UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TÉCNOLÓGICAS – CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO – BCC

KARLA ALEXSANDRA DE SOUZA JORIATTI

INTERPRETANDO EFEITOS ALGÉBRICOS POR MEIO DE MÔNADAS

JOINVILLE
2024

KARLA ALEXSANDRA DE SOUZA JORIATTI

INTERPRETANDO EFEITOS ALGÉBRICOS POR MEIO DE MÔNADAS

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Cristiano Damiani Vasconcellos
Coorientador: Paulo Henrique Torrens

JOINVILLE
2024

KARLA ALEXSANDRA DE SOUZA JORIATTI

INTERPRETANDO EFEITOS ALGÉBRICOS POR MEIO DE MÔNADAS

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Cristiano Damiani Vasconcellos
Coorientador: Paulo Henrique Torrens

BANCA EXAMINADORA:

Orientador:

Dr. Cristiano Damiani Vasconcellos
UDESC

Coorientador:

Me. Paulo Henrique Torrens
University of Kent

Membros:

Dra. Karina Girardi Roggia
UDESC

Me. Gabriela Moreira

Joinville, Novembro de 2024

"*Responderemos. Olho por olho e dente por Dente. A luta já começou.*" (Carlos Marighela – Rádio Libertadora, [1969])

RESUMO

Palavras-chave: SSA, Mônadas, Efeitos, Programação Funcional.

ABSTRACT

Keywords: SSA, Monads, Efects, Functional Programming.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IR	Intermediate Representation
CFG	Control Flow Graph
SSA	Static Single Assignment
ANF	Administrative Normal Form
CPS	Continuation Passing Style

SUMÁRIO

1 INTRODUÇÃO

Os compiladores são parte fundamental no âmbito da ciência da computação, desempenhando um papel essencial na tradução de código em linguagem de programação (e.g., Java e Haskell) para código de baixo nível (e.g., bytecode). Para além da simples tradução de código, os compiladores são capazes de otimizar o programa fonte com o objetivo de melhorá-lo de acordo com algum critério de desempenho. Dessa forma, muitos compiladores são desenvolvidos para gerar uma representação intermediária que será utilizada para aplicar diversas otimizações a partir dela.

Para criar um compilador, uma série de análises deve ser feita. Durante esse processo de criação, uma de suas etapas visa a otimização do programa compilado. Nessa etapa a escolha adequada de uma estrutura de dados para representação intermediária terá influência direta no poder e eficiência do compilador. Um exemplo de representação intermediária é o grafo de fluxo de controle, que utiliza a notação de grafos para gerar todos os caminhos possíveis que podem ocorrer durante a execução de um programa (ALLEN, 1970). A forma de atribuição estática única é um caso particular de grafo de fluxo de controle onde cada uso de variável está ligado a uma única atribuição. Tal representação é utilizada predominantemente nos compiladores de linguagens imperativas (e.g., GCC e Clang) para aplicação de otimizações baseadas em análise de fluxo de dados (CYTRON et al., 1989).

Em 1998, Appel demonstra, a partir da análise da imutabilidade de variáveis em linguagens funcionais, que a forma de atribuição estática única é equivalente a um programa em linguagem funcional (APPEL, 1998). Tais linguagens utilizam as definições de cálculo lambda apresentadas por Church (1932) como base, programando por meio de funções aninhadas. Ademais, programas em linguagens funcionais apresentam uma série de vantagens que podem ser aproveitadas na etapa de otimização, como, por exemplo, códigos funcionais puros separados de códigos com efeitos (Haskell e Miranda), modularidade (e.g., utilização de funções de ordem superior para "colar"funções e método de avaliação preguiçosa que permite "colar"programas), garantias comuns a linguagens funcionais (principalmente linguagens funcionais puras), entre outros (HUGHES, 1989).

Em linguagens funcionais puras, como Haskell e Miranda, os programas escritos não produzem efeitos colaterais, ou seja, não possuem alterações em estados de memória durante a computação. Para poder lidar com efeitos dentro de linguagens puras, Wadler (1995) propõe o uso de mônadas, uma abstração matemática proveniente do estudo de Teoria das Categorias. Outra forma de lidar com efeitos foi apresentada por Leijen (2014), introduzindo o conceito de inferência e tratamento de efeitos por meio de um Sistema de Tipos e Efeitos.

Diversos trabalhos basearam-se na descoberta de Appel sobre a equivalência entre a forma de atribuição estática única e linguagens funcionais. Dentre estas contribuições, vale mencionar o trabalho de Torrens, Vasconcellos e Gonçalves (2017), que sugerem uma linguagem intermediária capaz de ser interpretada tanto como um grafo de fluxo de controle na forma de

atribuição estática única quanto como uma representação funcional pura. E, além do trabalho supracitado, Rigon, Torrens e Vasconcellos (2020) também criam uma representação funcional a partir de uma representação na forma de stribuição estática única utilizando um sistema de efeitos colaterais. O sistema de tipos e efeitos apresentado por Rigon, Torrens e Vasconcellos (2020), baseado no trabalho de Leijen (2014), possui as vantagens de uma linguagem impura, porém ainda permite uma menor incidência de erros devido a identificação de trechos de código que causam efeitos colaterais. O sistema de efeitos impõe que as generalizações sejam aplicadas somente a computações puras e ainda garante que expressões com efeitos colaterais estejam encapsuladas de forma a não afetar o restante do programa. Além do sistema de efeitos, foi aplicada a transformação da forma de atribuição estática única para uma representação na forma-normal-A capaz de deixar um programa mais natural e intuitivo e permitir a compilação de efeitos, oposto a técnica mais comumente utilizada, a passagem para *continuation-passing-style*, onde os efeitos serão removidos após a transformação (FLANAGAN et al., 1993).

O objetivo geral deste trabalho é apresentar uma versão de representação funcional intermediária equivalente a apresentada por Rigon, Torrens e Vasconcellos, porém com tratamento de efeitos por meio do uso de mônadas. A tradução de código funcional com sistema de efeitos para um código funcional que utiliza mônadas será baseado nos conceitos demonstrados por Vazou e Leijen (2016), que apresentam a formalização da tradução do sistema de efeitos da linguagem de programação koka para mônadas.

O presente trabalho irá abordar o assunto partindo da fundamentação teórica, apresentada pelo capítulo 2, e finalizará com a proposta do trabalho, presente no capítulo 3, utilizando X exemplos de tradução de código de sistema de efeitos para código monádico.

2 FUNDAMENTAÇÃO TEÓRICA

Este capitulo concentra-se nos aspectos teóricos relacionados ao trabalho, definindo alguns conceitos-chave para a compreensão do tema abordado. Será fornecida, na seção 2.1, uma visão geral sobre representação intermediária de código, estendendo a noção de grafos de fluxo de controle, sua versão na forma de atribuição estática única e representações em λ-cálculo. A conversão da forma de atribuição estática única para código funcional, abordando o trabalho de Appel (1998), será descrita pela seção 2.2, seguindo para a definição de efeitos colaterais e seus tratamentos em linguagens funcionais, apresentando os conceitos de mônadas e sistemas de efeitos, na seção 2.3, e finalizando com a conversão de um sistema de efeitos para código monádico, fundamentado no trabalho de Vazou e Leijen (2016) abordados na seção 2.4. Para melhor compreensão, será definido que: **(1)** um código base irá se referir ao código imediatamente anterior a representação intermediária, podendo ser o próprio código-fonte ou outra representação; **(2)** um código-fonte será o primeiro código recebido pelo compilador; e **(3)** código alvo é o código mais próximo em linguagem de máquina, ou o último código que o compilador retorna.

2.1 REPRESENTAÇÃO INTERMEDIÁRIA DE CÓDIGO

Um compilador funciona como uma série de etapas que levam do código-fonte até o código de máquina, também conhecido como código-alvo, por meio da troca de informações e traduções entre uma etapa e outra. Para haver coerência entre as informações é necessário que essa passagem seja feita utilizando representações equivalentes. Uma estrutura de dados gerada pelo compilador, que representa o programa fonte, é chamada de Representação Intermediária (do inglês, *intermediate representation*, ou IR) (COOPER; TORCZON, 2014). Um compilador pode utilizar uma ou mais IRs, podendo variar desde uma estrutura em árvore ou grafo até alguma outra linguagem de programação, como C e algumas representações de código funcionais (AHO et al., 2008).

As IRs permitem diferentes distribuições entre as etapas de um compilador. Classicamente, um compilador é decomposto em duas fases principais, front-end e back-end, ligadas por meio de uma IR, como apresentada pela figura 1. Neste modelo o front-end irá codificar suas informações a partir do código-fonte para uma IR e repassar ou diretamente para o back-end ou para outra IR que repassará para o back-end posteriormente, onde ocorrerá o mapeamento dessa codificação para o código alvo. Para além do modelo em duas fases, a introdução de múltiplas IRs possibilita acrescentar fases ao projeto de um compilador, entre o front-end e o back-end. Denominada *otimizador*, a fase intermediária recebe como entrada uma IR e retorna também uma IR semanticamente equivalente à entrada. Esta fase se caracteriza pela aplicação de otimizações e transformações IR-para-IR. O modelo estruturado utilizando a fase de otimizador é conhecido como compilador de três fases (figura 2).

Em termos de classificação, as IRs podem ser agrupados por meio de suas características.

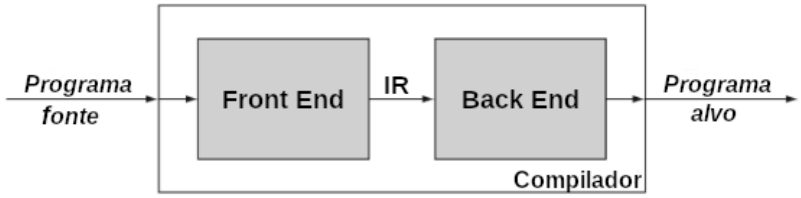

Figura 1 – Compilador de duas fases
Fonte: (COOPER; TORCZON, 2014)

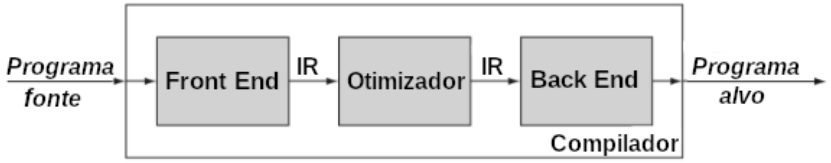

Figura 2 – Compilador de três fases
Fonte: (COOPER; TORCZON, 2014)

No domínio estrutural encontram-se as classes de IRs gráficas, onde todo conhecimento repassado é codificado em um grafo. Nestas representações todo algoritmo deve ser expresso por meio de objetos gráficos, como: nós, arestas, listas ou árvores. Outra classe capaz de caracterizar estruturalmente uma representação são as IRs lineares. Muito semelhantes aos pseudocódigos de máquinas abstratas, IRs lineares organizam-se por meio de sequências de operações lineares simples. Para além da necessidade de escolher uma abordagem ou outra, é possível fazer a combinação de representações lineares e gráficas em uma única interpretação. Estas são as IRs híbridas. Possibilitando a maximização dos pontos fortes e a eliminação de fraquezas de cada categoria, esta abordagem comumente utilizará uma representação linear de baixo nível associada a grafos para capturar blocos de código em linha e fluxos de controle, respectivamente (COOPER; TORCZON, 2014).

2.1.1 Grafo de Fluxo de Controle

Uma IR baseada em controle e análise de fluxo de dados é capaz de capturar informações sobre o fluxo de dados ao longo dos caminhos de execução do programa e verificar as alterações nos fluxos de controle geradas a partir de expressões booleanas (AHO et al., 2008). Estas informações permitem que uma série de otimizações sejam aplicadas sobre o programa (e.g., eliminação de fluxo de controle inútil e eliminação de sub-expressão comum).

O grafo de fluxo de controle (do inglês, *control flow graph*, ou CFG) é um exemplo de IR para análise de controle e fluxo de dados modelado como um grafo dirigido $G = (N, E)$, representando, respectivamente, blocos de código básicos n, de forma que $n \in N$, e a transferência de controles de um bloco a outro, modelado como uma aresta e, tal que $e \in E$. Um bloco básico é uma sequência linear de código onde não ocorrem ramificações no grafo, podendo representar

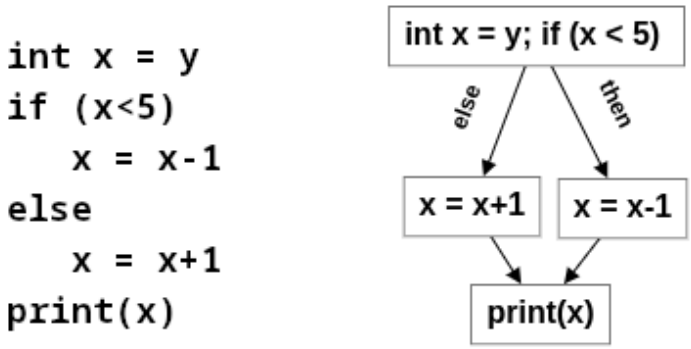

Figura 3 – Modelagem de *If Statement* em CFG
Fonte: Elaborado pelo autor.

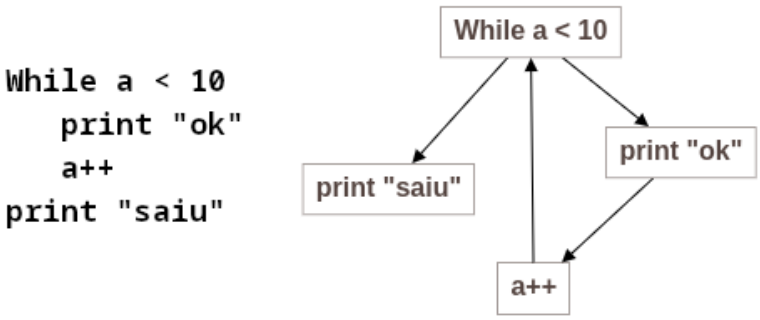

Figura 4 – Modelagem de *While Statement* em CFG
Fonte: Elaborado pelo autor.

mais de uma linha do código-fonte traduzido, como mostra a figura 3 (ALLEN, 1970).

Uma das características mais comuns em CFGs são as ramificações. Elas irão ocorrer sempre que uma expressão booleana for avaliada pelo programa, como demonstrado na figura 4, onde (`a < 10`) é avaliado e o fluxo é ramificado para (`print "saiu"`) e (`print "ok"`). Além disso, a figura 4 também apresenta um loop no grafo, identificado por meio da revisitação do nó (`while a < 10`) partindo de si, o que representa um laço de repetição no programa-fonte. Além das ramificações de saída, é possível que um nó seja alcançado por mais de um nó predecessor, como mostra a figura 3, onde (`print(x)`) pode ser acessado pelos nós (`x = x+1`) e (`x = x-1`).

Dentre as opções de modelagem possíveis para um CFG, a escolha de tipo de bloco a ser implementado é de extrema importância. É possível utilizar blocos de instrução única em substituição a implementação comum de bloco básico, como no caso apresentado na figura 4, onde cada bloco representa uma única instrução independente de gerar ramificação no grafo ou não. Essa escolha de projeto pode simplificar algoritmos para análise e otimização, contudo utiliza mais memória e leva mais tempo para percorrer do que a modelagem utilizando blocos básicos.

Um compilador pode utilizar um CFG, explicita ou implicitamente, em diversas etapas para auxiliar no processo de otimização por análise de fluxos. Outros grafos também podem ser derivados a partir do CFG, fornecendo diferentes tipos de informações que auxiliam na

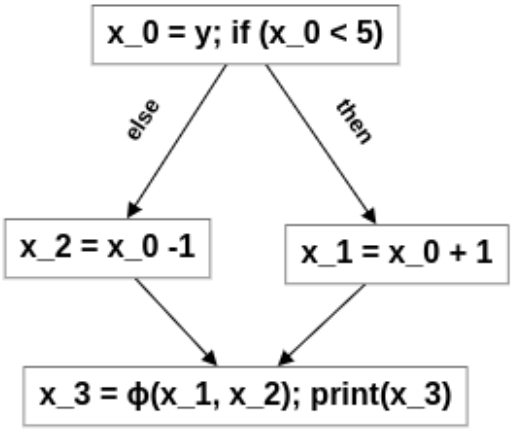

Figura 5 – Modelagem de *If Statement* em SSA
Fonte: Elaborado pelo autor.

realização de análises mais profundas (e.g., grafo de dependência e grafo de chamada). Para além das derivações, CFGs também podem possuir casos específicos, com regras próprias, assim como os casos de atribuição estática única.

2.1.2 Forma de Atribuição Estática Única

A forma de atribuição estática única (do inglês, *static single assignment*, ou SSA) representa uma particularidade de CFG bastante utilizada como IR em compiladores de linguagens imperativas. Neste formato, será permitido, somente, que cada referência a uma variável esteja relacionada a uma única atribuição. Para que essa propriedade seja mantida é necessário um procedimento de renomeação de todas as variáveis com mais de uma atribuição no código base.

A figura 5 apresenta a modelagem do mesmo código representado anteriormente pela figura 3, porém agora em sua forma SSA. É possível notar que a variável `x`, utilizada diversas vezes no trecho de código base, foi subdividida em outras quatro variáveis (`x_0`, `x_1`, `x_2`, `x_3`) fragmentadas de modo a estabelecer o critério de atribuição única. Ao renomear estas variáveis, surge o problema de mais de um fluxo convergindo para um mesmo bloco, assim como em (`print(x)`) apresentado na figura 3. Este bloco de código é diretamente dependente de seus dois blocos predecessores. Na forma SSA, para decidir qual variável deve ser utilizada, dentre `x_1` e `x_2`, uma função φ é implementada.

Uma função φ recebe como entrada as atribuições provenientes de diferentes arestas que fluem para um mesmo bloco. Quando o fluxo de controle passa para este bloco, a função é executada e o retorno será obtido a partir da aresta pela qual o controle entrou no bloco. Durante a passagem do código base para a forma SSA, o algoritmo deve ser capaz de renomear todas as variáveis alteradas e inserir funções φ em cada ponto de junção do CFG de forma eficiente. Para gerar uma representação SSA eficiente é necessário que as funções φ sejam colocadas somente

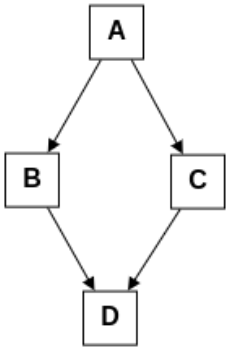

Figura 6 – Renomeação de blocos da Figura 3
Fonte: Elaborado pelo autor.

	A	B	C	D
D_{OM}	{A}	{A,B}	{A,C}	{A,D}
ID_{OM}	–	A	A	A

Tabela 1 – Tabela de dominância
Fonte: Elaborado pelo autor.

onde é necessário, pois o excesso destas funções aumenta o custo de qualquer algoritmo que precise percorrer grafo na forma SSA.

Uma maneira de auxiliar o algoritmo para criar a forma SSA com um número menor de funções é realizar uma análise de dominância entre os blocos. A partir da Figura 3, renomeando os blocos e abstraindo-se das informações internas a ele, foram criadas a tabela de dominância (Tabela 1) e a árvore de dominadores (Figura 7) para o CFG. A tabela 1 nos apresenta dois conceitos: dominância, dado por $D_{OM}(n)$, onde $m \in D_{OM}(n)$ se, para alcançar n é *necessário* passar por m; e dominante imediato, dado por $ID_{OM}(n)$, de forma que se $m \in D_{OM}(n)$ e m é o nó mais próximo de n neste conjunto, com $m \neq n$, então m é o dominante imediato de n. Se $m \in D_{OM}(n)$ e $m \neq n$ então m domina estritamente n, denotado por $m \gg n$.

Partindo da tabela de dominância é possível criar a árvore de dominadores, de forma que se $m \in ID_{OM}$ então existe uma aresta direcionada saindo de m para n. Como o bloco de entrada do grafo de fluxo não possui dominante imediato, este bloco será a raiz da árvore de dominadores.

O próximo passo é calcular as fronteiras de dominância, ou DF, para cada nó no grafo como apresentada pela tabela 2. A fronteira de dominância de um dado nó n é dada por todo nó m no grafo, de modo que n domina um predecessor de m, mas não domina estritamente m (CYTRON et al., 1989).

$$DF(n) = \{m \quad | \quad (\exists P \in Pred(m)) \quad (n \in D_{om}(P) \quad e \quad n \not\gg m)\} \tag{1}$$

Obtidas as fronteiras de dominância, o posicionamento de funções φ no grafo torna-se trivial. Se existe uma definição de x no nó n, então deverá existir uma função φ em cada bloco

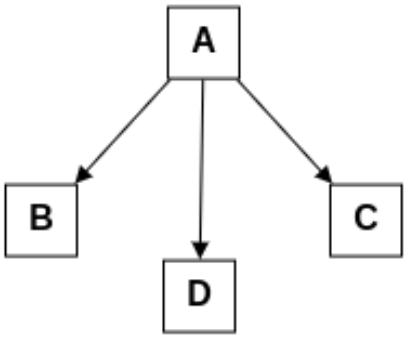

Figura 7 – Árvore de dominadores

	A	B	C	D
DF	–	{D}	{D}	–

Tabela 2 – Tabela de fronteira de dominância
Fonte: Elaborado pelo autor.

pertencente a $DF(n)$. O procedimento de cálculo de fronteiras de dominância é uma forma básica de melhorar a representação na forma SSA, podendo ser utilizado por outros algoritmos para tornar a inserção de funções φ ainda mais eficiente.

2.1.3 Forma-Normal-A

A utilização de IRs baseadas em λ-cálculo ocorre em diversos compiladores para linguagens funcionais. No trabalho de Appel e MacQueen (1987), um tipo de λ-cálculo modificado é utilizado como IR em um compilador padrão para a linguagem funcional ML. Definido por Church (1932), o λ-cálculo puro é a representação de aplicações de funções, variáveis e abstrações.

M := (M M)	(Aplicação)
| (λx.M)	(Abstração)
| x	(Variável)

Um termo em λ-cálculo pode ser, portanto, a aplicação de um termo sobre outro, uma abstração ou uma variável. Em uma abstração do tipo $(\lambda x.M)$, é dito que a variável x está ligada ao termo M e toda ocorrência de uma variável dentro de M que não está ligada é uma variável livre. O conjunto de variáveis livres de um termo M é definido como $FV(M)$.

Em um termo em λ-cálculo é possível aplicar substituições. Uma substituição é representada como $M[x := N]$, de forma que toda ocorrência de x em M será substituída por N. Outro

conceito, para além da substituição, é a noção de termos α-equivalentes. É possível demonstrar a equivalência entre dois termos a partir da mudança de variáveis que estão ligadas em uma abstração, e.g., os termos $(\lambda x.x)$ e $(\lambda y.y)$ são α-equivalentes, pois, apenas trocando o nome de suas variáveis ligadas, podemos chegar de um termo a outro. Esta mudança de variáveis ligadas é chamada de α-conversão ou α-redução. Uma aplicação bastante comum de substituição e α-conversão é o uso em β-reduções. Uma β-redução ocorre sempre que uma abstração recebe argumentos, ou seja, a função é aplicada a seus argumentos.

$$(\lambda \text{x.M})\ \text{N} := \text{M[x:=N]} \qquad (\beta\text{-redução})$$

Caso uma variável que ocorre livre em N apareça em alguma λ-abstração de M, então uma α-conversão deve ser feita no termo antes de aplicar a substituição, para que assim não ocorra a ligação desta variável. Por exemplo, a β-redução do termo $((\lambda x \lambda y.x)y)$ será $(\lambda z.y)$, com a mudança da variável ligada de y para z.

Outra redução bastante utilizada em λ-termos é a η-redução, que ocorre como uma noção de extensionalidade. Nesta redução uma abstração do tipo $(\lambda x.fx)$ é convertida apenas para f, desde que $x \notin FV(f)$. Isso ocorre pela consideração de equivalência entre termos que ao serem aplicados geram o mesmo resultado. Adicionalmente, existe a possibilidade de incrementar o λ-cálculo com a definição de contextos. Um contexto é um termo com um "buraco", representado por "[]", que pode ser preenchido por uma sub-expressão. O ato de fazer este preenchimento é chamado de *filling*, e.g., o *filling* do contexto $(\lambda x.y[])$ com o termo $(\lambda z.x)$ é o termo $(\lambda x.y(\lambda z.x))$. O preenchimento acontece entre um contexto e um termo e resulta também em um termo, podendo ligar variáveis que antes eram livres. A classe sintática que define a gramática para contextos é apresentada a seguir.

$$\text{E} = [] \mid \text{E}\ ([]\ \text{M})$$

A forma-normal-A (do inglês, *Administrative normal form*, ou ANF) é uma versão de cálculo lambda utilizada como IR. Um λ-termo estará em ANF se não puderem ser aplicadas mais reduções do tipo A. Sabry e Felleisen (1992) definem o conjunto de reduções A pelas regras:

$$\begin{array}{lll}
E[((\lambda x.M)N)] \longrightarrow ((\lambda x.E[M])N) & x \notin FV(E) & (\beta_{lift}) \\
E[((MN)L)] \longrightarrow ((\lambda x.E[L])(MN)) & x \notin FV(E,L) & (\beta_{flat}) \\
((\lambda x.x)M) \longrightarrow M & & (\beta_{id}) \\
((\lambda x.E[(yx)])M) \longrightarrow E[(yM)] & x \notin FV(E[y]) & (\beta_{\Omega})
\end{array}$$

Tais regras derivam na seguinte gramática para λ-cálculo que respeita as restrições impostas por ANF:

```
E := V
   | let x = V in E
   | let x = V V in E
V := x
   |λx. E
```

Além disso, Flanagan et al. (1993) demonstram a equivalência entre as máquinas abstratas para compiladores que utilizam *continuation-passing-style* (CPS) e ANF, sendo que a máquina abstrata para ANF utiliza um número menor de passos para alcançar o objetivo. A utilização de continuações para compilar de forma eficiente uma linguagem funcional foi apresentada, inicialmente, por Steele (1978) ao criar um compilador para Scheme (dialeto de Lisp). Entretanto, a ideia se popularizou com Appel, em seu livro "Compiling with Continuantions"(APPEL, 2007). Alguns artigos na área de continuações apresentam noções de isomorfismo entre as representações já discutidas no presente trabalho. No artigo de Kelsey (1995) uma proposta de conversão de CPS para SSA é apresentada a partir de funções de tradução CPS$\longrightarrow$SSA e SSA$\longrightarrow$CPS; Chakravarty, Keller e Zadarnowski (2004) criam uma formalização do mapeamento de programas na forma SSA para ANF; e Sabry e Felleisen (1992) comprovam a correspondência entre ANF e o λ-cálculo de Moggi, λ_c (MOGGI, 1988).

2.2 CONVERSÃO DE SSA PARA CÓDIGO FUNCIONAL

Como visto na seção 2.1.2, um programa em SSA terá cada variável ligada a uma única atribuição. Para que isso ocorra, é preciso renomear as variáveis utilizadas no código e implementar funções φ para definir o valor de variáveis que pertencem a pontos de junção no grafo. Para demonstrar a correspondência entre SSA e código funcional, Appel (1998) se utiliza de alguns exemplos de códigos em SSA, o qual também será aqui utilizado. Seja o seguinte trecho de código em uma linguagem de programação qualquer:

```
i <- 1
j <- 1
k <- 0
while k < 100
    if j < 20
        j <- i
        k <- k + 1
    else
        j <- k
        k <- k + 2
return j
```

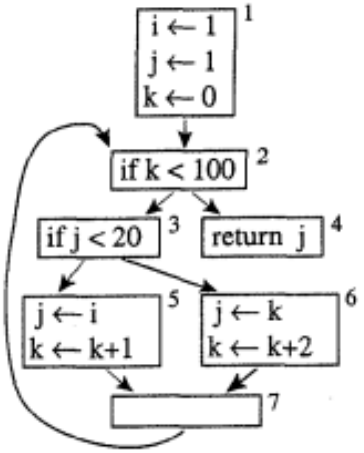

Figura 8 – Modelagem CFG
Fonte: (APPEL, 1998)

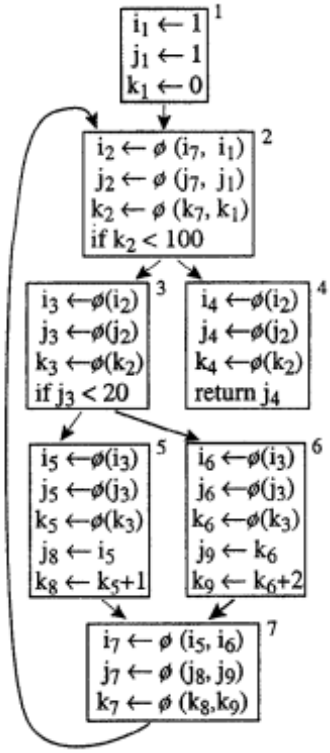

Figura 9 – Modelagem SSA
Fonte: (APPEL, 1998)

Sua versão em CFG é apresentada pela figura 8 e a versão em SSA pela figura 9. Outra forma de visualizar o programa na forma SSA é como um conjunto de funções mutuamente recursivas, onde cada função recebe três argumentos, com exceção da função inicial. O seguinte trecho de código funcional é uma representação correspondente a forma SSA.

```
f1 = let i1 = 1
         j1 = 1
         k1 = 0
```

```
      in f2 i2 j1 k1
f2 i2 j2 k2 = if k2 < 100 then (f3 i3 j3 k3) else (f4 i2 j2 k2)
f3 i3 j3 k3 = if j3 < 20 then (f5 i3 j3 k3) else (f6 i3 j3 k3)
f4 i4 j4 k4 = j4
f5 i5 j5 k5 = let j8 = i5
                  k8 = k5 + 1
              in (f7 i5 j8 k8)
f6 i6 j6 k6 = let j9 = k6
                  k9 = k6 + 2
              in (f7 i6 j9 k9)
f7 i7 j7 k7 = (f2 i7 j7 k7)
```

Entretanto, o repasse de argumentos nem sempre é necessário visto que muitas funções se quer utilizam estes argumentos e apenas os repassam para o próximo bloco de código. Para não haver um excesso de argumentos desnecessários é possível utilizar o conceito de funções aninhadas. Seguindo o mesmo caminho, a figura 9 também possui um excesso de funções φ. Colocando tais funções apenas onde necessário, chegamos a figura 10, que corresponde perfeitamente ao código utilizando funções aninhadas a seguir.

```
f = let i1 = 1
        j1 = 1
        k1 = 0
    in let f2 (j2, k2) = if k2 < 100
                         then let f7 (j4, k4) = f2 (j4, k4)
                              in if j2 < 20
                                 then let j3 = i1
                                          k3 = k2 + 1
                                      in f7 (j3, k3)
                                 else let j5 = k2
                                          k5 = k2 + 1
                                      in f2 (j5, k5)
                         else return j2
       in f2 (j1, k1)
```

Dessa forma, é possível notar uma relação direta entre as funções φ e as chamadas de função no código. Apenas os blocos que utilizam funções φ possuem uma chamada de função correspondente no trecho de código.

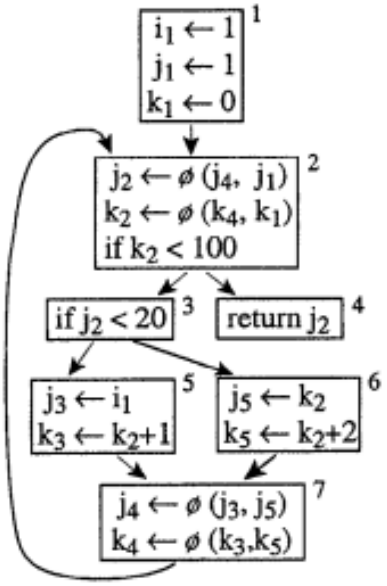

Figura 10 – Modelagem de SSA com o mínimo de funções φ
Fonte: (APPEL, 1998)

2.3 EFEITOS

Mitchell (2002) define um efeito colateral (do inglês, *side effect*) como mudanças visíveis no estado da máquina como resultado da valoração de uma expressão. Dessa forma, duas ocorrências de uma mesma expressão podem ter resultados diferentes, não permitindo transparência referencial. Nesse contexto, ao retornar um valor não observável, consequentemente, um beta-redex $(\lambda x.a)b$ é observacionalmente distinto de um beta-reduto $a[x := b]$ para algum efeito em b. A grande necessidade de IRs parte do suposto de que o λ-cálculo não trabalha bem com efeitos colaterais, podendo duplicar estes efeitos. Partindo deste pressuposto, Moggi (1988) propõe o λ_c isomórfico a ANF e capaz de lidar bem com efeitos.]

No campo de linguagens funcionais, existe uma divisão entre linguagens puras ou impuras. Linguagens puras utilizam λ-cálculo puro, enquanto as impuras estendem o λ-cálculo com uma série de possíveis efeitos. Porém, mesmo linguagens ditas "puras", podem, atualmente, simular e tratar estes efeitos por meio do uso de mônadas. Atualmente é possível, também, separar código puro de código com efeito colateral a partir do uso de um sistema de tipos e efeitos.

2.3.1 Mônadas

A utilização de mônadas para linguagens funcionais foi apresentada por Wadler (1995), o qual incorporou o conceito proveniente do estudo de teoria das categorias. Proposta como uma forma de simular efeitos colaterais em linguagens puras, a mônada de Wadler fundamentava-se na tríade (M, *unit*, $*$). Nesta tríade, M era o construtor de tipo da classe, *unit* o encapsulamento de um dado comum em uma mônada e $*$ o mapeamento de uma função em uma mônada. As assinaturas de tipo para os dois últimos conceitos é apresentada a seguir.

$$unit :: a \rightarrow M\ a$$
$$<*> :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow Mb$$

Utilizando mônadas é possível simular diversos efeitos como exceções, entrada e saída, não-determinismo, entre outros. Além disso, para ser válida, é necessário que as operações de uma mônada criada siga três regras: *left unit, right unit* e Associatividade.

- *Left unit:* Computar o valor de a, ligá-lo a b e depois computar n é equivalente a n com todas as ocorrências de b substituídas por a.

$$unit\ a *\lambda b.\ n = n[a/b]$$

- *Right unit:* Computar m, ligá-lo a a e depois retornar a é equivalente a m.

$$m *\lambda a.\ unit\ a = m$$

- *Associatividade:* a ordem dos parenteses neste tipo de computação é irrelevante.

$$m * (\ \lambda a.\ n *\lambda b.\ o) = (m *\lambda a.\ n) *\lambda b.\ o$$

Wadler também apresenta formas de utilizar mônadas para trabalhar com *Arrays* e implementar *Parsers*.

2.3.2 Sistema de Tipos e Efeitos

Outra maneira de lidar com efeitos, introduzida por Leijen (2014), é o sistema de tipos e efeitos. Um sistema de tipos descreve um conjunto de regras de inferência para as construções de uma linguagem. Tais construções são formalizadas por meio de sentenças, e.g., $\Gamma \vdash M : A$, de forma que o programa M possui tipo A no contexto Γ (CARDELLI, 1996). A forma geral de uma sentença é:

$$\frac{\Gamma_1 \vdash e_1 : \sigma_1 \qquad ... \qquad \Gamma_n \vdash e_n : \sigma_n}{\Gamma \vdash e : \sigma}$$

onde a parte superior da sentença é a premissa e a parte inferior é a conclusão. Tomando a formalização de sistemas de tipo como base, Leijen (2014) estende o modelo para apresentar efeitos. Uma sentença, neste modelo, possui a forma $\Gamma \vdash e : \sigma|\varepsilon$, significando que no contexto Γ, e possui tipo σ com efeito ε. Uma sentença geral, pode ser determinada da seguinte forma:

$$\frac{\Gamma_1 \vdash e_1 : \sigma_1|\varepsilon_1 \qquad ... \qquad \Gamma_n \vdash e_n : \sigma_n|\varepsilon_n}{\Gamma \vdash e : \sigma|\varepsilon}$$

O sistema de efeitos de Leijen utiliza, também, o conceito de efeitos polimórficos. Em alguns casos, o efeito de uma função é determinado a partir dos efeitos de algum de seus argumentos. Um exemplo disso é a função *map*, que realiza o mapeamento de uma função sobre cada elemento de uma lista. O tipo de *map* é definido como:

$$map : \forall \alpha \beta \mu . (list\langle \alpha \rangle, \alpha \to \mu \beta) \to \mu list\langle \beta \rangle$$

onde *map* recebe uma lista de elementos com tipo α e uma função que recebe um argumento de tipo α e retorna um tipo β com efeito μ. Assim, *map* absorve o efeito da função passada como argumento e retorna uma lista de tipo β com efeito μ. Outra característica proposta no trabalho de Leijen é o *row-polymorphism* utilizando duplicação de rótulos, dessa forma $\langle exn, exn \rangle \neq \langle exn \rangle$ e a combinação de dois efeitos básicos *exn* e *div* gera um *effect-row* $\langle exn, div \rangle$.

O sistema apresentado foi implementado na linguagem de programação Koka, onde os efeitos básicos são: *total, exn, div, ndet, alloc*$\langle h \rangle$*, read*$\langle h \rangle$*, write*$\langle h \rangle$*, io*. Sendo *total* a não ocorrência de efeitos colaterais, *exn* a possibilidade de exceção, *div* a possibilidade de entrar em loop, *ndet* para não-determinismo, *alloc*$\langle h \rangle$*, read*$\langle h \rangle$ *e write*$\langle h \rangle$ para representar efeitos de heap e *io* para efeitos de *input* e *output*. Koka também utiliza, internamente, um tradutor direto de tipo que traduz um programa com efeitos para um programa monádico equivalente.

2.4 CONVERSÃO DE SISTEMA DE EFEITOS PARA MÔNADAS

A correspondência entre sistema de efeitos e mônadas foi observada por Wadler e Thiemann (2003), através da proposta de uma tradução do sistema de efeitos de (TALPIN; JOUVELOT, 1992) para mônadas. Apesar da tradução para um sistema de efeitos específico, fica implícito no trabalho que qualquer sistema de efeitos pode ser adaptado para mônadas. Seguindo esta ideia, Vazou e Leijen (2016) formalizam a tradução do sistema de efeitos de Leijen para mônadas e vice-versa. Como grande diferencial, é apresentada a tradução de efeitos definidos pelo usuário para mônadas, assim como a tradução de *row-effects* (ver sec. 2.3.2) para mônadas.

A figura 11 apresenta a conversão de uma função com o efeito *amb* para código monádico equivalente. Para fazer essa passagem, o tradutor insere um *bind* e passa a continuação no ponto sempre que um valor monádico é retornado. Quando um valor puro é retornado em um contexto monádico, a função *unit* é utilizada, um *lifting* para transformar o valor puro em uma mônada. Além disso, toda a tradução realizada baseia-se na inferência de tipos feita pelo compilador anteriormente. Dessa forma é possível determinar onde posicionar o *bind* e o *unit* e verificar se eles seguem ou não as leis monádicas de identidade e associatividade.

Para lidar com polimorfismo, cada função que utiliza efeitos polimórficos recebe um dicionário como argumento extra. O dicionário vai encontrar e retornar as funções de *unit* e *bind*

Figura 11 – Conversão de código com efeito para código monádico
Fonte: (VAZOU; LEIJEN, 2016)

correspondentes ao efeito gerado. Quando existe a necessidade de traduzir uma função com mais de um efeito (*row-effects*), uma terceira mônada é criada representando a junção destes efeitos e morfismos de *lifting* de cada mônada de efeito para a mônada de junção também são criados.

3 PROPOSTA

REFERÊNCIAS

AHO, Alfred V et al. **Compiladores: Princípios, técnicas e ferramentas**. 2th. ed. São Paulo, SP, Brasil: Pearson Education, 2008. Citado 2 vezes nas páginas 12 e 13.

ALLEN, Frances E. Control flow analysis. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 5, n. 7, p. 1–19, 1970. Citado 2 vezes nas páginas 10 e 14.

APPEL, Andrew W. Ssa is functional programming. **Acm Sigplan Notices**, ACM New York, NY, USA, v. 33, n. 4, p. 17–20, 1998. Citado 5 vezes nas páginas 10, 12, 19, 20 e 22.

APPEL, Andrew W. **Compiling with continuations**. [S.l.]: Cambridge university press, 2007. Citado na página 19.

APPEL, Andrew W; MACQUEEN, David B. A standard ml compiler. In: SPRINGER. **Conference on Functional Programming Languages and Computer Architecture**. [S.l.], 1987. p. 301–324. Citado na página 17.

CARDELLI, Luca. Type systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 1, p. 263–264, 1996. Citado na página 23.

CHAKRAVARTY, Manuel MT; KELLER, Gabriele; ZADARNOWSKI, Patryk. A functional perspective on ssa optimisation algorithms. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 82, n. 2, p. 347–361, 2004. Citado na página 19.

CHURCH, Alonzo. A set of postulates for the foundation of logic. **Annals of mathematics**, JSTOR, p. 346–366, 1932. Citado 2 vezes nas páginas 10 e 17.

COOPER, Keith D; TORCZON, Linda. **Contruindo Compiladores**. 2th. ed. Rio de Janeiro, RJ, Brasil: Elsevier, 2014. Citado 2 vezes nas páginas 12 e 13.

CYTRON, Ron et al. An efficient method of computing static single assignment form. In: **Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. New York, NY, USA: Association for Computing Machinery, 1989. p. 25–35. Citado 2 vezes nas páginas 10 e 16.

FLANAGAN, Cormac et al. The essence of compiling with continuations. In: **Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation**. New York, NY, USA: Association for Computing Machinery, 1993. p. 237–247. Citado 2 vezes nas páginas 11 e 19.

HUGHES, John. Why functional programming matters. **The computer journal**, Oxford University Press, v. 32, n. 2, p. 98–107, 1989. Citado na página 10.

KELSEY, Richard A. A correspondence between continuation passing style and static single assignment form. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 30, n. 3, p. 13–22, 1995. Citado na página 19.

LEIJEN, Daan. Koka: Programming with row polymorphic effect types. **arXiv preprint arXiv:1406.2061**, 2014. Citado 3 vezes nas páginas 10, 11 e 23.

MITCHELL, John C. **Concepts in Programming Languages**. [S.l.]: Cambridge University Press, 2002. Citado na página 22.

MOGGI, Eugenio. Computational lambda-calculus and monads. Citeseer, 1988. Citado 2 vezes nas páginas 19 e 22.

RIGON, Leonardo Filipe; TORRENS, Paulo; VASCONCELLOS, Cristiano. Inferring types and effects via static single assignment. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. New York, NY, USA: Association for Computing Machinery, 2020. p. 1314–1321. Citado na página 11.

SABRY, Amr; FELLEISEN, Matthias. Reasoning about programs in continuation-passing style. **SIGPLAN Lisp Pointers**, Association for Computing Machinery, New York, NY, USA, V, n. 1, p. 288–298, jan 1992. ISSN 1045-3563. Disponível em: <https://doi.org/10.1145/141478.141563>. Citado 2 vezes nas páginas 18 e 19.

STEELE, Guy L. **Rabbit: A compiler for Scheme**. [S.l.]: Massachusetts Institute of Technology, 1978. Citado na página 19.

TALPIN, Jean-Pierre; JOUVELOT, Pierre. Polymorphic type, region and effect inference. **Journal of functional programming**, Cambridge University Press, v. 2, n. 3, p. 245–271, 1992. Citado na página 24.

TORRENS, Paulo; VASCONCELLOS, Cristiano; GONÇALVES, Junia. A hybrid intermediate language between ssa and cps. In: **Proceedings of the 21st Brazilian Symposium on Programming Languages**. New York, NY, USA: Association for Computing Machinery, 2017. p. 1–3. Citado na página 10.

VAZOU, Niki; LEIJEN, Daan. From monads to effects and back. In: SPRINGER. **Practical Aspects of Declarative Languages: 18th International Symposium**. St. Petersburg, FL, USA, 2016. p. 169–186. Citado 4 vezes nas páginas 11, 12, 24 e 25.

WADLER, Philip. Monads for functional programming. In: SPRINGER. **Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques**. Båstad, Sweden, 1995. p. 24–52. Citado 2 vezes nas páginas 10 e 22.

WADLER, Philip; THIEMANN, Peter. The marriage of effects and monads. **ACM Transactions on Computational Logic (TOCL)**, ACM New York, NY, USA, v. 4, n. 1, p. 1–32, 2003. Citado na página 24.