" até pode ser inicialmente representada nos combinadores de interação, mas não é possível reduzi-la corretamente até sua forma normal (lembre-se que a representação de 2 em números de Church duplica seu argumento).

Para entender por que isso acontece, vamos analisar a interação γδ na Figura 18. Repare que ela gera dois termos δ, em que ao considerar as portas 1 das células δ, apenas a lambda conectada a "b" estaria conectada ao corpo, e ao considerar as portas 2, apenas a lambda conectada a "a" estaria conectada ao corpo. Assim, estamos compartilhando o mesmo corpo para duas aplicações diferentes da mesma função. As células δ duplicarão incrementalmente o corpo e, eventualmente se encontrariam para duplicarem uma a outra, o que não pode acontecer,

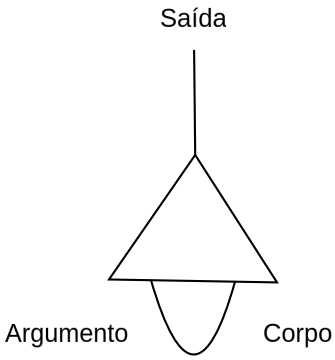

Figura 15 – Exemplo: representação de $\lambda x.x$ em combinadores de interação.

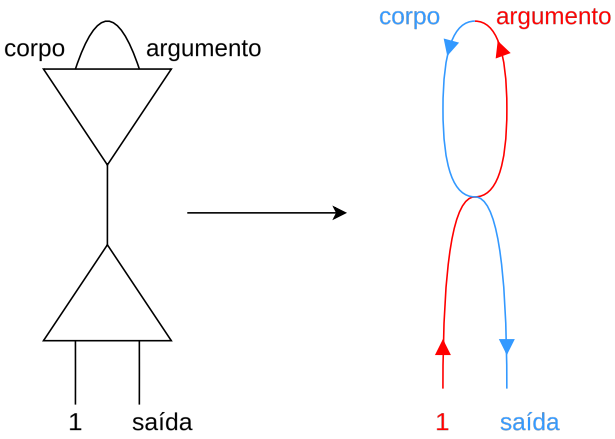

Figura 16 – Interação da aplicação do número 1 à função identidade

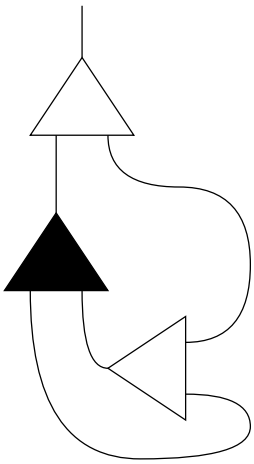

Figura 17 – Exemplo: representação de $\lambda x.xx$ em combinadores de interação.

e por isso a aniquilação δδ é correta ao eliminar as células e somente ligar os fios remanescentes. Porém, quando o corpo é uma outra função que duplica seu argumento, isto é, que possui uma célula δ, precisaríamos duplicar também tal célula δ, o que não acontece em δδ.

Essa é uma limitação muito significativa, portanto precisamos contorná-

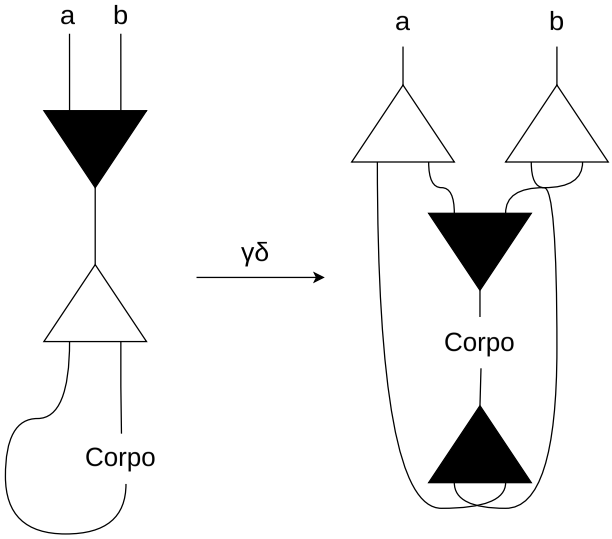

Figura 18 – Interação γδ

la. Para tanto, adotaremos a mesma abordagem descrita por Asperti e Guerrini (1999) para alterar a interação δδ e fazê-la levar em consideração o *nível* em que cada δ ocorre. Podemos entender "nível" como a maneira que uma célula foi criada na rede. Note que a única situação em que queremos uma aniquilação entre células δ é quando ambas foram geradas a partir da mesma interação γδ. Assim, quando uma interação γδ ocorrer, marcaremos, ambas as células δ com uma mesma etiqueta (*label*) única na rede. Assim, podemos dizer que δδ é uma aniquilação se as células tiverem *labels* iguais, ou uma comutação, se elas tiverem *labels* diferentes. Observe a nova definição de δδ na Figura 19.

Com a adição dos *labels*, podemos computar uma quantidade bem maior de termos, mas, ainda sim, há uma outra limitação: Se uma função-λ que duplica seu argumento é duplicada, então seus clones não podem duplicar um ao outro. Por exemplo, a expressão $(\hat{}\,2\,2) \equiv (\lambda a.(a\,a))\,(\lambda f\,x.f\,(f\,x))$ seria reduzida erroneamente com o sistema que temos até agora. Porém, contornar essa limitação envolve um *overhead* muito grande, sendo necessária a introdução de novas células no sistema e um algoritmo semelhante a um *book-keeping*, também descritos por Asperti e Guerrini (1999). Por isso, o IC não suporta essas expressões do cálculo-λ, assim permitindo a computação de um subconjunto dele. A decisão de não suportar todo o cálculo-λ foi tomada pois expressões desse tipo não são comuns, e sempre é possível reescrevê-las de outra forma que evite essa sequência de duplicações.

Com isso, já conseguimos representar termos do cálculo lambda nos combinadores de interação, e tornar um termo lambda irredutível é equivalente a tornar uma rede de interação irredutível. Por fim, uma grande parte do IC é definir

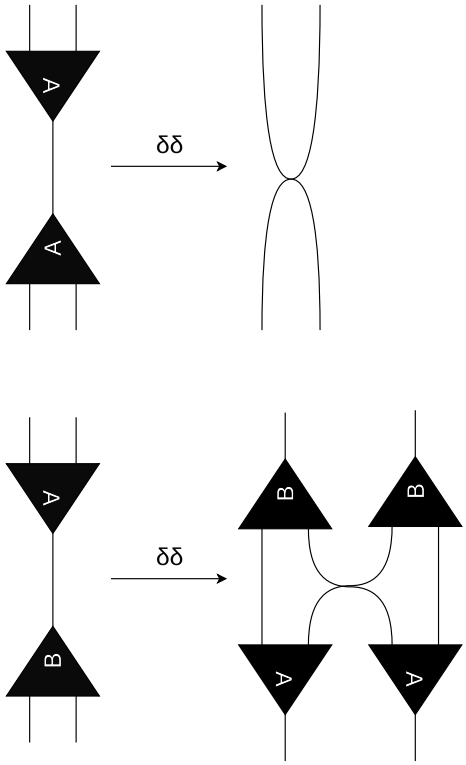

Figura 19 – Interação δδ levando *labels* em consideração

uma notação semelhante a do cálculo-λ, para que possamos representar os combinadores de interação de forma textual. Vamos nomear as diferentes estruturas apresentadas anteriormente na Figura 14, acompanhadas de como representá-las em forma de texto através de uma gramática:

- **VAR** $Nome$ representa uma variável (fio).
- **ERA** $*$ representa um apagador. (ε)
- **LAM** $\lambda Nome.Termo$ representa um termo lambda. (γ)
- **APP** $(Termo\ Termo)$ representa uma aplicação. (γ)
- **SUP** $\&Label\{Termo, Termo\}$ representa uma superposição (que será explicada ainda nesta seção). (δ)
- **DUP** $!\&Label\{Name, Name\} = Term;$ representa uma duplicação. (δ)
- $Termo ::= VAR|ERA|LAM|APP|SUP|DUP$
- $Nome$ é uma string qualquer que não contenha espaços ou caracteres utilizados para definir os termos anteriores.
- $Label$ é um valor numérico.

Tal como uma célula γ pode corresponder a uma lambda ou uma aplicação, faremos uma célula δ corresponder a uma duplicação ou uma **superposição**. Na interação γδ são geradas duas células δ e, enquanto uma tem a semântica de duplicar o corpo da função, a outra tem a semântica de receber dois argumentos e aplicá-los ao mesmo corpo. A primeira será chamda de DUP, e a segunda será chamada de SUP. Veja a Figura 20.

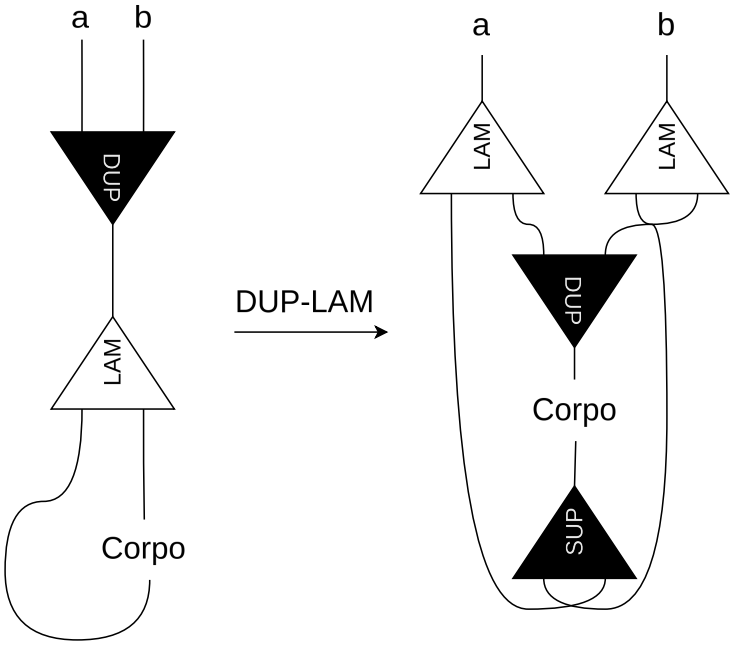

Figura 20 – Duplicação de uma lambda gerando um DUP e um SUP

Na figura 20, foram adicionados indicadores dentro das células para que o tipo de cada uma fique claro, mas, na prática, isso é desnecessário, pois conseguimos inferir os tipos, assim como será descrito na seção 2.3.1.1.

Já podemos evidenciar uma vantagem do IC sobre outras linguagens puramente funcionais de avaliação preguiçosa: a facilidade em fazer avaliação ótima. Observe, no exemplo a seguir, como a avaliação preguiçosa pode ser utilizada para fazer avaliação ótima de algumas expressões.

```
1  // A variável ‘x‘ é utilizada duas vezes no corpo da função.
2  fn foo(x) {
3    return x + x
4  }
5
6  // Chamar ‘foo‘ com o argumento ‘2 * 2‘ avaliado preguiçosamente
7  // resulta em
8  foo(2 * 2)
9  => {
10   // Trabalho duplicado terá que ser realizado!
11   return (2 * 2) + (2 * 2)
12 }
13
14 // Uma maneria de resolver isso seria transformando ‘x‘ em um
```

```
// ponteiro...
foo(2 * 2)
=> {
  p = (2 * 2) // ponteiro (análogo a DUP)
  return p + p
}
=> {
  p = 4
  return p + p
}
=> {
  return 4 + 4 // Ótimo!
}

// Mas isso ainda não funciona quando uma função é quem está
// sendo duplicada! Observe o caso abaixo:
fn foo(f) {
  p = f
  // ambos ‘a -> a‘ e ‘b -> b‘ são a função identidade.
  return p(a -> a) + p(b -> b)
}

foo((c -> c(2 * 2)))
=> {
  p = (c -> c(2 * 2))
  return p(a -> a) + p(b -> b)
}
// no código acima, não podemos avaliar ‘(2 * 2)‘, pois não
// sabemos se ele será utilizado no corpo de ‘c‘. Se ele não for
// utilizado, seria trabalho desperdiçado (avaliação não ótima).
// Assim, ‘p‘ já está apontando para uma expressão que não pode
// ser reduzida. Isso resultará em:
=> {
  return (c -> c(2 * 2))(a -> a) + (c -> c(2 * 2))(b -> b)
}
=> {
  return (a -> a)(2 * 2) + (b -> b)(2 * 2)
}
=> {
  return (2 * 2) + (2 * 2) // Trabalho duplicado!
}
```

Conforme pôde ser visto no pseudocódigo anterior, a estratégia de ponteiros ainda falha quando uma função é o argumento que está sendo duplicado. Com a introdução da superposição, podemos decompor o corpo de uma função em unidades menores e compartilhá-las entre diferentes chamadas, conforme feito na sequência:

```
fn foo(f) {
  p = f
  return p(a -> a) + p(b -> b)
}

// Em tempo de execução, quando um ponteiro (DUP) interage com
// uma lambda (LAM), transformaremos a lambda em uma aplicação
// de função, que nos permitirá tratar melhor o termo ‘2 * 2‘.
// Assuma que todas as variáveis têm escopo global.
foo((c -> c(2 * 2)))
```

```
11 => {
12   p = (c -> c(2 * 2))
13   return p(a -> a) + p(b -> b)
14 }
15 => {
16   // Abaixo, estamos aplicando tanto a função c0, quanto a
17   // função c1 ao mesmo argumento `2 * 2`.
18   p = {c0 c1}(2 * 2) // O par `{c0 c1}` forma uma superposição
19   return (c0 -> p)(a -> a) + (c1 -> p)(b -> b)
20 }
21 // Podemos criar uma regra de interação para que, quando uma
22 // superposição interagir com uma aplicação, seja criado um
23 // ponteiro `v`. Também precisaremos diferenciar a 1ª e a
24 // 2ª ocorrência de `p`.
25 => {
26   v = (2 * 2)
27   p = {c0 c1}(v)
28   return (c0 -> p)(a -> a) + (c1 -> p)(b -> b)
29 }
30 => {
31   v = (2 * 2)
32   p0 = c0(v)
33   p1 = c1(v)
34   return (c0 -> p0)(a -> a) + (c1 -> p1)(b -> b)
35 }
36 // Agora basta seguir o fluxo natural de redução, a começar
37 // pelas aplicações na linha 34.
38 => {
39   v = (2 * 2)
40   p0 = (a -> a)(v)
41   p1 = (b -> b)(v)
42   return p0 + p1
43 }
44 => {
45   v = (2 * 2)
46   p0 = v
47   p1 = v
48   return p0 + p1
49 }
50 => {
51   v = (2 * 2)
52   return v + v
53 }
54 => {
55   v = 4 // Avaliado uma única vez!
56   return v + v
57 }
58 => {
59   return 4 + 4
60 }
```

A ideia é que, enquanto uma duplicação (ponteiro) recebe um valor e produz duas cópias, a superposição recebe dois valores e os aplicam a uma mesma rede. Em outras palavras, se tenho um par de números e uma função que aceita como entrada um único número, através da superposição posso executar tal função com ambos os números do par ao mesmo tempo.

Assim, assumiremos o seguinte: toda célula δ que faz parte de um redex

com uma lambda é uma célula de duplicação. Quando essa interação γδ acontecer, serão geradas duas células δ: uma delas será uma superposição e, a outra, uma duplicação. Analogamente, assumiremos que toda célula δ que faz parte de um redex com uma aplicação é uma célula de superposição. Essas afirmações podem ser feitas com segurança em virtude do conceito de polaridade.

Com isso, já podemos representar uma rede de interação em forma textual. Repare, no exemplo abaixo o uso de "x" como uma variável global, que permite utilizá-la antes do LAM que faz sua ligação.

```
1 λt.((t x) λx.λy.y)
```

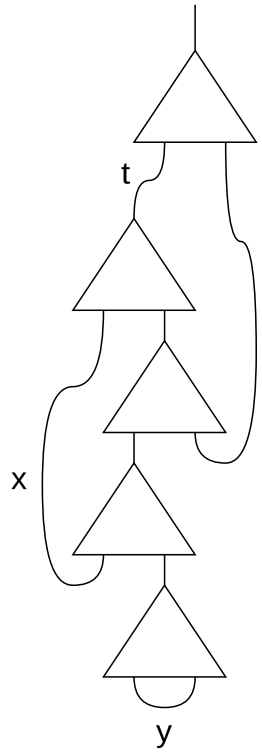

Figura 21 – λt.((t x) λx.λy.y) representado em um grafo

Apesar desse comportamento atípico, nada disso contradiz a teoria dos combinadores de interação, e podemos desenhar a rede equivalente sem problemas (veja Figura 21).

O cálculo de interação também possui uma representação textual para as regras de interação, mas elas serão evidenciadas apenas na seção 3.3.1.

2.3.1.1 Traduzindo combinadores de interação para cálculo de interação

O processo de tradução inverso, de combinadores de interação para cálculo de interação, é um tanto mais complexo, especialmente se considerarmos que a

representação das redes na memória HVM é feita de forma semelhante à estrutura de árvore, no sentido de que só há garantia de percorrê-las em uma única direção. Além disso, sem processamento adicional, surge uma ambiguidade, pois células construtoras podem ser entendidas tanto como abstrações lambda quanto como aplicações de funções, e células duplicadoras podem representar a duplicação ou a superposição de valores. Para eliminar essa ambiguidade, precisaremos, primeiro, entender o que é *polaridade* e como percorrer uma rede de combinadores de interação associando uma polaridade a cada porta, pois isso será utilizado pelo próximo algoritmo que veremos, que é o mesmo utilizado pela linguagem Bend, desenvolvida pela HOC (2024, 2025a).

Uma polaridade pode ser positiva ($+$) ou negativa ($-$). Semanticamente, uma porta $+$ representa a produção de um valor e, uma porta $-$, o consumo de um valor. Uma porta $+$ só pode se conectar a uma porta $-$ e vice-versa.

Regras de polaridade:

- A polaridade de uma porta livre é $-$.
- A polaridade de γ é ou $+(+-)$ ou $-(-+)$, isto é, a porta 0 e porta 1 têm a mesma polaridade, e a porta 2 tem polaridade oposta.
- A polaridade de δ é ou $-(++)$ ou $+(--)$.
- A polaridade de ε é $+$.

Com isso, se tivermos a árvore de uma rede com uma porta livre, já podemos definir a polaridade de todas as suas portas, partindo da porta livre (raiz da árvore). O exemplo 22 mostra a mesma rede da Figura 17, porém com as portas polarizadas.

A polaridade, somada às seguintes regras, já é suficiente para classificar toda rede de combinadores de interação representáveis por árvore.

Regras de correspondência:

- **LAM:** γ com polaridade $+(+-)$.
- **APP:** γ com polaridade $-(-+)$.
- **DUP:** δ com polaridade $-(++)$.
- **SUP:** δ com polaridade $+(--)$.
- **ERA:** ε com polaridade $+$.

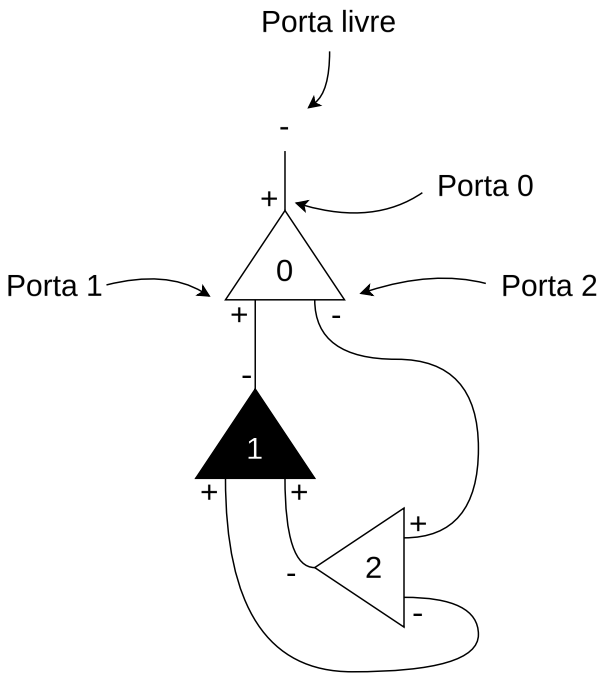

Figura 22 – Rede equivalente a $\lambda x.xx$ polarizada.

Na figura 22 é mostrada a rede de interação polarizada equivalente à expressão do cálculo-λ $\lambda x.x\,x$, tal que:

1. α_0 é um γ com polaridade $+(+-)$, portanto temos um $\lambda a.b$ (LAM).
2. α_1 é um δ com polaridade $-(++)$ e sua porta 0 (DUP) está ligada à porta 1 de α_0 (a), portanto o argumento do LAM será duplicado no corpo da função.
3. α_2 é um γ com polaridade $-(-+)$, portanto temos $(c\ d)$ (APP). Suas portas 0 e 1 estão ligadas às portas 1 e 2 de α_1, portanto $c = d = a$ e assim temos $(a\ a)$. Como a porta 2 de α_2 está ligada à porta 2 de α_0 (b), o resultado da aplicação será associado ao corpo do LAM ($b = (a\ a)$).
4. Por fim, temos $\lambda a.(a\ a)$.

A dificuldade surge quando tentamos polarizar uma rede com redexes. Ainda percorremos a árvore partindo da raiz e indo em direção às folhas; no entanto, pode ser que a porta da raiz não seja livre, o que já é o suficiente para que não consigamos definir nenhuma polaridade com o que sabemos até agora. Por exemplo, a partir da Figura 23, podemos determinar a polaridade de todas as portas da árvore 1, mas não há um ponto de partida que permita definir as polaridades das portas das árvores 2 e 3.

Vejamos a extensão do algoritmo para lidar com esse caso. Naturalmente, o primeiro passo é repetir o que aprendemos até agora e definir a polaridade de

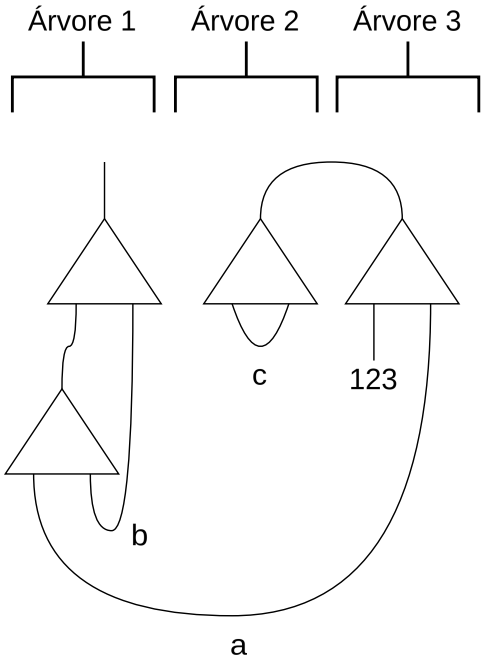

Figura 23 – Exemplo de rede com redex

todas as possíveis portas a partir das portas livres da rede, conforme foi feito na Figura 24.

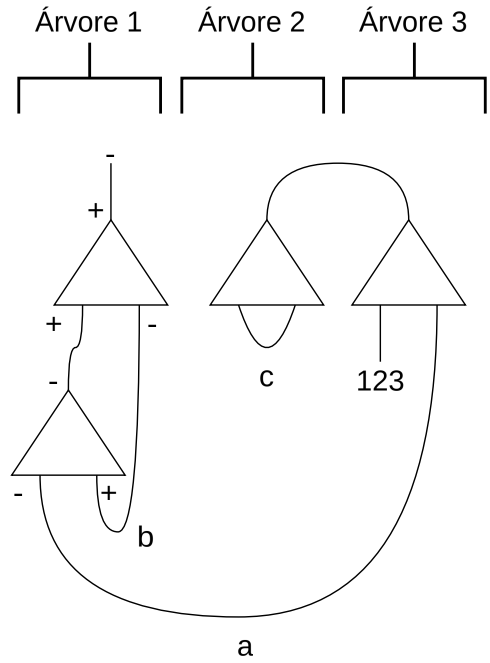

Figura 24 – Polarização da árvore com porta livre

Então percorremos as demais árvores normalmente, mas associando às portas variáveis em vez de polaridades. Para duas variáveis nomeadas com mesma letra, sua versão maiúscula é o contrário da sua versão minúscula, isto é, se Q é $+$,

então q é $-$, e vice-versa. Na figura 25, primeiro percorremos a árvore 2, depois, a 3. Note que, ao percorrer a árvore 3, o algoritmo identifica que a raiz já tem uma polaridade associada a sua outra extremidade, portanto utiliza o oposto dela.

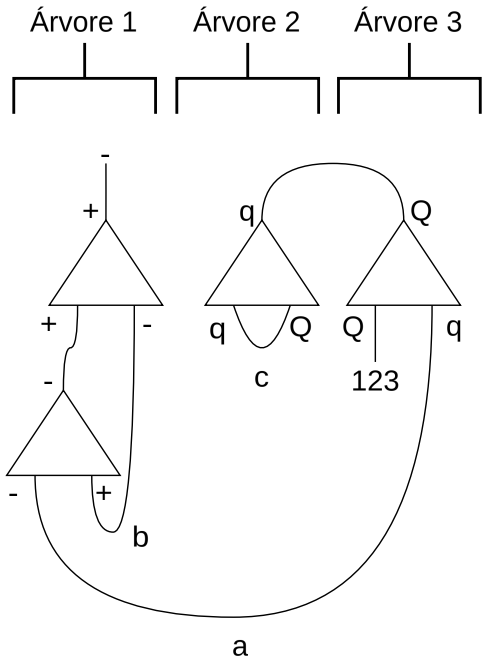

Figura 25 – Polarização com variáveis das árvore sem portas livres

Enquanto percorremos a árvore 3, notamos que a porta 2 com a variável q está conectada a uma porta de polaridade $-$ da árvore 1, assim podemos inferir que $q = +$ e $Q = -$. Com isso, sabemos a polaridade da rede inteira, mostrada na Figura 26.

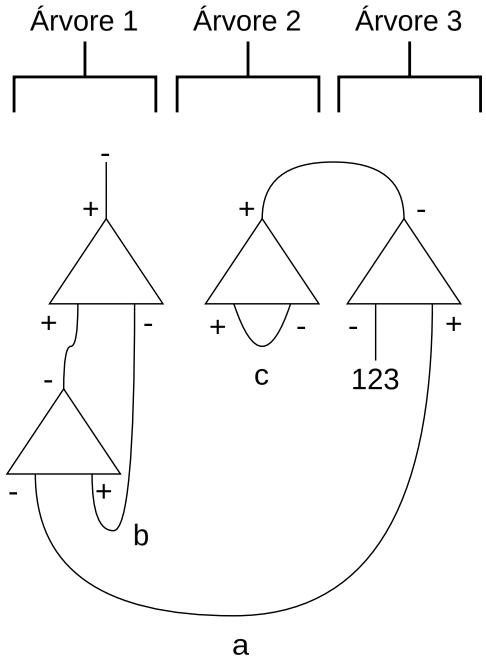

Figura 26 – Rede com redex com todas as portas polarizadas

2.3.2 Limitação da avaliação ótima na HVM3

A confluência forte garante que avaliar uma rede até que ela se torne irredutível sempre demanda a mesma quantidade de trabalho. Uma consequência disso é que não há uma forma de avaliação melhor ou pior que a outra, ou seja, a avaliação é sempre ótima nas redes de interação.

Porém, quando uma função lambda descarta seu argumento no IC, pode significar que a rede de interação correspondente se tornará desconexa, isto é, seja dividida em duas partes que não estão conectadas entre si, assim como pode ser observado na Figura 27. A confluência forte é uma propriedade que leva em consideração toda a rede, mas, sob a perspectiva do IC, interessa-nos apenas a parte da rede ligada à porta de saída, pois computar redexes fora dela seria trabalho desperdiçado. Por esse motivo, a HVM3 possui duas versões: a preguiçosa, que sempre avalia apenas a rede conectada à porta de saída e garante avaliação ótima; e a ansiosa, que resolve todos os redexes da rede em paralelo, mesmo se ela for desconexa.

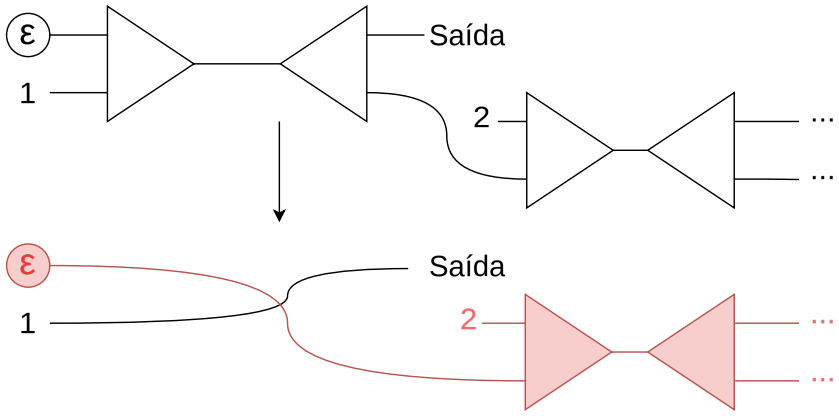

Figura 27 – Exemplo da interação "(λx.1) (+ 2 2)", que gera uma rede desconexa. Computar a rede em vermelho seria trabalho desperdiçado. Parte da rede equivalente a "(+ 2 2)" não está representada para simplificar o exemplo.

2.4 Trabalhos Relacionados

2.4.1 Warren's Abstract Machine: A Tutorial Reconstruction

Warren's Abstract Machine: A Tutorial Reconstruction (1991) é uma obra escrita por Hassan Aït-Kaci com foco em didática e, apesar de técnico, tem estilo acessível, sendo uma ótima referência para moldar a progressão da explicação do processo de construção de um compilador. A WAM do Prolog é reimplementada várias vezes ao longo do livro, começando com versões básicas da máquina compatíveis apenas com um pequeno subconjunto da linguagem e, progredindo passo a passo, esclarece como a linguagem lógica pode ser compilada e executada de maneira eficiente, até que todos os recursos nativos da linguagem sejam implementados. Esse livro é uma obra importantíssima para os interessados no Prolog, já que muitas das implementações modernas da linguagem são baseadas na WAM. Para esta monografia, a metodologia do livro é o elemento de maior interesse.

2.4.2 The Mechanical Evaluation of Expressions

A obra de Peter Landin, *The Mechanical Evaluation of Expressions* (1964), introduz a máquina SECD, um modelo abstrato para a avaliação de expressões no cálculo lambda. Este trabalho não apenas fundamentou a implementação de linguagens funcionais, como também influenciou profundamente o design de arquiteturas computacionais. O artigo, semelhante a esta monografia, descreve as estruturas de dados, modelo, memória e instruções de uma máquina virtual (SECD), que interpreta expressões lambda como programas de computador. O método

baseia-se na tradução sistemática de expressões para uma forma intermediária, que é então processada por essa máquina abstrata. Landin ilustra o funcionamento da máquina com exemplos práticos, mostrando seu comportamento passo a passo.

2.4.3 The Vine Programming Language

O Vine (2025) é uma linguagem de alto nível que é compilada para a máquina virtual Ivy. Tanto o Ivy quanto a HVM são *runtimes* experimentais baseados em Interaction Nets, que visam otimizar a execução de programas e fornecer paralelização facilitada. Apesar de compartilharem a mesma base teórica, suas implementações são distintas. O Ivy utiliza *Stacked Flow Graphs* (SFGs), projetado para representar o fluxo de controle de funções de maneira mais eficiente e paralelizável do que os tradicionais *Control Flow Graphs* (CFGs). Essa estrutura permite otimizações antes que a representação em redes de interação seja gerada.

2.4.4 HINet: Interaction Nets in Haskell

O HINet (2015) é uma implementação altamente concorrente de redes de interação em Haskell, desenvolvida por Wolfram Kahl na Universidade McMaster. A proposta central do HINet é explorar o paralelismo intrínseco das redes de interação, utilizando o GHC (Glasgow Haskell Compiler) como *runtime*, capaz de gerenciar milhões de pequenas *threads*. Ele executa arquivos .inet, escritos em uma versão restrita da linguagem Inets de Ian Mackie. O HINet implementa um paralelismo de granularidade extremamente fina para paralelização. Isso significa que, ao executar programas em múltiplos núcleos, o aumento de desempenho pode ser limitado; por exemplo, ao aumentar os recursos de um programa para quatro núcleos, o aumento de desempenho pode ser inferior a quatro vezes.

3 Desenvolvimento

3.1 As versões da HVM

Semelhante a como foi feito por Aït-Kaci (1991) com a linguagem de programação Prolog, o desenvolvimento da HVM3 será um processo incremental, realizado através da implementação de submáquinas. Cada nova versão, ou submáquina, agregará funcionalidades à versão anterior, começando do zero e culminando em uma máquina completa e funcional. Esta abordagem passo a passo visa a facilitar a compreensão do projeto, garantindo uma sequência de introdução de funcionalidades, da mais básica à mais avançada, exceto quando uma funcionalidade simples depende de outra mais complexa.

No momento da escrita deste trabalho, a versão de avaliação ansiosa (*strict*) da HVM3 ainda não está em um estado tão avançado quanto a de avaliação preguiçosa, portanto, as submáquinas serão baseadas nesta versão. Na Tabela 1 são comparadas as versões da HVM3, evidenciando conceitos da versão ansiosa que não serão detalhados nesta monografia.

Característica	HVM3 *Lazy*	HVM3 *Strict*
Paralelismo	Não faz execução paralela para que seja possível garantir a avaliação ótima da rede.	Faz execução massivamente paralela da rede ao custo de talvez realizar trabalho desnecessário.
Momento em que a avaliação ocorre	A avaliação de um redex é adiada até que seu resultado seja necessário.	Um redex é avaliado assim que é encontrado, independentemente de seu resultado ser usado.
Uso de Memória	Pode ser mais eficiente se partes do programa nunca forem avaliadas. No entanto, pode reter redes grandes por muito tempo sem avaliá-las, possivelmente levando a um consumo de memória maior.	Todos os redexes são avaliados em paralelo, dando preferência a redexes de aniquilação, o que pode diminuir o uso de memória.
Desempenho	Tende a ser mais rápida na execução em um único núcleo do processador, e pode ser mais rápida que a avaliação paralela da versão strict se muitas partes da rede não forem efetivamente utilizadas.	Pode realizar trabalho computacional desnecessário se o resultado de uma expressão nunca for utilizado, mas tenta compensar essa desvantagem com a velocidade da execução massivamente paralela.

Tabela 1 – Tabela Comparativa: HVM3 *Lazy* vs. HVM3 *Strict*

3.2 HVM3 v1: Modelo de memória

A versão 1 da HVM3 será apenas um conjunto de definições de estruturas de dados e funções utilitárias para manipular memória e termos (células). Ela ainda não será capaz de computar nenhuma rede de interação.

3.2.1 Term

Vamos primeiro definir a estrutura mais básica da HVM3: o termo (Term). Ele é um valor de 64 bits que corresponde a um fio ou a uma célula qualquer nos combinadores de interação. Os bits de um termo serão divididos de acordo com o seguinte layout:

VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVLLLLLLLLLLLLLLLLSTTTTTTT

S: sub (1 bit). Vale 1 se esta for uma substituição, e 0 caso contrário (será detalhado posteriormente).

T: tag (7 bits). Identifica o tipo do termo.

L: lab (16 bits). Um rótulo usado para ativar interações de comutação ou aniquilação com termos DUP e SUP.

V: val (40 bits). O valor pode ser o endereço de um outro termo, um número, ou uma entrada no mapa de substituição, e será definido com base na tag.

Caso o termo represente uma célula binária, o endereço para o qual "val" aponta será, na verdade, um *array* de dois elementos (termos). Posteriormente, introduziremos outros tipos de tags em que "val" pode apontar para um *array* de ainda mais elementos; por isso, não há uma limitação para a quantidade de termos que o endereço em "val" aponta.

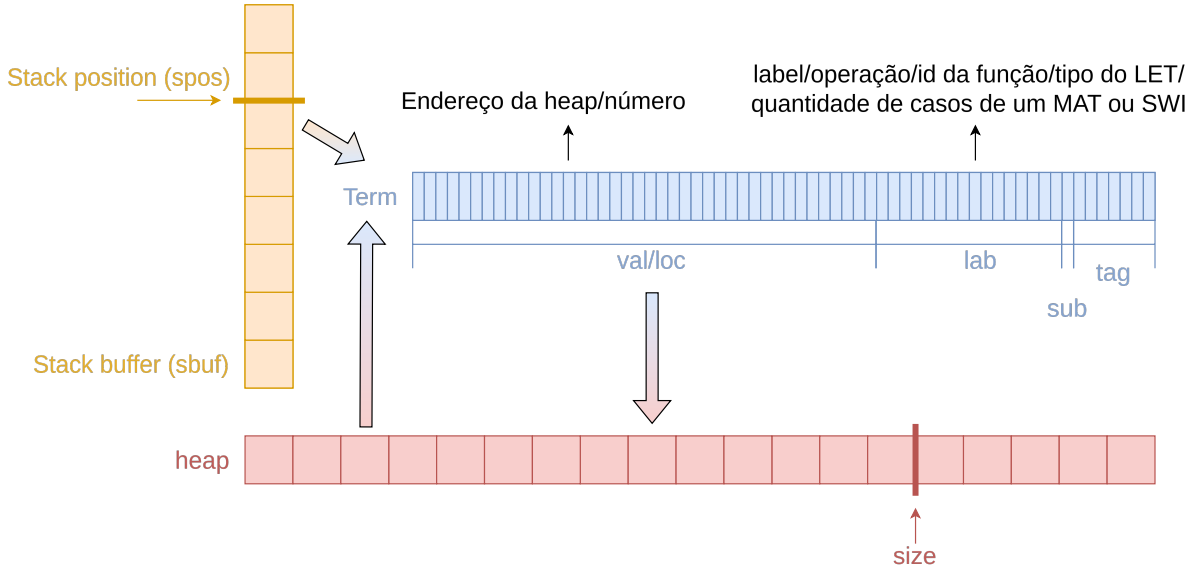

Figura 28 – Modelo de memória da HVM3 *Lazy*

```
type Term = inteiro sem sinal de 64 bits
type Tag = inteiro sem sinal de 8 bits
type Lab = inteiro sem sinal de 16 bits
```

Como estamos trabalhando com pseudocódigo, poderíamos definir Tag como um inteiro sem sinal de 7 bits, mas, realisticamente, um computador só dará suporte a tipos numéricos com uma quantidade de bits que seja uma potência de 2, ou seja, desperdiçaremos alguns bits na prática. O mesmo vale para as outras definições futuras.

Outras funções utilitárias para termos seguem abaixo. Elas são triviais de serem implementadas com manipulação de bits. Todas são funções puras.

```
// Cria um Term novo
fn term_new(tag: Tag, lab: Lab, loc: Loc) -> Term
```

```
3
4  // Extrai a tag de um Term
5  fn term_tag(term: Term) -> Tag
6
7  // Extrai a lab de um Term
8  fn term_lab(term: Term) -> Lab
9
10 // Obtém o valor do bit s de um Term
11 fn term_get_bit(term: Term) -> Bit
12
13 // Define o bit s de um Term como 1
14 fn term_set_bit(term: Term) -> Term
15
16 // Define o bit s de um Term como 0
17 fn term_rem_bit(term: Term) -> Term
```

3.2.2 Loc

Loc representa um endereço de memória. O *layout* do termo nos limita a um val de 40 bits, portanto, poderemos endereçar até 2^{40} regiões de memória. Posteriormente, Loc também poderá representar números e caracteres além de endereços.

```
1  type Loc = inteiro sem sinal de 64 bits
2
3  // Extrai o loc de um Term
4  fn term_loc(x: Term) -> Loc
5
6  // Define a loc de um Term com o valor informado
7  fn term_set_loc(term: Term, loc: Loc) -> Term
8
9  // Extrai um termo da memória da HVM.
10 fn take(loc: Loc) -> Loc {
11   val: Term = HVM.heap[loc]
12   HVM.heap[loc] = 0 // Valor arbitrário
13   return val
14 }
15
16 // Armazena um termo na memória da HVM e o marca como uma
17 // substituição
18 fn sub(loc: Loc, term: Term) {
19   HVM.heap[loc] = term_set_bit(term)
20 }
```

3.2.3 Memória global

Em tempo de execução, precisamos, no mínimo, de um *buffer* para armazenar os termos e outro para os redexes. Esses elementos são definidos na estrutura a seguir:

```
1  type State = struct {
2    sbuf: Term[] // buffer da pilha de redução
3    spos: u64 // posição da pilha de redução
4    heap: Term[] // buffer global de Termos
```

```
  size: u64 // posição do buffer global de Termos
}

HVM: State // Memória global da HVM
```

O heap será implementado através de um *bump allocator*. Apesar de ser problemático quanto à reutilização de memória, trata-se de uma solução bastante rápida, e no momento, é o alocador de escolha da HVM3.

A implementação de um bump allocator é relativamente simples: aloque um pedaço grande de memória e crie um ponteiro para o início da memória alocada (nesse caso, o ponteiro é a variável "size"); avance o ponteiro sempre que o programa solicitar uma parte dessa memória. Assim, endereços de memória menores que o ponteiro são considerados ocupados, e endereços maiores ou iguais que o ponteiro são considerados livres. Não há maneira fácil de liberar memória com esse algoritmo.

```
HVM.heap = alloc(MAX_HEAP_SIZE * sizeof(Term))
HVM.size = 0

// Aloca espaço para armazenar 'aridade' quantidade de termos
fn alloc_node(aridade: Loc) -> Loc {
  if (HVM.size + aridade > MAX_HEAP_SIZE) {
    erro()
  }
  u64 p = HVM.size
  HVM.size += aridade
  return p
}
```

A pilha de redução é implementada de maneira semelhante, mas a estrutura de pilha permite que liberemos a memória facilmente apenas recuando o ponteiro.

```
HVM.sbuf = alloc(MAX_STACK_SIZE * sizeof(Term))

// Coloca um termo no topo da pilha de redexes
fn spush(term: Term)

// Remove e retorna o elemento do topo da pilha de redexes
fn spop() -> Term
```

3.3 HVM3 v2: Combinadores de interação

A HVM3 possui todas as células dos combinadores de interação, e mais algumas outras que, apesar de teoricamente dispensáveis, aumentam significativamente a eficiência computacional e facilitam a implementação de algumas funcionalidades. Inicialmente, começaremos apenas pelo essencial: variáveis, células Eraser (ε), Constructor (γ) e Duplicator (δ).

A sintaxe da linguagem interpretada pela HVM3 também é uma representação textual de redes de combinadores de interação, e em grande parte é igual ao cálculo de interação (IC) discutido superficialmente na seção 2.3.1. Primeiramente, toda a sintaxe que aprenderemos será muito semelhante ao IC, e, depois, expandiremos-a.

A partir deste ponto, células serão chamadas de termos (terms) e fios de variáveis (vars), assim como no IC.

VARIABLE (VAR) Equivalentes a fios nos combinadores de interação. As variáveis na HVM3 têm escopo global e uso afim (podem ocorrer no máximo uma vez no corpo de uma função, a menos que sejam explicitamente duplicadas), ou seja, aparecem uma vez como argumento e uma vez efetivamente no corpo da função.

- **Sintaxe:** x // qualquer caractere minúsculo

Figura 29 – Termo VAR

ERASER (ERA) Representa o apagamento de um termo. Equivalente a uma célula apagadora (ε) nos combinadores de interação.

- **Sintaxe:** *
- **Exemplo:** (* K) // K é qualquer termo; parênteses indicam uma aplicação, que será explicada em breve
- **Layout**
  - tag: ERA
  - lab: Não utilizado
  - val: Não utilizado

Figura 30 – Termo ERA

LAMBDA (LAM) Representa uma função lambda. É associada a uma célula construtora (γ) nos combinadores de interação.

- **Sintaxe**: λ<argumento> <corpo> // <argumento> é uma VAR

- **Exemplo:** λx (* x)
- **Layout**
  - tag: LAM
  - lab: Não utilizado
  - val: Endereço do corpo do LAM

  Observe que apenas o corpo do LAM é armazenado, portanto não sabemos exatamente qual é o argumento e onde ele é utilizado. Veremos, mais adiante, que podemos guardar o argumento e corpo da função no mesmo espaço de memória.

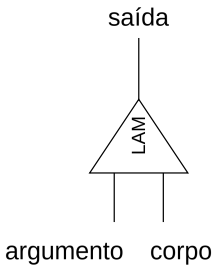

Figura 31 – Termo LAM

APPLICATION (APP) Aplica uma função a um argumento. É associada a uma célula construtora (γ) nos combinadores de interação.

- **Sintaxe:** (<função> <argumento>)
- **Exemplo:** (λx x 10) (aplica a função identidade ao número 10)
- **Layout**
  - tag: APP
  - lab: Não utilizado
  - val: Endereço de uma região contígua de memória com dois termos: o primeiro é a função, e o segundo é o argumento.

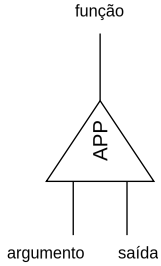

Figura 32 – Termo APP

DUPLICATOR (DUP) Permite copiar um termo, sendo essencial para lidar com as variáveis não-afins do IC. É associada a uma célula duplicadora (δ) nos combinadores de interação.

- **Sintaxe:** !&<label>{<variavel0> <variavel1>} = <valor> <corpo>
  - <label>: Um valor numérico opcional que afeta a interação DUP-SUP. Pode ser:
    - FshSup: &{a b} // usa um label novo/fresh, gerado automaticamente.
    - StaSup: &0{a b} // usa um label estático (igual a 0, neste caso).
  - <variavel0>, <variavel1>: As novas variáveis, que são cópias do <valor>.
  - <valor>: O termo a ser duplicado.
  - <corpo>: O termo onde as novas variáveis são usadas.
- **Exemplo:** !&0{x y} = V B // duplica 'V' em x e y no corpo B.
- **Layout** Em tempo de execução, não existe um termo DUP que armazene dois valores. Na verdade, criamos dois termos separadamente, porém com mesmo "lab" e mesmo "val".
  - tag: DP0 ou DP1
  - lab: Qualquer valor. É o mesmo para DP0 e DP1
  - val: Endereço do termo que foi duplicado

  A separação do DUP em termos DP0 e DP1 é crucial para o direcionamento da execução na HVM3 *Lazy*. Conforme será detalhado na seção 3.10, a ordem em que os termos são avaliados depende da ordem em que termos são colocados em uma pilha de execução, e com essa separação podemos escolher qual dos termos colocar na pilha e, consequentemente, escolher qual caminho do grafo percorrer primeiro. Por exemplo, durante uma interação DUP-SUP, o termo colocado na stack é o que possui a porta de mesmo índice pela qual a execução se iniciou. Assim, se "entramos" no caminho pela porta 0 do DUP (DP0), colocamos o termo da porta 0 do SUP na pilha e definimos o bit de substituição do termo da porta 1.

SUPERPOSITION (SUP) A superposição é a inovação essencial que permite a avaliação ótima. Ela não é um simples par de termos, mas sim um mecanismo para o compartilhamento da execução parcial de funções. Quando uma função é duplicada, seu argumento se torna uma superposição das variáveis de cada cópia da função, fazendo com que seu corpo, com exceção da parte superposta (que efetivamente utiliza os argumentos), possa ser avaliado uma

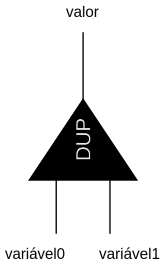

Figura 33 – Termo DUP

única vez para todas as suas cópias. É associada a uma célula duplicadora (δ) nos combinadores de interação.

- **Sintaxe:** &<label>{<termo0> <termo1>}
  - <label>: Similar a DUP.
- **Exemplo:** &0{A B} (uma superposição com *label* 0 contendo os termos A e B)
  - tag: SUP
  - lab: Qualquer valor
  - val: Endereço de uma região contígua de memória com dois termos

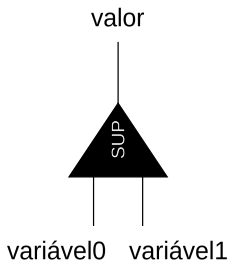

Figura 34 – Termo SUP

Veja um exemplo de representação da soma de 1 + 2 com base nos numerais de Church e sua representação gráfica na figura 36.

```
1  @um = λf λx (f x)
2
3  @dois = λf λx
4    !&{f0 f1} = f
5    (f1 (f0 x))
6
7  @soma = λm λn λf λx
8    !&{f0 f1} = f
9    ((m f1) ((n f0) x))
10
11 @umMaisDois = λf λx
12   !&{f0 f1} = f
```

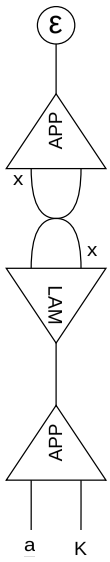

Figura 35 – Exemplo da representação de (λx (* x) a) em redes de interação

```
13  ((λf2 λx0 (f2 x0) f1)
14   ((λf3 λx1
15      !&{f4 f5} = f3
16      (f5 (f4 x1)) f0) x
17   )
18  )
```

3.3.1 Interações

A maneira com que esses termos interagem é, basicamente, a mesma com que os combinadores de interação interagem, mas vamos rever as interações levando em consideração, também, a representação textual do IC. Nas figuras, também estão evidenciadas o tipo de cada termo, o que é importante para a eficiência da HVM, mas não é importante na teoria dos combinadores de interação.

APP-ERA A função de um termo ERA é apagar outros termos. É natural que tentar aplicá-lo apagará outros termos. Neste caso, tanto o termo APP quanto a rede ligada a “a” serão apagadas, e o termo ERA será ligado a “K” para continuar apagando o resto da rede. Esta regra de interação é ligeiramente diferente da apresentada na Figura 12, e pode ser vista como uma otimização, pois em vez de propagar um termo ERA também para “a”, a HVM3 pode, simplesmente, ignorar a rede ligada àquela porta pelo resto da execução. Esse é um padrão que se repetirá para outras interações que normalmente exigem a propagação de termos ERA.

$$\frac{\text{(* a)}}{*}$$

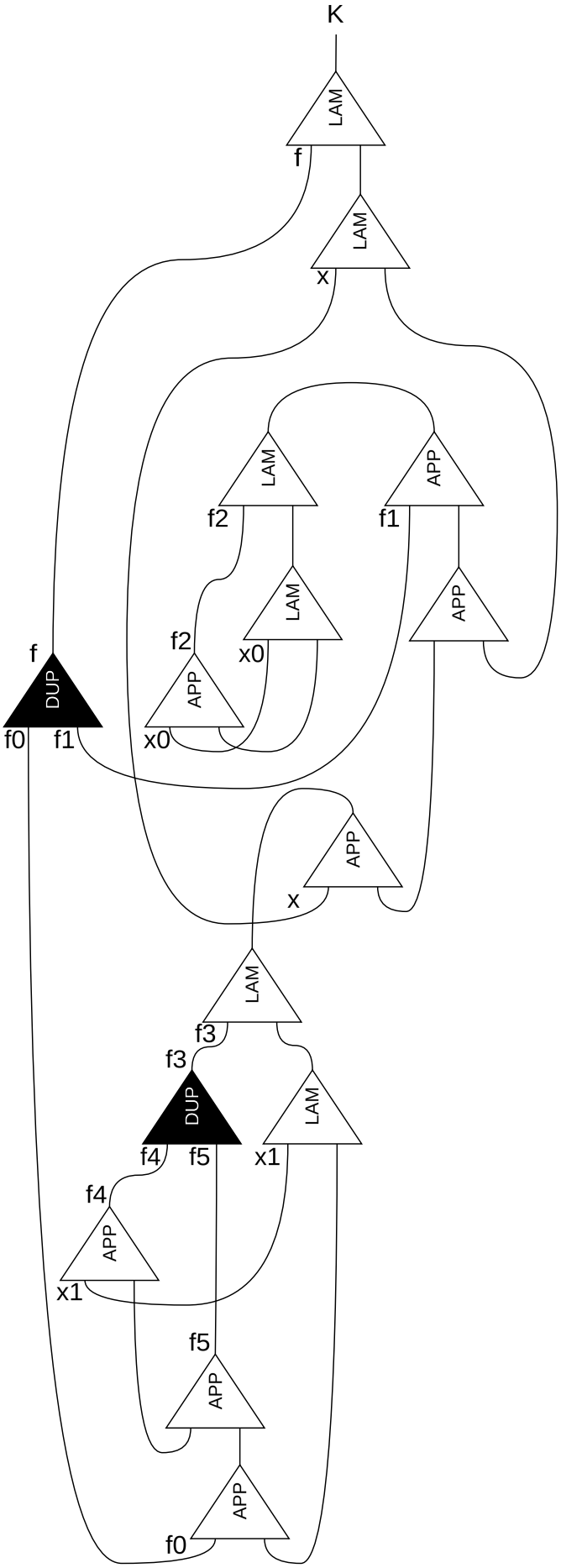

Figura 36 – Rede de interação da HVM3 que representa 1 + 2

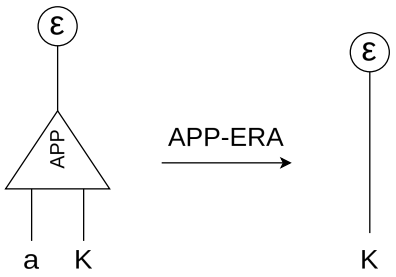

Figura 37 – Interação APP-ERA

```
1 fn reduce_app_era(app: Term, era: Term) -> Term {
2   return era
3 }
```

APP-LAM Aplicar um LAM é muito semelhante a fazer uma β-redução: basta substituir a variável e eliminar o termo LAM.

$$\frac{(\lambda x.f\ a)}{x \leftarrow a}$$
$$f$$

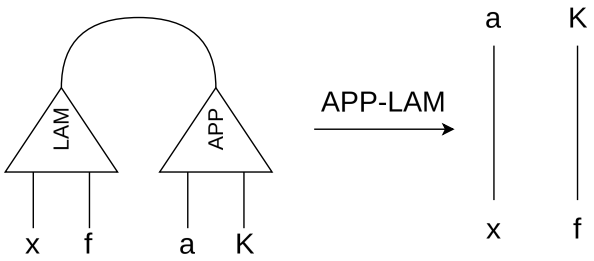

Figura 38 – Interação APP-LAM

Já que o termo LAM será eliminado ao mesmo tempo em que seu argumento é substituído, não precisamos gastar memória para armazenar o corpo e o argumento do LAM separadamente. Se em outros locais do código HVM ocorrer uma variável que não está ligada, ou seja, *flag* sub = 0, não há nada que possamos fazer, simplesmente deixaremos para resolver o redex *mais tarde* (veja 3.10). Já que o LAM não aplicado sempre tem sub = 0, podemos armazenar o corpo do LAM em um endereço, e após APP-LAM ser executado, sobrescrever esse mesmo endereço com o valor ligado do argumento com sub = 1. Em outras palavras, o mesmo endereço de memória armazena tanto o corpo, quanto o argumento, e sabemos o que está sendo armazenado com base no bit sub.

Por exemplo, "λx (y x)" será armazenado como "(0x02 0x01)". Temos que aplicar o valor do endereço 0x01 na função do endereço 0x02. Se sub de 0x01 é

1, fazemos a aplicação normalmente, caso contrário, deixamos para executar essa interação mais tarde, pois sabemos que 0x01 ainda não foi ligado (e no momento está armazenando o corpo de um LAM).

```
fn reduce_app_lam(app: Term, lam: Term) -> Term {
  app_loc: Loc = term_loc(app)
  lam_loc: Loc = term_loc(lam)
  corpo: Term = HVM.heap[lam_loc]
  arg: Term = HVM.heap[app_loc + 1]

  sub(lam_loc, arg)
  return corpo
}
```

DUP-ERA Substituímos DP0 e DP1 por termos ERA, essencialmente duplicando-o. K é qualquer outro termo (o resto do programa em que a substituição ocorrerá).

$$\frac{\text{! \&L\{r s\} = *} \quad K}{\begin{array}{c} r \leftarrow * \\ s \leftarrow * \\ K \end{array}}$$

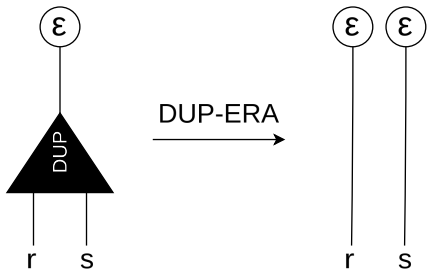

Figura 39 – Interação DUP-ERA

```
fn reduce_dup_era(dup: Term, era: Term) -> Term {
  dup_loc: Loc = term_loc(dup)
  sub(dup_loc, era)
  return era
}
```

APP-SUP Superposições existem para que duas aplicações ocorram com o mesmo valor. Para isso, é claro que precisamos duplicar tal valor e associar cada cópia à cada aplicação da superposição.

$$\frac{\text{(\&L\{a b\} c)} \quad \text{! \&L\{c0 c1\} = c}}{\text{\&L\{(a c0) (b c1)\}}}$$

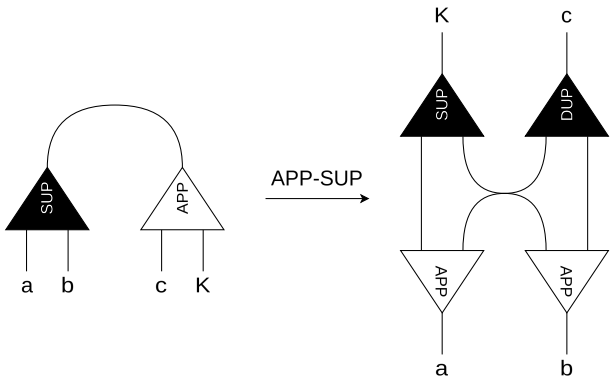

Figura 40 – Interação APP-SUP

```
fn reduce_app_sup(app: Term, sup: Term) -> Term {
  app_loc: Loc = term_loc(app)
  sup_loc: Loc = term_loc(sup)
  sup_lab: Lab = term_lab(sup)

  arg: Term = HVM.heap[app_loc + 1]
  tm1: Term = HVM.heap[sup_loc + 1]

  loc: Loc = alloc_node(3)
  ap0: Loc = sup_loc
  ap1: Loc = loc + 0
  su0: Loc = app_loc
  dup: Loc = loc + 2

  set(ap0 + 1, term_new(DP0, sup_lab, dup))

  set(ap1 + 0, tm1)
  set(ap1 + 1, term_new(DP1, sup_lab, dup))

  // Reciclar app_loc com o resultado da superposição
  set(su0 + 0, term_new(APP, 0, ap0))
  set(su0 + 1, term_new(APP, 0, ap1))

  set(dup + 0, arg)

  return term_new(SUP, sup_lab, su0)
}
```

DUP-LAM Duplicar um LAM envolve duplicar seu corpo e argumento. Assim como vimos na seção 2.3.1, o resultado incluirá um DUP acompanhado de um SUP de mesmo label. Veja na Figura 41, que a superposição não aponta para a porta principal do LAM, portanto precisamos do auxílio de VARs para ligá-la. Isso não é necessário para DUP pois, em tempo de execução, dividimos-o em DP0 e DP1.

$$
\begin{array}{c}
\text{! \&L\{r s\} = λx f} \\
\text{K} \\
\hline
\text{r ←λx0 f0} \\
\text{s ←λx1 f1} \\
\text{x ←\&L\{x0 x1\}} \\
\text{! \&L\{f0 f1\} = f} \\
\text{K}
\end{array}
$$

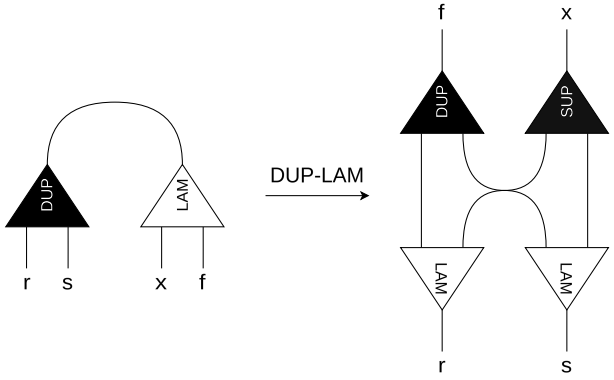

Figura 41 – Interação DUP-LAM

```
fn reduce_dup_lam(dup: Term, lam: Term) -> Term {
  dup_loc: Loc = term_loc(dup)
  lam_loc: Loc = term_loc(lam)
  dup_lab: Lab = term_lab(dup)

  bod: Term = got(lam_loc)

  loc: Loc = alloc_node(5)
  lam0: Loc = loc + 0
  lam1: Loc = loc + 1
  sup: Loc = loc + 2
  dup: Loc = loc + 4

  sub(lam_loc, term_new(SUP, dup_lab, sup))

  set(lam0, term_new(DP0, dup_lab, dup))
  set(lam1, term_new(DP1, dup_lab, dup))
  set(su0, term_new(VAR, 0, lam0))
  set(su0 + 1, term_new(VAR, 0, lam1))
  set(dup, bod)

  if (term_tag(dup) == DP0) {
    sub(dup_loc, term_new(LAM, 0, lam1))
    return term_new(LAM, 0, lam0)
  } else {
    sub(dup_loc, term_new(LAM, 0, lam0))
    return term_new(LAM, 0, lam1)
  }
}
```

DUP-SUP Há duas possibilidades de interações para DUP-SUP, dependendo se suas labels são iguais ou não.

$$\frac{\begin{array}{c}\text{! \&L\{x y\} = \&L\{a b\}}\\ \text{K}\end{array}}{\begin{array}{c}\text{x} \leftarrow \text{a}\\ \text{y} \leftarrow \text{b}\\ \text{K}\end{array}} \quad \text{(Para labels iguais)}$$

$$\frac{\begin{array}{c}\text{! \&L\{x y\} = \&R\{a b\}}\\ \text{K}\end{array}}{\begin{array}{c}\text{x} \leftarrow \text{\&R\{a0 b0\}}\\ \text{y} \leftarrow \text{\&R\{a1 b1\}}\\ \text{! \&L\{a0 a1\} = a}\\ \text{! \&L\{b0 b1\} = b}\\ \text{K}\end{array}} \quad \text{(Para labels diferentes)}$$

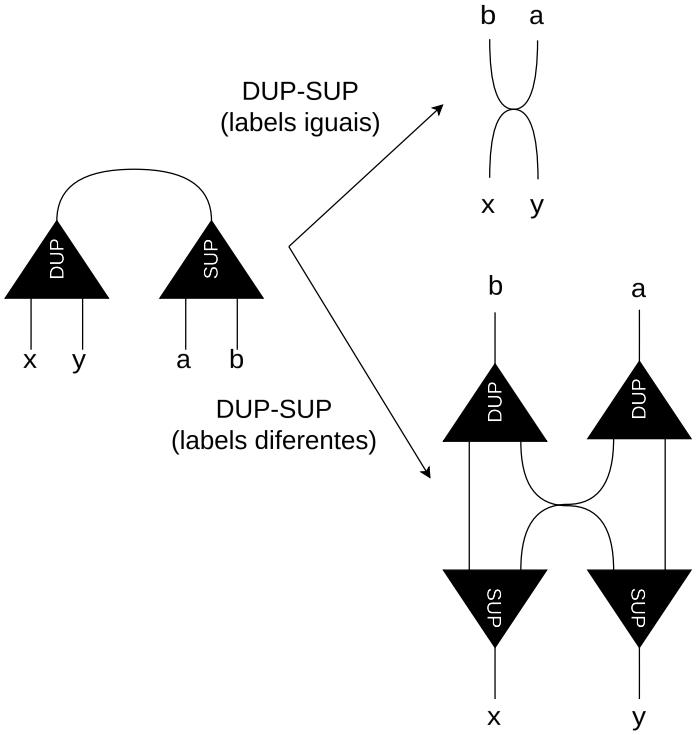

Figura 42 – Interação DUP-SUP

```
fn reduce_dup_sup(dup: Term, sup: Term) -> Term {
  dup_loc: Loc = term_loc(dup)
  dup_lab: Lab = term_lab(dup)
  sup_lab: Lab = term_lab(sup)
  sup_loc: Loc = term_loc(sup)
  if (dup_lab == sup_lab) {
    tm0: Term = HVM.heap[sup_loc + 0]
    tm1: Term = HVM.heap[sup_loc + 1]
    if (term_tag(dup) == DP0) {
      sub(dup_loc + 0, tm1)
      return tm0
    } else {
      sub(dup_loc + 0, tm0)
      return tm1
```

```
15      }
16    } else {
17      loc: Loc = alloc_node(4)
18      du0: Loc = sup_loc + 0
19      du1: Loc = sup_loc + 1
20      su0: Loc = loc + 0
21      su1: Loc = loc + 2
22
23      set(su0 + 0, term_new(DP0, dup_lab, du0))
24      set(su0 + 1, term_new(DP0, dup_lab, du1))
25      set(su1 + 0, term_new(DP1, dup_lab, du0))
26      set(su1 + 1, term_new(DP1, dup_lab, du1))
27      if (term_tag(dup) == DP0) {
28        sub(dup_loc + 0, term_new(SUP, sup_lab, su1))
29        return term_new(SUP, sup_lab, su0)
30      } else {
31        sub(dup_loc + 0, term_new(SUP, sup_lab, su0))
32        return term_new(SUP, sup_lab, su1)
33      }
34    }
35  }
```

3.3.2 Variáveis globais

Ao contrário do cálculo de interação, na semântica da HVM3, todas as variáveis têm escopo local por padrão, portanto, o código a seguir é inválido.

```
1 λt ((t x) λx λy y)
```

Todavia, podemos fazer com que uma variável tenha escopo global, bastando que seu nome inicie com “$”. Assim, o código abaixo é perfeitamente válido.

```
1 λt ((t $x) λ$x λy y)
```

3.4 HVM3 v3: Operações Aritméticas

A versão 3 da HVM3 expande as capacidades da máquina para incluir suporte a valores numéricos e operações aritméticas básicas. Esta seção detalha a representação de números inteiros e caracteres, bem como as interações que permitem a realização de cálculos.

3.4.1 Termos Numéricos: U32 e CHR

A HVM3 introduz termos específicos para representar valores numéricos de 32 bits (U32) e caracteres (CHR), que internamente são tratados como seus equivalentes inteiros. Não há suporte para números de ponto flutuante, e ambos esses termos são representados pelo mesmo termo W32, mostrado na Figura 43.

Inteiro de 32 bits sem sinal (U32) Representa um número inteiro sem sinal de 32 bits.

- **Sintaxe:** 123 // qualquer sequência de dígitos
- **Layout**
  - tag: U32
  - lab: Não utilizado
  - val: O valor inteiro de 32 bits

CHARACTER (CHR) Representa um único caractere, que é internamente convertido para seu valor inteiro Unicode.

- **Sintaxe:** 'a' // um caractere entre aspas simples
- **Layout**
  - tag: CHR
  - lab: Não utilizado
  - val: O valor inteiro do caractere

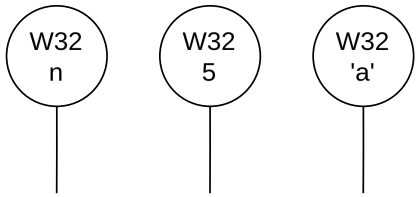

Figura 43 – Termo W32. "n" pode assumir o valor de qualquer U32 ou CHR

3.4.2 Operações Aritméticas: OP2, OPX e OPY

As operações aritméticas na HVM3 não são feitas em um único passo. Na verdade, a avaliação segue o seguinte processo:

1. Um operador (ex: +) primeiro interage com seu primeiro operando (ex: a).
2. Ele se transforma em um estado intermediário, em que é garantido que o primeiro valor já foi avaliado.
3. Então, nesse novo estado, ele interage com o segundo operando (ex: b) para produzir o resultado final (ex: a + b).

Para compreender a representação desse conceito na HVM3, Nicolas Abril (2025a), desenvolvedor na HOC, esclareceu que três termos são usados, mas apenas os dois últimos existem efetivamente na rede em tempo de execução:

Operador binário (OP2) Representa uma operação binária.

- **Sintaxe:** (o a b) // “o” é o operador e “a” e “b” são os operandos
- **Exemplo:** (+ 2 3)
- **Layout:** Não é representado diretamente na memória

Operador X (OPX) Representa a primeira parte de uma operação binária, composta pelo operador e os operandos ainda não avaliados.

- **Sintaxe:** <op(a b) // Não faz parte da sintaxe de arquivos .hvm
- **Exemplo:** <+(2 3)
- **Layout:**
  - tag: OPX
  - lab: O código da operação.
  - val: Endereço de uma região de memória contígua com espaço para dois termos, que são os respectivos operandos. Esse espaço alocado será reaproveitado ao criar um termo OPY, posteriormente.

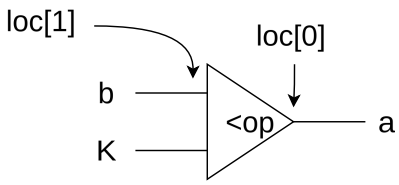

Figura 44 – Termo OPX

Operador Y (OPY) Representa a segunda parte de uma operação binária, é criado após um termo OPX interagir com seu primeiro operando.

- **Sintaxe:** >op(a b) // Não faz parte da sintaxe de arquivos .hvm
- **Exemplo:** >+(2 3)
- **Layout:**
  - tag: OPY
  - lab: O mesmo código de operação do OPX correspondente
  - val: Endereço de uma região de memória contígua com os dois operandos: a primeira posição guarda o segundo operando, e a segunda, o primeiro operando, ou seja, é a ordem contrária de um OPX. Isso ocorre, pois uma interação ocorre sempre com loc[0], e o OPY ainda precisa interagir com seu segundo operando para que a operação aconteça.

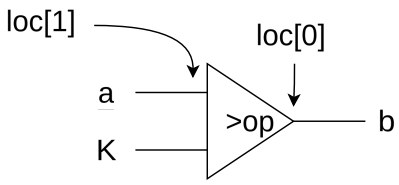

Figura 45 – Termo OPY

As operações suportadas são: ADD, SUB, MUL, DIV, MOD, EQ, NE, LT, GT, LTE, GTE, AND, OR, XOR, LSH, RSH.

3.4.3 Interações com Termos Numéricos

Um termo não avaliado é aquele em que ainda há redexes a serem feitos. Pode ser que um termo esteja ligado com o resto da rede através da sua porta principal, mas somente após o resto da rede se reduzir a, por exemplo, um U32, é que saberemos qual regra de interação aplicar.

Um OPX guarda ambos os operandos como Termos ainda não avaliados, enquanto um OPY guarda ambos os operandos, mas, nesse caso, o primeiro já foi avaliado e o segundo operando, não. Mais tarde, o OPY interagirá com seu próprio segundo operando, mas, dessa vez, ele já terá sido avaliado, e então a operação aritmética finalmente poderá acontecer. Veja as interações com U32 abaixo e a figura 48. No pseudocódigo, as variáveis que se referem a termos U32 foram nomeadas como w32 para evitar confusão com o tipo de dado u32.

- **OPX-{U32,CHR}:** Quando um OPX encontra um termo U32, ele se reorganiza para se tornar um OPY, trocando a posição do 2º operando e armazenando o termo interagido como o 1º operando.

```
fn reduce_opx_w32(opx: Term, nmx: Term) -> Term {
  opx_lab: Lab = term_lab(opx) // será U32 ou CHR
  opx_loc: Loc = term_loc(opx)
  // Obtém o segundo operando
  nmy: Term = HVM.heap[opx_loc + 1]
  set(opx_loc + 0, nmy) // Move o segundo operando
  set(opx_loc + 1, nmx) // Move o primeiro operando
  return term_new(OPY, opx_lab, opx_loc)
}
```

- **OPY-{U32,CHR}:** Quando um OPY encontra um termo U32 (que sempre será seu segundo operando) a operação aritmética é finalmente executada.

```
fn reduce_opy_w32(opy: Term, w32: Term) -> Term {
  opy_loc: Loc = term_loc(opy)
  t: Tag = term_tag(w32) // será U32 ou CHR
  x: u32 = term_loc(HVM.heap[opy_loc + 1])
  y: u32 = term_loc(w32)
```

```
  result: u32

  switch (term_lab(opy)) {
    case OP_ADD: result = x + y
    case OP_SUB: result = x - y
    // Outras operações são análogas e triviais...
    default: erro()
  }
  return term_new(t, 0, result)
}
```

- **OPX-SUP:** Interação de comutação.

$$\frac{\text{<op(\&L\{x0 x1\} y)}}{\text{! \&L\{y0 y1\} = y}}$$
$$\text{\&L\{<op(x0 y0) <op(x1 y1)\}}$$

```
fn reduce_opx_sup(opx: Term, sup: Term) -> Term
```

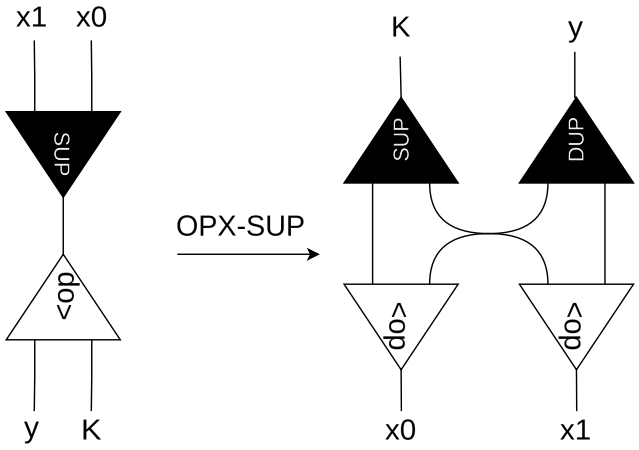

Figura 46 – Interação OPX-SUP

- **OPY-SUP:** Interação de comutação.

$$\frac{\text{>op(y \&L\{x0 x1\})}}{\text{! \&L\{y0 y1\} = y}}$$
$$\text{\&L\{>op(y0 x0) >op(y1 x1)\}}$$

```
fn reduce_opy_sup(opy: Term, sup: Term) -> Term
```

- **{OPX,OPY}-ERA:** Quando um dos operandos é um ERA, o resultado da operação é um ERA, apagando o outro operando.
- **{OPX,OPY}-LAM:** Interação inválida.

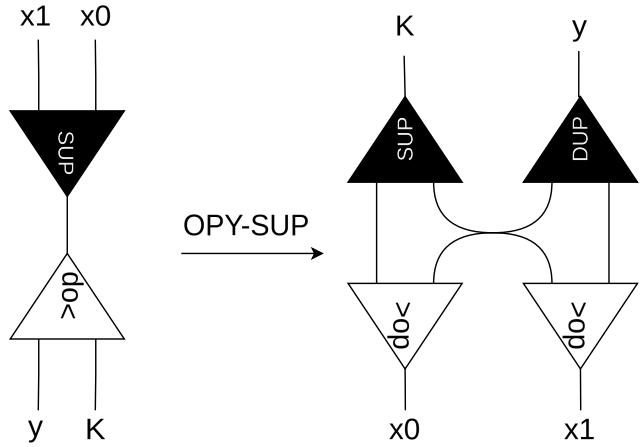

Figura 47 – Interação OPY-SUP

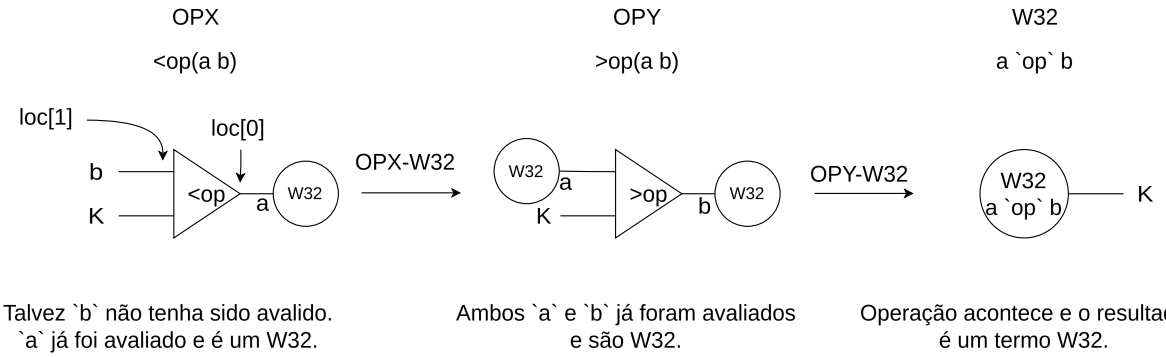

Figura 48 – Sequência de interações para que um OP2 seja completamente avaliado

3.5 HVM3 v4: Referências e funções

Duplicar funções-λ repetidamente para reaproveitá-las no código é, no mínimo, uma tarefa complicada. A adição de termos de referência nos permite representar funções nomeadas e utilizá-las quantas vezes quisermos no código sem termos duplicadores adicionais.

Referência (REF) Representa uma função nomeada. Ele não contém o corpo da função diretamente, mas sim uma referência a ele. Apenas a chamada não avaliada de uma função é representada por um REF, e não a declaração.

- **Sintaxe:** @<nome>(<arg0> <arg1> ...)
- **Exemplo:** @mul(x y) = (* x y) // declara a função (não é um termo REF)
- **Exemplo:** @mul(10 2) // chama a função ‘mul’
- **Layout:**

- tag: REF
- lab: O identificador único do corpo da função.
- val: Endereço de uma região de memória contígua com espaço para "n" termos, que são os argumentos da função. Este é o primeiro termo com aridade maior que binária citado até agora.

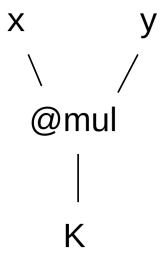

Figura 49 – Termo REF com 2 argumentos

Precisamos de um lugar para armazenar o corpo das funções. Vamos criar um *array* na memória global da HVM, tal que o índice do *array* será o id da função em um REF.

```
type State = struct {
  ... // Outros campos vistos anteriormente
  book: (Term[] -> Term)[]
}
```

No código fonte original da HVM3, o corpo das funções é compilado para código C, mas vamos simplificar isso e dizer que nosso book conterá funções na nossa linguagem fictícia, funções essas que recebem um *array* com argumentos do REF e, a partir deles, geram uma rede de interação completamente independente, retornando o termo que possui uma porta livre pronta para ser conectada à rede que está sendo efetivamente avaliada.

3.5.1 Interações com REF

REF-{Term} Também chamada de **CALL**, esta interação ocorre quando REF interage com qualquer termo. Consiste em expandir o corpo da função apontada no REF, ligando os argumentos corretamente. Isso é feito utilizando a função em *book*, que foi gerada por um compilador antecipadamente.

```
fn reduce_ref(ref: Term) -> Term {
  f: Term[] -> Term = HVM.book[term_lab(ref)]
  args: Term[] = HVM.heap[term_loc(ref)]

  return f(args)
}
```

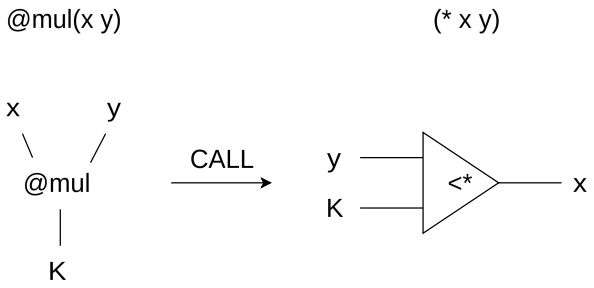

Figura 50 – Interação CALL

3.5.2 Otimizações

As funções e referências na HVM3 servem primariamente para otimização.

Duplicação rápida

A primeira otimização está na duplicação de termos. Copiar uma rede termo a termo, percorrendo o grafo e avaliando múltiplos duplicadores, é um processo lento e com um overhead de memória considerável. Na HVM3, uma definição de função é tratada como um termo imutável; isso permite que a HVM copie o subgrafo que representa a função de uma só vez, através de uma cópia de memória, o que é muito mais eficiente.

Pré-cálculo de execução parcial

A segunda forma de otimização é a capacidade de pré-calcular o resultado da execução parcial de funções para valores esperados. Como uma função tem uma estrutura fixa, o compilador pode analisar seu comportamento. Por exemplo, se uma função realiza um *match* em um valor conhecido em tempo de compilação, o compilador pode gerar um código especializado que já contém o resultado da aplicação parcial dessa função, assim, o termo MAT nem sequer existirá em tempo de execução.

Cópia de referências

A terceira otimização é uma regra de interação específica para quando uma referência a uma função interage com uma superposição (REF-SUP). Em vez de duplicar o corpo da função e depois lidar com a superposição, a HVM pode, simplesmente duplicar a própria referência. A regra é a seguinte:

$$\frac{\texttt{@foo(\&L\{ax ay\} b c ...)}}{\begin{array}{c}\texttt{! \&L\{bx by\} = b}\\ \texttt{! \&L\{cx cy\} = c}\\ \texttt{...}\end{array}} \quad \text{(quando “L” não faz parte de “@foo”)}$$

$$\texttt{\&L\{ @foo(ax bx cx ...) @foo(ay by cy ...)\}}$$

Essa interação surge apenas no processo de compilação de uma função, não fazendo parte do loop de redução em tempo de execução, que será evidenciado na seção 3.10.

Essa abordagem nem sempre resulta em um ganho de performance, em termos de número de interações, No entanto, o ganho obtido por não ter que duplicar o corpo da função passo a passo é, geralmente, maior.

3.6 HVM3 v5: *Match* Numérico (Switch)

A HVM3 otimiza o casamento de padrões para números inteiros usando um termo especializado chamado SWI. Ele funciona de forma semelhante a uma estrutura *switch-case* em linguagens como C. Esta é uma otimização importante, pois permite a implementação eficiente de funções recursivas sobre números ao compilar termos REF. O comportamento e sintaxe do SWI é muito parecido com os do termo MAT, que será evidenciado na próxima seção e, por isso, as interações serão reaproveitas para o MAT e o pseudo código nomeará as funções com o prefixo “mat”.

3.6.1 Sintaxe e Estrutura

- **Sintaxe:**

```
~<valor> {
  0: <corpo0>
  <caso1>: <corpo1>
  ...
  <casoGenérico>: <corpoGenérico>
}
```

- **Tipos de Caso:**
  - **Literal:** <n>: <corpo> // Executa o corpo se o valor for exatamente ‘n‘
  - **Predecessor:** <n>+<p>: <corpo> // Executa o corpo se o valor não for igual a nenhum outro caso. A variável ‘p‘ recebe ‘valor - m‘, em que ‘m‘ é a quantidade de casos anteriores a este. O valor de ‘n‘ é completamente ignorado, mas normalmente é igual a ‘m‘ para melhorar a legibilidade do código.
  - **Padrão:** _: <corpo> // Executa o corpo para qualquer valor que não tenha casado com os casos anteriores.
- **Regras de Estrutura:** A sintaxe correspondente a um SWI tem regras bastante rígidas, conforme outro desenvolvedor da HOC, Lorenzo W. Battistela (2025), explica:

- É obrigatório que o primeiro caso seja 0.
- É obrigatório que o último caso seja de Predecessor ou Padrão.
- Pode haver apenas um caso Predecessor ou Padrão.
- Os casos devem estar em ordem crescente e não pode haver “buracos” na sequência de números literais. O exemplo abaixo é inválido pois o caso dois foi pulado.

```
@f(n) = ~n {
  0: 100
  1: 200
  3: 400
  3+p: p
}
```

- **Exemplo**

```
@minha_funcao(n) = ~n {
  0: 100
  1: 200
  2+p: p
}

@minha_funcao(0) // retorna 100
@minha_funcao(1) // retorna 200
@minha_funcao(5) // casa com 2+p. p = 5-2 = 3. Retorna 3.
```

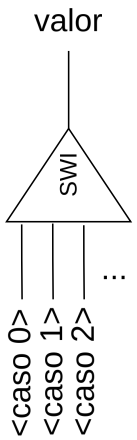

Figura 51 – Termo SWI

3.6.2 *Layout* de Memória e Implementação

Um termo SWI possui o seguinte *layout* de memória:

- **tag:** SWI
- **lab:** A quantidade de casos.

- **val:** Endereço para um bloco de memória contendo:
  - Posição 0: O termo a ser avaliado (<valor>).
  - Posições 1 a ‘lab‘: Ponteiros para os corpos dos casos literais (0, 1, ...).
  - Posição ‘lab‘+1: Ponteiro para o corpo do caso predecessor/padrão.

3.6.3 Interações com SWI

SWI-W32

$$\frac{\sim\text{num } \{\ 0\text{: <caso 0> } 1\text{: <caso 1> ... } n{+}p\text{: <caso n>}\}}{\text{<caso num>}} \quad \text{(Se num < n)}$$

$$\frac{\sim\text{num } \{\ 0\text{: <caso 0> } 1\text{: <caso 1> ... } n{+}p\text{: <caso n>}\}}{\begin{array}{c}p \leftarrow \text{num - n} \\ \text{<caso n>}\end{array}} \quad \text{(Se num >= n)}$$

$$\frac{\sim\text{num } \{\ 0\text{: <caso 0> } 1\text{: <caso 1> ... } \_\text{: <caso n>}\}}{\text{<caso num>}} \quad \text{(Se num < n)}$$

$$\frac{\sim\text{num } \{\ 0\text{: <caso 0> } 1\text{: <caso 1> ... } \_\text{: <caso n>}\}}{\text{<caso n>}} \quad \text{(Se num >= n)}$$

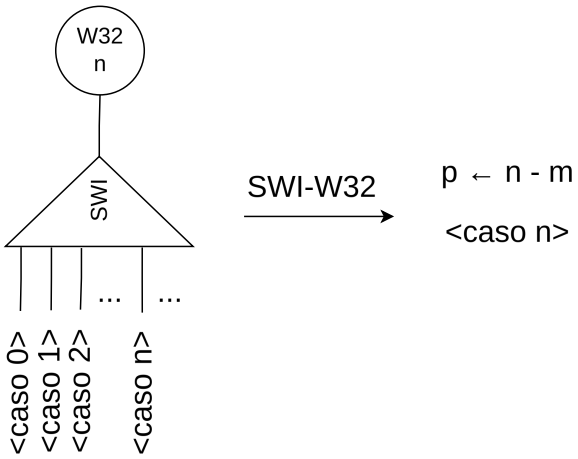

Figura 52 – Interação SWI-W32

```
fn reduce_mat_w32(mat: Term, w32: Term) -> Term {
  mat_loc: Loc = term_loc(mat)
  mat_len: u64 = term_lab(mat) // Quantidade de casos
  w32_val: u64 = term_loc(w32) // O valor numérico

  // Se o valor for menor que o número de casos literais,
  // o resultado é o corpo do caso correspondente.
  if (w32_val < mat_len - 1) {
    return HVM.heap[mat_loc + 1 + w32_val]
```

```
  } else {
    // Caso contrário, trata-se do caso predecessor/padrão.
    // Obtém o corpo do caso padrão
    f: Term = HVM.heap[mat_loc + mat_len]

    // Cria um novo termo numérico com o valor subtraído
    novo_w32: Term = term_new(W32, 0,
                        w32_val - (mat_len - 1))

    // Cria um termo de aplicação para (f novo_w32)
    app: Loc = mat_loc // Reutiliza o espaço do termo 'mat'
    set(app + 0, f)
    set(app + 1, novo_w32)

    return term_new(APP, 0, app)
  }
}
```

SWI-ERA

$$\frac{\sim * \; \{\ldots\}}{*}$$

```
fn reduce_mat_era(opx: Term, sup: Term) -> Term
```

SWI-SUP Se o <valor> de um SWI for um SUP, será gerado um SWI novo para cada termo do SUP, e os casos serão duplicados.

$$\frac{\sim \&L\{x\ y\}\ \{\ldots\}}{\&L\{\ \sim x\ \{\ldots\}\ \ \sim y\ \{\ldots\}\ \}}$$

```
fn reduce_mat_sup(opx: Term, sup: Term) -> Term {
  mat_tag: Tag = term_tag(mat)
  mat_lab: Lab = term_lab(mat)
  mat_loc: Loc = term_loc(mat)
  sup_loc: Loc = term_loc(sup)
  sup_lab: Lab = term_lab(sup)

  tm0: Term = got(sup_loc + 0)
  tm1: Term = got(sup_loc + 1)
  mat_len: u64 = mat_lab

  loc: Loc = alloc_node(1 + mat_len + mat_len)
  mat0: Loc = mat_loc
  mat1: Loc = loc + 0
  sup0: Loc = sup_loc

  set(mat0 + 0, tm0)
  set(mat1 + 0, tm1)
  // Duplicar cada caso do SWI
  for (i: u64 = 0; i < mat_len; i++) {
    du0: Loc = loc + 1 + mat_len + i
    set(du0 + 0, got(mat_loc + 1 + i))
    set(mat0 + 1 + i, term_new(DP0, sup_lab, du0))
    set(mat1 + 1 + i, term_new(DP1, sup_lab, du0))
  }
```

```
  set(sup0 + 0, term_new(mat_tag, mat_lab, mat0))
  set(sup0 + 1, term_new(mat_tag, mat_lab, mat1))
  return term_new(SUP, sup_lab, sup0)
}
```

SWI-LAM Interação inválida.

Com o *match* numérico, já se torna mais simples implementar alguns algoritmos, como a função fatorial.

```
@fat(x) = ~x {
  0: 1
  1+p: !&{p0 p1} = p
    (* (+1 p0) @fat(p1))
}
@fat(5) // 120
```

REFs não são necessários para recursão; vamos, pois, reescrever o código sem utilizá-los.

```
!&{$fat0 fat1} = λx
  ~x {
    0: 1
    1+p: !&{p0 p1} = p
      (* (+1 p0) ($fat0 p1))
  }
  (fat1 5) // 120
```

3.7 HVM3 v6: Tipos de Dados Algébricos (ADTs) com CTR

A HVM3 oferece suporte a Tipos de Dados Algébricos (ADTs) que permitem a criação de estruturas de dados complexas, como listas e árvores. A manipulação desses tipos é realizada, principalmente, através de dois termos: o Construtor (CTR) e o Match (MAT).

3.7.1 Construtores (CTR)

Um termo Construtor (CTR) representa uma das variantes de um ADT. Cada construtor pode conter zero ou mais campos, que são outros termos.

Sintaxe de Definição (data) Os ADTs são definidos usando a palavra-chave *data*, seguida pelo nome do tipo e seus construtores entre chaves. Os nomes dos construtores devem começar com "#".

```
data Bool { #True #False }
data List { #Nil #Cons{head tail} }
data Tree { #Leaf #Node{left right} }
```

Sintaxe de Uso Para criar uma instância de um construtor, usa-se um termo para cada campo, na mesma ordem em que foram declarados.

```
#Cons{1 #Nil} // Uma lista com o elemento 1
#Node{#Leaf #Leaf} // Uma árvore com duas folhas
```

Layout

- **tag:** CTR
- **lab:** O identificador único do construtor.
- **val:** O endereço de um bloco de memória contíguo que armazena os termos dos campos do construtor, na ordem em que foram declarados.

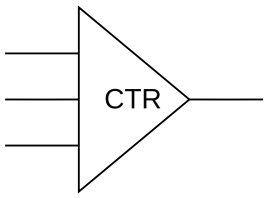

Figura 53 – Termo CTR com 3 campos

3.7.2 If-Let (IFL)

Ao compilar a sintaxe de um MAT, caso haja um caso padrão, o compilador da HVM3 não gera um termo MAT completo. Em vez disso, ele o traduz para uma cadeia de termos If-Let (IFL) aninhados. Essa otimização simplifica a árvore de redução.

Cada termo IFL testa um único construtor. Se o valor corresponder, ele executa o corpo do caso “then”. Caso contrário, ele passa o valor para o corpo do caso “else”, que por sua vez pode ser outro IFL, continuando a cadeia de testes até que um construtor corresponda ou o caso padrão final seja alcançado. As interações do IFL são as mesmas do MAT, exceto por IFL-CTR, a qual podemos criar um caso especial na mesma função de redução do MAT-CTR.

Layout:

- **tag:** IFL
- **lab:** O identificador do construtor a ser verificado
- **val:** Endereço para um bloco de memória contendo:
  - Posição 0: O termo a ser avaliado (<valor>).

- Posição 1: Ponteiro para o corpo do caso “then” (se o construtor corresponder).
- Posição 2: Ponteiro para o corpo do caso “else” (se não corresponder).

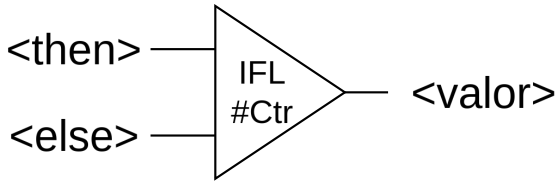

Figura 54 – Termo IFL

3.7.3 Casamento de Padrões (MAT)

O casamento de padrões em ADTs é feito com o termo MAT, que funciona de maneira análoga ao SWI. Ele avalia um termo e seleciona um corpo de código para executar, com base no construtor do valor.

3.7.3.1 Sintaxe e Estrutura

A sintaxe do MAT é muito semelhante à do SWI, utilizando o mesmo operador “∼” e a estrutura de chaves. A principal diferença está nos casos: em vez de literais numéricos, como “0:” ou “1+p:”, o MAT usa nomes de construtores.

- **Sintaxe:**

```
~<valor> {
  #<Construtor1>{<campo1>, ...}: <corpo1>
  #<Construtor2>{<campoA>, ...}: <corpo2>
  ...
  <caso_padrao>: <corpo_padrao>
}
```

- **Tipos de Caso:**
  - **Construtor:** #<Nome>{<var1>, <var2>, ...}: <corpo>: Se <valor> for uma instância do construtor #<Nome>, suas variáveis de campo (<var1>, <var2>, etc.) são ligadas e o corpo é executado.
  - **Padrão (Default):** <x>: <corpo> ou _: <corpo>: Se <valor> não corresponder a nenhum dos outros casos de construtor, ele é ligado à variável (x) e o corpo é executado.
- **Regras de Estrutura:**
  - É obrigatório que todos os construtores de um ADT estejam presentes para casamento no MAT, ou que o caso padrão exista.

  - Se todos os construtores do ADT estiverem presentes no MAT, é obrigatório que eles estejam ordenados de acordo com a ordem de declaração do ADT (feita com a palavra chave *data*).
  - Se o caso padrão existir, é obrigatório que ele seja o último listado.

- **Exemplo:**

```
1 @is_empty(list) = ~list {
2   #Nil: #True
3   #Cons{h t}: #False
4 }
5
6 @head(list) = ~list {
7   #Nil: 0 // Retorna um valor para indicar erro
8   #Cons{h t}: h
9 }
```

3.7.3.2 *Layout* de Memória e Implementação

O termo MAT é mais geral que o SWI e pode ser de dois tipos: MAT (para ADTs) ou IFL (If-Let, uma otimização para quando há um caso de construtor e um caso padrão).

***Layout* do termo MAT:**

- **tag:** MAT
- **lab:** O ID do ADT (ou seja, o ID do primeiro construtor do tipo de dado).
- **val:** Endereço para um bloco de memória contendo:
  - Posição 0: O termo a ser avaliado (<valor>).
  - Posições 1 em diante: Ponteiros para os corpos de cada caso, na ordem em que foram definidos.

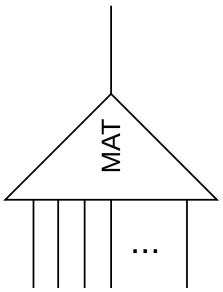

Figura 55 – Termo MAT com uma quantidade indeterminada de casos

Precisaremos alterar novamente a struct State para armazenar dados sobre ADTs e seus construtores.

```
type State = struct {
  ... // Outros campos vistos anteriormente
  cari: u16[] // Índice é o id de um construtor, conteúdo é a
              // sua aridade.
  clen: u16[] // Índice é o id de um construtor, conteúdo é a
              // quantidade de construtores que o ADT
              // correspondente tem.
  cadt: u16[] // Índice é o id de um construtor, conteúdo é o id
              // do ADT correspondente.
}
```

3.7.3.3 Interações com MAT

MAT-CTR Quando um termo MAT interage com um termo CTR, a HVM3 seleciona o corpo do caso correspondente.

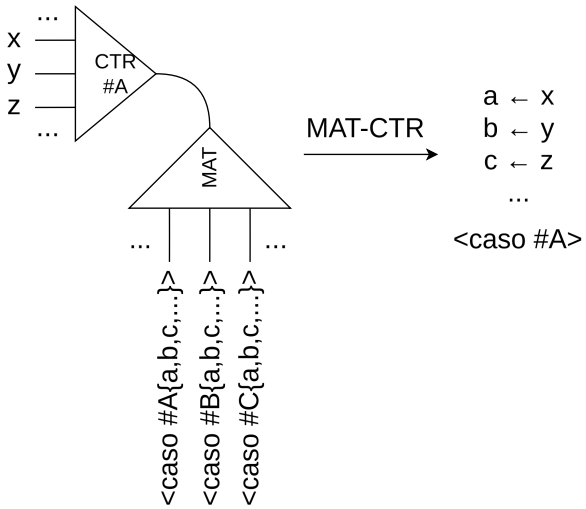

Figura 56 – Interação MAT-CTR

```
fn reduce_mat_ctr(mat: Term, ctr: Term) -> Term {
  mat_tag: Tag = term_tag(mat)
  mat_loc: Loc = term_loc(mat)
  mat_lab: Lab = term_lab(mat)
  ctr_loc: Loc = term_loc(ctr)
  ctr_lab: Lab = term_lab(ctr)
  mat_ctr: u64 = mat_lab
  ctr_num: u64 = ctr_lab
  // A aridade do construtor, ou seja, a quantidade de
  // campos
  ctr_ari: u64 = HVM.cari[ctr_num]
  loc: Loc = alloc_node(ctr_ari * 2)

  // Caso especial para If-Let (IFL)
  if (mat_tag == IFL) {
    // Se o ID do construtor for igual ao esperado pelo IFL
    if (mat_ctr == ctr_num) {
      // Pega o corpo do caso "then"
```

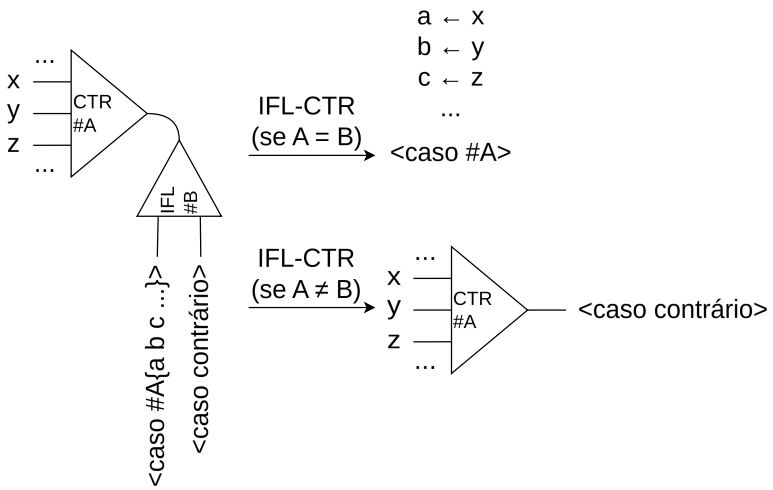

Figura 57 – Interação IFL-CTR

```
19      app: Term = HVM.heap[mat_loc + 1]
20
21      // Aplica sequencialmente cada campo do construtor ao
22      // corpo
23      for (i: u64 = 0; i < ctr_ari; i += 1) {
24        new_app: Loc = loc + i * 2
25        set(new_app + 0, app)
26        set(new_app + 1, HVM.heap[ctr_loc + i])
27        app = term_new(APP, 0, new_app)
28      }
29      return app
30    } else {
31      // Pega o corpo do caso "else"
32      app: Term = HVM.heap[mat_loc + 2]
33      // Aplica o construtor inteiro ao corpo do caso "else"
34      new_app: Loc = mat_loc // Reutiliza o espaço do 'mat'
35      set(new_app + 0, app)
36      set(new_app + 1, ctr)
37      app = term_new(APP, 0, new_app)
38      return app
39    }
40  }
41  // Caso geral para MAT de ADT
42  else {
43    cadt: u64 = HVM.cadt[mat_ctr]
44    clen: u64 = HVM.clen[mat_ctr]
45    if (ctr_num < cadt || ctr_num >= cadt + clen) {
46      // O construtor não é do mesmo tipo do ADT
47      erro();
48    }
49
50    // Calcula o índice do caso subtraindo o ID do ADT
51    // (mat_lab) do ID do construtor específico (ctr_lab)
52    case_idx: u64 = ctr_lab - mat_lab
53    // Pega o corpo do caso correspondente
54    app: Term = HVM.heap[mat_loc + 1 + case_idx]
55
56    // Aplica sequencialmente cada campo do construtor ao
57    // corpo
58    for (i: u64 = 0; i < ctr_ari; i += 1) {
```

```
59        new_app: Loc = loc + i * 2
60        set(new_app + 0, app)
61        set(new_app + 1, HVM.heap[ctr_loc + i])
62        app = term_new(APP, 0, new_app)
63      }
64      return app
65    }
66  }
```

Outras interações Em tempo de execução, termos MAT e SWI têm as mesmas regras de interação (exceto as que envolvem W32 e CTR), portanto, podemos reaproveitar as funções definidas anteriormente para SWI.

3.8 HVM3 v7: Let

Termos LET possibilitam a declaração de constantes.

Let (LET) Representa a declaração de uma variável.

- **Sintaxe (lazy):** ! <nome> = <term> <prox>
- **Sintaxe (strict):** !! <nome> = <term> <prox>
- **Exemplo (lazy):**

```
1 !z = (* x y)
2 z
```

- **Exemplo (strict):**

```
1 !!z = (* x y)
2 z
```

- **Layout:**
  - tag: LET
  - lab: LAZY ou STRI. Indica se o LET é *lazy* ou *strict*.
  - val: O endereço de um bloco de memória contíguo que armazena <term> na posição 0, e <prox> na posição 1.

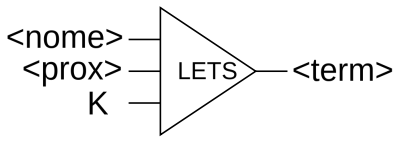

Figura 58 – Termo LET *strict* (LETS)

```
fn reduce_let(let: Term, val: Term) -> Term {
  let_loc: Loc = term_loc(let)
  bod: Term = got(let_loc + 1)
  sub(let_loc + 0, val)
  return bod
}
```

Ainda que seja possível, um LET *lazy* não precisa ser representado como uma célula na notação de grafos, interpretá-lo apenas como uma outra maneira de descrever a ligação de fios é suficiente.

O LET *strict* (LETS) força a avaliação imediata do <term>, economizando memória se este constituir uma rede extensa que permaneceria armazenada por muito tempo antes de ser reduzida. Isso pode ser feito através de uma única interação, que é ativada quando um LET *strict* interage com qualquer termo, como observado na Figura 59. Na prática, podemos implementar esse comportamento de maneira mais eficiente, apenas forçando que <term> seja colocado no topo da pilha de redução quando for encontrado (veja seção 3.10).

Figura 59 – Interação de um LETS com qualquer termo. <nome> e <term> formarão um redex logo após a interação.

A sintaxe LET também permite que termos referenciem a si mesmos, como na expressão abaixo:

```
λx.
  !$y = λ$z (x $y)
  $z
```

Porém, a HVM3, atualmente, não suporta variáveis cuja definição dependa de si mesma. Observe que “$y” aparece na sua própria definição e, portanto, o código anterior é ilegal na prática. Vale destacar que essa não é uma limitação absurda, e mesma com ela a linguagem de programação da HVM3 ainda é Turing completa, e qualquer função computável pode ser representada de uma maneira suportada pela máquina.

Com isso, outra questão surge: se um termo não pode referenciar a si mesmo, aparentemente todo LET *lazy* pode sofrer *inlining* sem alterar o comportamento do programa. Então qual é a utilidade de um LET *lazy*?

Em uma comunicação pessoal, Abril (2025b) explicou que, além ter potencial para melhorar a legibilidade do código, o LET *lazy* também pode ser uma oti-

mização para que um programa execute mais rapidamente. Apesar de uma variável global poder aparecer antes do termo que a liga, a HVM3 interpreta a rede primeiro na ordem em que os termos aparecem no código fonte, e se for tentado utilizar uma variável antes dela ser ligada, o interpretador precisará fazer *backtracking* do caminho percorrido na rede e tentar seguir por outro até que a variável seja ligada. Esse processo não tem um custo desprezível e, por isso é vantajoso que se tente ao máximo inserir no código o termo que faz a ligação da variável antes do que faz sua leitura. Com o auxílio do LET, isso fica bem fácil.

```
1 // Esta rede é avaliada mais eficientemente...
2 !a = λx λ$f x
3   &{$f a}
4
5 // ...que esta outra
6 &{$f λx λ$f x}
```

3.9 HVM3 v8: Priorizando redexes (INC/DEC)

Em uma comunicação pessoal, Abril e Battistela (2025) explicaram que, no modo de avaliação preguiçosa da HVM3, a avaliação dos termos ocorre sequencialmente e, por conta da confluência forte, não importa a ordem em que os termos são avaliados, o trabalho realizado sempre será o mesmo. Apesar disso, em algumas situações, pode ser interessante priorizar a avaliação de certos termos. Outro projeto da HOC é o SupGen (cujo nome final ainda está sendo discutido), que pode enumerar todos os valores possíveis de uma definição de tipo algébrico. Nesse contexto, é vantajoso priorizar a produção de alguns valores conforme uma heurística. Por exemplo, se precisamos encontrar uma única função de acordo com sua definição de tipo, vários candidatos serão gerados, e é interessante que valores com menor Complexidade de Kolmogorov sejam gerados primeiro, pois quando uma função que atenda aos requisitos for encontrada, podemos parar de produzir novos candidatos.

Para atender a essa necessidade, a HVM3 introduz dois termos unários: Incremento (INC) e Decremento (DEC), que permitem ao programador priorizar ou postergar a redução de determinados termos. Eles são significativos apenas no modo “collapse” de avaliação da HVM3, que permite que um programa produza mais de um resultado se a saída for um termo SUP, ou um aninhamento de SUPs, em que cada valor contido neles será uma saída diferente do programa; isto é, a estrutura de SUPs aninhados será planificada para uma stream de outputs.

INCREMENT (INC) Aumenta a prioridade de redução de um termo.

- **Sintaxe:** ↑<termo>

- **Exemplo:** ↑&0{1,2}
- **Layout**
  - tag: INC
  - lab: Não utilizado
  - val: Endereço do termo a ser priorizado

DECREMENT (DEC) Diminui a prioridade de redução de um termo.

- **Sintaxe:** ↓<termo>
- **Exemplo:** ↓&0{1,2}
- **Layout**
  - tag: DEC
  - lab: Não utilizado
  - val: Endereço do termo a ser postergado

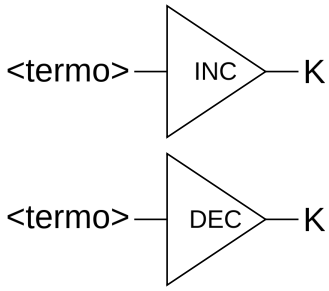

Figura 60 – Termos INC e DEC

Esses operadores são particularmente úteis em conjunto com superposições (SUP). Por padrão, a HVM3 pode resolver os termos dentro de uma superposição em qualquer ordem (mais especificamente, na ordem em que aparecem no código fonte no caso da HVM3 *lazy*). No entanto, ao usar um INC, podemos forçar um lado da superposição a ser resolvido antes do outro.

Considere o seguinte exemplo com superposições aninhadas:

```
1 // Sem priorização, o resultado será 1 2 3 4
2 &{&{1, 2}, &{3, 4}}
3
4 // Com priorização, o segundo termo é resolvido primeiro,
5 // resultando em 3 4 1 2
6 &{&{1, 2}, ↑&{3, 4}}
```

3.9.1 Interações com INC/DEC

Os termos INC e DEC são tratados, principalmente, no algoritmo de collapse (HOC Team, 2025b), que ainda está em desenvolvimento, mas a sua base é colocar

os termos da rede em uma fila de prioridade, em que a prioridade é ajustada de acordo com termos INC/DEC que estão dentro de SUPs.

Interações com termos que não sejam SUP são simples, geralmente propagando o operador para "fora" de outros termos.

{INC,DEC}-DUP Duplicar um termo priorizado resulta em duas cópias priorizadas.

$$\frac{\begin{array}{c}\text{! \&L\{r s\} = } \uparrow f \\ K\end{array}}{\begin{array}{c}\text{! \&L\{f0 f1\} = } f \\ r \leftarrow \uparrow f0 \\ s \leftarrow \uparrow f1 \\ K\end{array}}$$

```
fn reduce_dup_inc(opx: Term, sup: Term) -> Term
fn reduce_dup_dec(opx: Term, sup: Term) -> Term
```

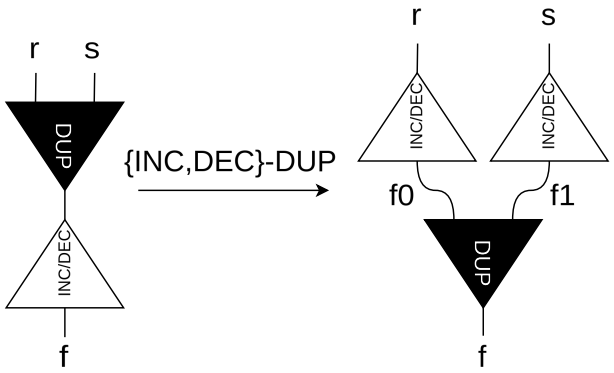

Figura 61 – Interação {INC,DEC}-DUP

{INC,DEC}-APP

$$\frac{(\uparrow f\ x)}{\uparrow(f\ x)}$$

```
fn reduce_app_inc(opx: Term, sup: Term) -> Term
fn reduce_app_dec(opx: Term, sup: Term) -> Term
```

{INC,DEC}-{MAT,SWI}

$$\frac{(\sim\uparrow t\ \{...\})}{\uparrow(\sim t\ \{...\})}$$

```
fn reduce_mat_inc(opx: Term, sup: Term) -> Term
fn reduce_mat_dec(opx: Term, sup: Term) -> Term
```

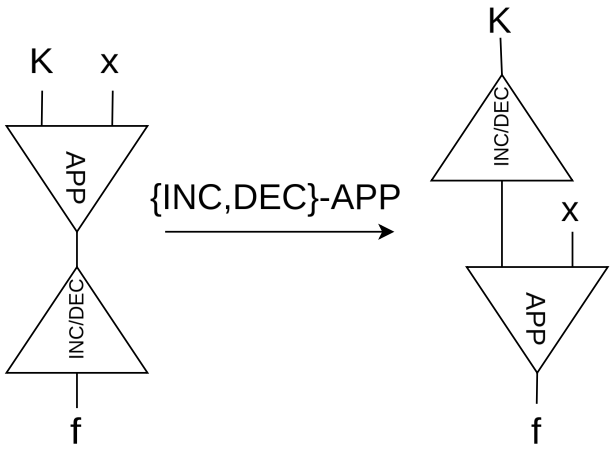

Figura 62 – Interação {INC,DEC}-APP

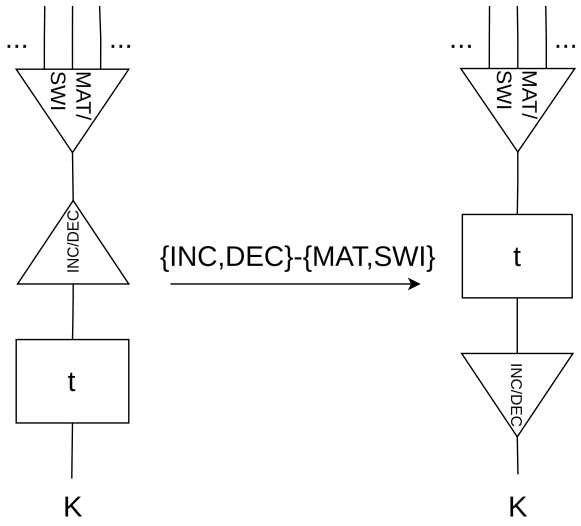

Figura 63 – Interação {INC,DEC}-{MAT,SWI}

{INC,DEC}-OPX

$$\frac{<op(\uparrow x\ y)}{\uparrow<op(x\ y)}$$

```
1 fn reduce_opx_inc(opx: Term, sup: Term) -> Term
2 fn reduce_opx_dec(opx: Term, sup: Term) -> Term
```

{INC,DEC}-OPY

$$\frac{>op(x\ \uparrow y)}{\uparrow>op(x\ y)}$$

```
1 fn reduce_opy_inc(opx: Term, sup: Term) -> Term
2 fn reduce_opy_dec(opx: Term, sup: Term) -> Term
```

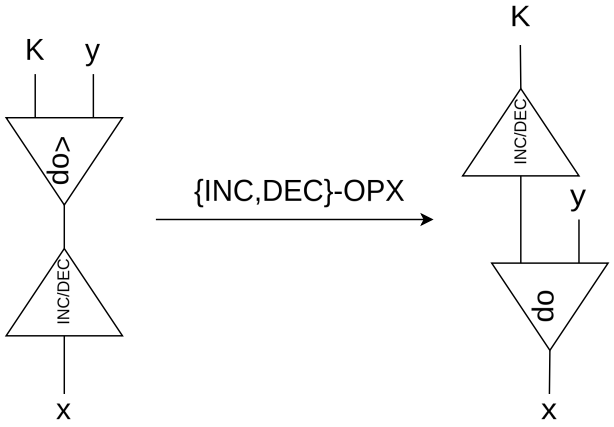

Figura 64 – Interação {INC,DEC}-OPX

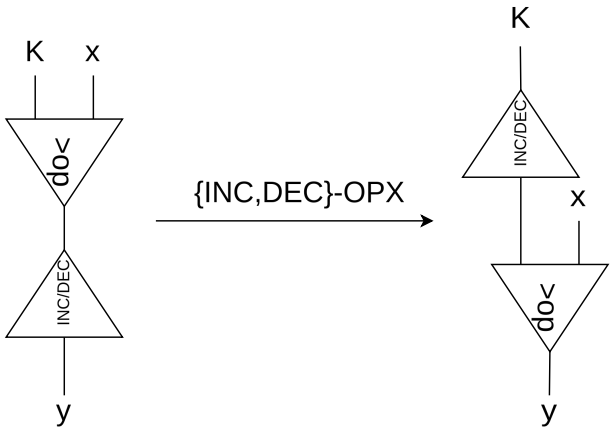

Figura 65 – Interação {INC,DEC}-OPY

3.10 Avaliando uma rede até o fim

Para avalizar uma rede até o fim, vamos assumir que o código fonte da rede já passou pela etapa de Parse e a struct “HVM” está carregada com a rede do programa a ser executado. O início de um programa se dá pela REF “@main”, que sempre deverá ser guardada no Loc 0 do heap. Então, basta reduzir recursivamente o Loc 0 até que não haja mais redexes.

Quando a HVM tenta reduzir um termo, ela navega pela rede em busca de um redex para interagir, expandindo REFs e linkando termos substituíveis (como VARs e DUPs). Durante esse processo, ela empilha os termos por onde passa. Se um redex é encontrado, ocorre a interação e o processo se repete tentando interagir o resultado obtido com o termo anterior que está na pilha. Se nenhum redex é encontrado e o caminho inteiro foi percorrido, o termo visitado já é irredutível (pelo menos naquele caminho). Então, a máquina precisa voltar pelo caminho que fez, atualizando os ponteiros dos termos na pilha para que eles se conectem dire-

tamente ao termo que foi reduzido.

```
fn run() -> Term {
  return reduce(HVM.heap[0])
}

fn is_whnf(lab: Lab) -> Bool {
  // Um termo está na sua Forma Normal de Cabeça Fraca (Weak
  // Head Normal Form) se ele possui um dos seguintes labels:
  // ERA, LAM, SUP, CTR, W32, CHR, INC, DEC
}

fn reduce(term: Term) -> Term {
  if (is_whnf(term_tag(term)))
    return term

   next: Term = term
   stop: u64 = HVM.spos // Fundo da pilha para a redução atual
   spos: u64 = stop

  while (true) {
    tag: Tag = term_tag(next)
    lab: Lab = term_lab(next)
    loc: Loc = term_loc(next)

    // Substituir variáveis e mover ‘next‘ para termos candidatos
    // a formarem um redex
    switch (tag) {
      case LET: {
        switch (lab) {
          case LAZY: {
            next = reduce_let(next, got(loc + 0))
            continue
          }
          case STRI: {
            spush(next, &HVM.sbuf, &spos)
            next = got(loc + 0)
            continue
          }
          default: erro()
        }
      }
      // Os termos abaixo nunca formam um redex entre si. Podemos
      // apenas colocá-los na pilha de redução e continuar
      // percorrendo a rede até encontrarmos um termo que possa
      // formar um redex.
      case APP, MAT, IFL, SWI, OPX, OPY: {
        spush(next, &HVM.sbuf, &spos)
        next = got(loc + 0)
        continue
      }
      case DP0, DP1: {
        sub: Term = got(loc + 0)
        if (term_get_bit(sub) == 0) {
          spush(next, &HVM.sbuf, &spos)
          next = sub
        } else {
          next = term_rem_bit(sub)
        }
        continue
      }
      case VAR: {
```

```
61          sub: Term = got(loc)
62          if (term_get_bit(sub) != 0) {
63            next = term_rem_bit(sub)
64            continue
65          }
66        }
67        case REF: {
68          HVM.spos = spos
69          next = reduce_ref(next)
70          spos = HVM.spos
71          continue
72        }
73        default: noop
74      }
75
76      // Se a pilha está vazia, o termo original alcançou sua WHNF
77      if (spos == stop) {
78        HVM.spos = spos
79        return next
80      }
81
82      prev: Term = spop(&HVM.sbuf, &spos)
83      switch (term_tag(prev)) {
84        case LET: {
85          next = reduce_let(prev, next)
86          continue
87        }
88        case APP: switch (tag) {
89          case ERA: {
90            next = reduce_app_era(prev, next)
91            continue
92          }
93          case LAM: {
94            next = reduce_app_lam(prev, next)
95            continue
96          }
97          case SUP: {
98            next = reduce_app_sup(prev, next)
99            continue
100         }
101         case CTR: {
102           next = reduce_app_ctr(prev, next)
103           continue
104         }
105         case W32, CHR: {
106           next = reduce_app_w32(prev, next)
107           continue
108         }
109         case INC: {
110           next = reduce_app_inc(prev, next)
111           continue
112         }
113         case DEC: {
114           next = reduce_app_dec(prev, next)
115           continue
116         }
117         default: noop
118       }
119       case DP0, DP1: switch (tag) {
120         case ERA: {
121           next = reduce_dup_era(prev, next)
```

```
122           continue
123         }
124         case LAM: {
125           next = reduce_dup_lam(prev, next)
126           continue
127         }
128         case SUP: {
129           next = reduce_dup_sup(prev, next)
130           continue
131         }
132         case CTR: {
133           next = reduce_dup_ctr(prev, next)
134           continue
135         }
136         case W32, CHR: {
137           next = reduce_dup_w32(prev, next)
138           continue
139         }
140         case INC: {
141           next = reduce_dup_inc(prev, next)
142           continue
143         }
144         case DEC: {
145           next = reduce_dup_dec(prev, next)
146           continue
147         }
148         default: noop
149       }
150       case MAT, IFL, SWI: switch (tag) {
151         case ERA: {
152           next = reduce_mat_era(prev, next)
153           continue
154         }
155         case LAM: {
156           next = reduce_mat_lam(prev, next)
157           continue
158         }
159         case SUP: {
160           next = reduce_mat_sup(prev, next)
161           continue
162         }
163         case CTR: {
164           next = reduce_mat_ctr(prev, next)
165           continue
166         }
167         case W32, CHR: {
168           next = reduce_mat_w32(prev, next)
169           continue
170         }
171         case INC: {
172           next = reduce_mat_inc(prev, next)
173           continue
174         }
175         case DEC: {
176           next = reduce_mat_dec(prev, next)
177           continue
178         }
179         default: noop
180       }
181       case OPX: switch (tag) {
182         case ERA: {
```

```
183           next = reduce_opx_era(prev, next)
184           continue
185         }
186         case LAM: {
187           next = reduce_opx_lam(prev, next)
188           continue
189         }
190         case SUP: {
191           next = reduce_opx_sup(prev, next)
192           continue
193         }
194         case CTR: {
195           next = reduce_opx_ctr(prev, next)
196           continue
197         }
198         case W32: {
199           case CHR: next = reduce_opx_w32(prev, next)
200           continue
201         }
202         case INC: {
203           next = reduce_opx_inc(prev, next)
204           continue
205         }
206         case DEC: {
207           next = reduce_opx_dec(prev, next)
208           continue
209         }
210         default: noop
211       }
212       case OPY: switch (tag) {
213         case ERA: {
214           next = reduce_opy_era(prev, next)
215           continue
216         }
217         case LAM: {
218           next = reduce_opy_lam(prev, next)
219           continue
220         }
221         case SUP: {
222           next = reduce_opy_sup(prev, next)
223           continue
224         }
225         case CTR: {
226           next = reduce_opy_ctr(prev, next)
227           continue
228         }
229         case W32, CHR: {
230           next = reduce_opy_w32(prev, next)
231           continue
232         }
233         case INC: {
234           next = reduce_opy_inc(prev, next)
235           continue
236         }
237         case DEC: {
238           next = reduce_opy_dec(prev, next)
239           continue
240         }
241         default: noop
242       }
243       default: noop
```

```
244     }
245
246     // Se chegar neste ponto do código, significa que nenhuma
247     // interação foi feita, então 'next' é um VAR ainda não
248     // ligado. Não há mais nenhuma redução a ser feita, mas
249     // precisamos atualizar os termos na stack com o resultado
250     // parcial que temos até agora.
251     // Tentar linkar o fim com o início da pilha.
252     spush(prev, &HVM.sbuf, &spos)
253     while (spos > stop) {
254       host: Term = spop(&HVM.sbuf, &spos)
255       set(term_loc(host) + 0, next)
256       next = host
257     }
258     HVM.spos = spos
259     return next
260   }
261 }
```

3.11 Estado Atual e Perspectivas Futuras da HVM3

A HVM3 *lazy* encontra-se em contínuo desenvolvimento. Embora mais madura que sua versão **strict**, permanece incerto o escopo das alterações antes da primeira versão estável, com seus algoritmos internos e sua sintaxe passando por revisões constantes, sintaxe essa que não é nem mesmo compartilhada completamente entre as versões. Mesmo assim, a equipe da Higher Order Company (HOC), responsável pelo projeto, tem como visão de longo prazo a unificação das abordagens *lazy* e *strict* em um único ecossistema, com a ambição de criar um avaliador ótimo e massivamente paralelo, assim aproveitando as vantagens de ambas as versões.

Diante deste cenário de evolução rápida, é natural que o conteúdo técnico desta monografia se torne obsoleto em um futuro próximo. Portanto, para garantir a precisão e a reprodutibilidade do trabalho aqui apresentado, ressalta-se que toda a análise, documentação e exemplos de código foram baseados na versão do repositório da HVM3 da HigherOrderCO (2025), correspondente ao *commit* de *hash* `290d5127ef58707a71c4e70b6a1407004679acc6`.

4 Conclusão

Este trabalho teve como objetivo principal a exploração e documentação da *Higher-order Virtual Machine* (HVM3), um modelo de computação baseado em redes de interação. Foi apresentada uma sequência de submáquinas que permitem implementar a HVM3 incrementalmente, já sendo possível calcular qualquer função computável a partir da sua versão 1, apesar da baixa eficiência dessa versão. Buscou-se desmistificar seu funcionamento, desde o modelo conceitual até as especificidades do seu código-fonte, com um foco educacional e detalhado. Para alcançar esse objetivo, o trabalho foi estruturado de forma a construir o conhecimento progressivamente, partindo dos fundamentos teóricos do cálculo lambda e das redes de interação, até a análise aprofundada das inovações introduzidas pela HVM3.

A metodologia adotada envolveu uma revisão bibliográfica para solidificar os conceitos de base, seguida por um estudo direcionado ao repositório da HVM3, em particular ao seu *runtime*. Este processo permitiu a criação de uma documentação em português, que não apenas traduz, mas também contextualiza e explica o código e os conceitos da HVM3, utilizando uma abordagem passo a passo. O desenvolvimento desta monografia concentrou-se em documentar as adições que a HVM3 faz aos combinadores de interação para tornar esse modelo de computação eficiente na prática.

As principais contribuições deste trabalho podem ser resumidas em:

- **Documentação didática:** Foi criada uma documentação clara e acessível não apenas sobre a HVM3, mas também sobre os conceitos necessários para compreendê-la, preenchendo assim uma lacuna na comunidade lusófona de programação e facilitando a documentação do projeto em outros idiomas futuramente. A explicação detalhada de cada componente e regra de interação visa a facilitar o aprendizado para pesquisadores e estudantes com um nível básico de programação.
- **Modelo Conceitual da HVM3:** O trabalho elucidou o modelo conceitual da HVM3, demonstrando como ela expande os combinadores de interação para suportar funcionalidades que facilitam a expressividade e tendem a ser mais eficientes se representadas em estruturas mais próximas de instruções nativas dos processadores modernos, como operadores nativos e tipos de dados algébricos.

Ao longo do desenvolvimento, ficou evidente a capacidade da HVM3 de realizar redução ótima de termos do cálculo-λ. Isso, unido com a natureza inerentemente paralela das redes de interação, são características que a tornam uma alternativa promissora às arquiteturas de computação tradicionais para certas classes de problemas, especialmente aqueles que podem ser massivamente paralelizados.

4.1 Trabalhos Futuros

Neste trabalho, foi contemplado, principalmente, o *runtime* que acompanha o compilador. Uma direção natural para um trabalho futuro seria explorar, também, o parser da linguagem e o processo de compilação de funções definidas em termos REF.

Outra possibilidade, seria comparar o funcionamento da HVM3 *Lazy* com a HVM3 *Strict*, quando esta estiver em uma fase mais avançada de desenvolvimento, incluindo também benchmarks de desempenho.

Com a mesma motivação de contribuir com a comunidade do projeto, a criação de ferramentas de desenvolvimento para a HVM3, como depuradores visuais que mostrem a evolução da rede de interação passo a passo ou programas de *Language Server Protocol* (LSP), seriam adições valiosas para a usabilidade e adoção desta tecnologia.

Por fim, poderiam ser investigados novos tipos de termos ou regras de interação para otimizar ainda mais a execução de certos algoritmos ou para adicionar novas funcionalidades à máquina virtual, como suporte a tipos de dados mais complexos ou novas operações matemáticas, expandindo, assim, as funcionalidades nativas da HVM3.

Referências

ABRIL, N. **Funcionamento do OP2**. 2025. Acessado: 2025-09-02. Disponível em: <https://discord.com/channels/912426566838013994/915345481675186197/1398259391479939195>. Citado na página 61.

____. **Utilidade do Lazy LET**. 2025. Acessado: 2025-09-02. Disponível em: <https://discord.com/channels/912426566838013994/915345481675186197/1411850157477068943>. Citado na página 79.

ABRIL, N.; BATTISTELA, L. W. **Semântica de INC/DEC**. 2025. Acessado: 2025-09-02. Disponível em: <https://discord.com/channels/912426566838013994/915345481675186197/1410635526540890186>. Citado na página 80.

ASPERTI, A.; GUERRINI, S. **The Optimal Implementation of Functional Programming Languages**. [S.l.: s.n.], 1999. ISBN 0521621127. Citado na página 31.

AïT-KACI, H. **Warren's Abstract Machine: A Tutorial Reconstruction**. [s.n.], 1991. 939 p. ISBN 9780262255585. Disponível em: <https://doi.org/10.7551/mitpress/7160.001.0001>. Citado 2 vezes nas páginas 42 e 44.

BATTISTELA, L. W. **Sintaxe do SWI**. 2025. Acessado: 2025-09-02. Disponível em: <https://discord.com/channels/912426566838013994/915345481675186197/1402299942952763412>. Citado na página 68.

BöHM, C.; GROSS, W. The cuch as a formal and description language em formal language description languages for computer programming. **Journal of Symbolic Logic**, North-Holland Publishing Company, v. 40, n. 1, p. 179–197, 1966. Disponível em: <https://doi.org/10.2307/2272275>. Citado na página 19.

CHURCH, A. A set of postulates for the foundation of logic. **Annals of Mathematics**, Annals of Mathematics, Trustees of Princeton University on Behalf of the Annals of Mathematics, Mathematics Department, Princeton University, v. 33, n. 2, p. 346–366, 1932. ISSN 0003486X, 19398980. Disponível em: <https://doi.org/10.2307/1968337>. Citado 2 vezes nas páginas 13 e 16.

____. A set of postulates for the foundation of logic. **Annals of Mathematics**, Annals of Mathematics, v. 34, n. 4, p. 839–864, 1933. ISSN 0003486X. Disponível em: <https://doi.org/10.2307/1968702>. Citado na página 18.

HigherOrderCO. **HVM3**. 2025. Acessado: 2025-09-02. Disponível em: <https://github.com/HigherOrderCO/HVM3/commit/3576f0ebf189c65f98e223164b1d29f36e537bda>. Citado na página 89.

HINDLEY, J. R.; SELDIN, J. P. **Lambda-Calculus and Combinators: An Introduction**. 2. ed. Cambridge University Press, 2008. Disponível em: <https://doi.org/10.1017/CBO9780511809835>. Citado na página 16.

HOC Team. **Linear Readback**. 2024. Acessado: 2025-04-08. Disponível em: <https://github.com/HigherOrderCO/Bend/blob/75fec29e75015e2f235e9b831866234ff2db4f1e/src/term/readback/linear.rs>. Citado na página 37.

____. **Bend compilation and readback**. 2025. Acessado: 2025-04-08. Disponível em: <https://github.com/HigherOrderCO/Bend/blob/58372e05fb96ec228cd09310f6a6418f1967a48f/docs/compilation-and-readback.md>. Citado na página 37.

____. **HVM3 Collapse.hs**. 2025. Acessado: 2025-08-30. Disponível em: <https://github.com/HigherOrderCO/HVM3/blob/3576f0ebf189c65f98e223164b1d29f36e537bda/src/HVM/Collapse.hs#L495>. Citado na página 81.

KAHL, W. **HINet: Interaction Nets in Haskell**. 2015. Acessado: 2025-04-16. Disponível em: <https://www.cas.mcmaster.ca/~kahl/Haskell/HINet/>. Citado na página 43.

LAFONT, Y. Interaction nets. In: **Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1989. (POPL '90), p. 95–108. ISBN 0897913434. Disponível em: <https://doi.org/10.1145/96709.96718>. Citado 2 vezes nas páginas 13 e 19.

____. Interaction combinators. **Information and Computation**, v. 137, n. 1, p. 69–101, 1997. ISSN 0890-5401. Disponível em: <https://doi.org/10.1006/inco.1997.2643>. Citado na página 27.

LANDIN, P. J. The mechanical evaluation of expressions. **The Computer Journal**, v. 6, n. 4, p. 308–320, 01 1964. ISSN 0010-4620. Disponível em: <https://doi.org/10.1093/comjnl/6.4.308>. Citado na página 42.

MAZZA, D. A denotational semantics for the symmetric interaction combinators. **Mathematical Structures in Computer Science**, v. 17, n. 3, p. 527–562, 2007. Disponível em: <https://doi.org/10.1017/S0960129507006135>. Citado na página 28.

TAELIN, V. **Interaction-Calculus**. 2025. Acessado: 2025-04-02. Disponível em: <https://github.com/VictorTaelin/Interaction-Calculus>. Citado na página 28.

Vine Team. **The Vine Programming Language**. 2025. Acessado: 2025-04-16. Disponível em: <https://vine.dev/>. Citado na página 43.