Uma das principais funções de um sistema de tipos é auxiliar a construção de programas com menor incidência de erros. Por exemplo, o sistema de tipos utilizado na linguagem Haskell garante a ausência de erros de tipo durante a execução de um programa, isso, somado à ausência de efeitos colaterais, possibilita a criação de programas com maior probabilidade de estarem corretos. Entretanto, a manipulação de dados que implicitamente causam efeitos colaterais não é muito intuitiva para o programador habituado a trabalhar com linguagens imperativas. A linguagem de programação funcional *Koka* define uma sintaxe mais próxima a linguagens imperativas: além de possibilitar a verificação dos tipos das funções, a linguagem também permite a inferência de efeitos colaterais, garantindo uma maior segurança aos programas escritos na mesma. SSA (*Static Single Assigment*) é uma representação intermediária de código, geralmente utilizada para auxiliar o processo de otimização. Essa representação é facilmente traduzida para uma representação funcional. O objetivo deste projeto é a definição do núcleo de uma linguagem com sintaxe imperativa e a construção de um compilador que garanta as propriedades de tipos e efeitos dos códigos implementados. Para isso, propõe-se traduzir o código fonte dessa linguagem para a representação funcional de SSA, através de grafos de fluxo de controle. Será definido, também, um algoritmo capaz de inferir tipos e efeitos colaterais dessa representação.

Orientador: Prof. Dr. Cristiano Damiani Vasconcellos

JOINVILLE, 2019

ANO 2019 LEONARDO FILIPE RIGON | INFERÊNCIA DE TIPOS E EFEITOS POR MEIO DE GRAFOS DE FLUXO DE CONTROLE

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA – PPGCA

DISSERTAÇÃO DE MESTRADO

INFERÊNCIA DE TIPOS E EFEITOS POR MEIO DE GRAFOS DE FLUXO DE CONTROLE

LEONARDO FILIPE RIGON

JOINVILLE, 2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM COMPUTAÇÃO APLICADA

LEONARDO FILIPE RIGON

INFERÊNCIA DE TIPOS E EFEITOS POR MEIO DE GRAFOS DE FLUXO DE CONTROLE

JOINVILLE

2019

LEONARDO FILIPE RIGON

INFERÊNCIA DE TIPOS E EFEITOS POR MEIO DE GRAFOS DE FLUXO DE CONTROLE

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Computação Aplicada.

Orientador: Dr. Cristiano Damiani Vasconcellos

JOINVILLE

2019

Ficha catalográfica elaborada pelo programa de geração automática da

Biblioteca Setorial do CCT/UDESC,

com os dados fornecidos pelo(a) autor(a)

Rigon, Leonardo Filipe
Inferência de tipos e efeitos por meio de grafos de fluxo de controle / Leonardo Filipe Rigon. -- 2019.
94 p.

Orientador: Cristiano Damiani Vasconcellos
Dissertação (mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2019.

1. Sistemas de tipos. 2. Efeitos colaterais. 3. Grafos de fluxo de controle. 4. Linguagens funcionais. 5. Efeitos Algébricos. I. Damiani Vasconcellos, Cristiano. II. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. III. Título.

Inferência de Tipos e Efeitos por meio de Grafos de Fluxo de Controle

por

Leonardo Filipe Rigon

Esta dissertação foi julgada adequada para obtenção do título de

Mestre em Computação Aplicada

Área de concentração em "Ciência da Computação",
e aprovada em sua forma final pelo

CURSO DE MESTRADO ACADÊMICO EM COMPUTAÇÃO APLICADA
DO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA
UNIVERSIDADE DO ESTADO DE SANTA CATARINA.

Banca Examinadora:

Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC (Orientador/Presidente)

Profa. Dra. Karina Girardi Roggia
CCT/UDESC

Dr. Andre Murbach Maidl
Elastic

Joinville,SC, 20 de setembro de 2019.

Dedico essa pesquisa aos meus tios, Maria Luíza e Alvino, à minha mãe, meus amigos, meus professores e demais colegas que estiveram ao meu lado durante a vida acadêmica.

AGRADECIMENTOS

Gostaria de deixar meus agradecimentos aos meus tios, Maria Luíza e Alvino, por terem me recebido em sua casa durante a pós-graduação. À minha mãe por ter me ajudado quando precisei. Ao meu amigo Paulo com quem aprendi muito e que me ajudou em diversos momentos. Ao meu orientador, professor Cristiano, por ter aceitado me orientar, mesmo com minha falta de embasamento teórico em computação. À minha amiga Erica por me ajudar nos piores momentos durante a pesquisa. A todos os amigos e colegas com quem tive a oportunidade de conviver no período do mestrado. Ao grupo de fundamentos da computação (Função), pelos momentos de aprendizado e alegria. À NG informática por ter me proporcionado recursos para poder concluir a pesquisa, especialmente à minha equipe de trabalho, o PeD. Além disso, agradeço ao apoio das agências CAPES e FAPESC por terem financiado essa pesquisa.

It's the job that's never started as takes longest to finish.

J. J. R. Tolkien

RESUMO

Uma das principais funções de um sistema de tipos é auxiliar a construção de programas com menor incidência de erros. Por exemplo, o sistema de tipos utilizado na linguagem Haskell garante a ausência de erros de tipo durante a execução de um programa, isso, somado à ausência de efeitos colaterais (alterações de valores gravados na memória do computador), possibilita a criação de programas com maior probabilidade de estarem corretos. Entretanto, a manipulação de dados que implicitamente causam efeitos colaterais não é muito intuitiva para o programador habituado a trabalhar com linguagens imperativas. A linguagem de programação funcional *Koka* define uma sintaxe mais próxima a linguagens imperativas: além de possibilitar a verificação dos tipos das funções, a linguagem também permite a inferência de efeitos colaterais, garantindo uma maior segurança aos programas escritos na mesma. SSA (*Static Single Assigment*) é uma representação intermediária de código, geralmente utilizada para auxiliar o processo de otimização. Essa representação é facilmente traduzida para uma representação funcional. O objetivo deste projeto é a definição do núcleo de uma linguagem com sintaxe imperativa e a construção de um compilador que garanta as propriedades de tipos e efeitos dos códigos implementados. Para isso, propõe-se traduzir o código fonte dessa linguagem para a representação funcional de SSA, através de grafos de fluxo de controle. Será definido, também, um algoritmo capaz de inferir tipos e efeitos colaterais dessa representação.

Palavras-chaves: Sistemas de Tipos, Efeitos Colaterais, Grafos de Fluxo de Controle, Linguagens Funcionais.

ABSTRACT

One of the main purposes of a type system is to reduce the error occurrence in software. Haskell uses a type system that ensures the absence of type errors in the program at running time. Thus, the non-existence of side effects (mutation of values in the computer memory) allows the creation of more correct programs. Yet, data manipulation that implicitly causes side effects is not intuitive for programmers who are used to work with imperative languages. The functional programming language Koka defines a very similar syntax to imperative languages. Koka allows for type checking in functions along with side effects inference, resulting in safer safe programs. The SSA (Single Static Assignment) is a intermediate code representation used in the optimization process. This representation is easily converted syntactically to a functional representation. The main purpose of this project is to define of a core programming language with imperative syntax and the construction of a compiler that ensures the type and effect properties of implemented code. To achieve the goal, it is proposed to translate the source code of this language to a SSA functional representation using control flow graphs. An algorithm will also be defined to infer types and side effects on this representation.

Key-words: Type Systems, Collateral Effects, Flow-Control Graphs, Functional Languages.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

CFG Grafo de Fluxo de Controle (*Control Flow Graph*)

IO Entrada e Saída (*Input Output*)

IR Representação Intermediária (*Intermediate Representation*)

SSA Atribuição Estática Única (*Single Static Assignment*)

ST Transformador de Estados (*State Transformer*)

SUMÁRIO

1 INTRODUÇÃO

Um dos propósitos de um sistema de tipos é auxiliar no desenvolvimento de códigos com menor incidência de erros. Cardelli (1996) explica que a utilização de um sistema de tipos visa diminuir a ocorrência de erros em tempo de execução de programas. Portanto, linguagens de programação que fazem uso destes mecanismos podem tornar o processo de criação de programas mais seguro e eficaz. A linguagem Haskell utiliza uma extensão do sistema de tipos de Damas e Milner (1982), que permite a detecção de erros em tempo de compilação. Isso garante que erros relacionados a tipos não ocorrerão em tempo de execução. A não existência de efeitos colaterais em linguagens funcionais puras, é, também, um fator que auxilia na construção de programas com comportamentos mais facilmente verificáveis. Entende-se como efeito colateral a alteração de um estado que irá afetar computações futuras como, por exemplo, alterações de valores de memória, leitura e escrita em arquivos.

Aplicações gerais utilizam funções ou métodos que podem causar efeitos colaterais. Embora não permita, de fato, tais efeitos colaterais, Haskell permite a simulação de efeitos utilizando um conceito matemático inspirado na teoria das categorias chamado, *mônada*. Para seu uso não é necessário conhecimento prévio da teoria das categorias, mas a utilização de *mônadas* pode causar dificuldades ao programador habituado a linguagens imperativas. Isso deve-se ao fato de que programas em linguagens funcionais são obtidos por intermédio da combinação de expressões.

Leijen (2014) propõe uma linguagem de programação funcional pura, denominada Koka, que fornece um método alternativo para lidar com efeitos colaterais. O sistema de tipos de Koka possibilita a inferência tanto de tipos quanto a de efeitos colaterais em uma função. Para executar esta inferência, a linguagem propõe alterações ao sistema de tipos de Damas e Milner (1982), agregando a este a capacidade de efetuar a tipagem e combinação de efeitos colaterais. A Figura 1 exibe o código de um programa escrito em Koka. Este programa efetua o cálculo do enésimo número da sequência de Fibonacci, utilizando elementos que causam efeitos colaterais (impressão de dados e emissão de erros).

Além das características descritas, pode-se notar na Figura 1 que Koka proporciona uma sintaxe mais aproximada de linguagens do paradigma imperativo. Atributo que facilita a transição de um programador para este modelo de programação. Diferentemente de muitas abordagens encontradas na literatura, Koka não trata efeitos apenas como rótulos sintáticos; o sistema de efeitos permite inferir o tipo de efeito de uma função. Seu funcionamento permite, além da tipagem, a combinação de efei-

Figura 1 – Exemplo - Programa em Koka utilizando efeitos colaterais

```
1  fun fibonacci(n : int) {
2      if (n < 0) return (error("negative number"))
3      print("result:")
4      return (fib(n))
5  }
6
7  fun fib(n : int) {
8      fibs(n, 0, 1)
9  }
10
11 fun fibs(n : int, x : int, y : int) {
12   if (n==0) return x
13   fibs(n - 1, y, x+y)
14 }
```

Fonte: O Autor.

tos colaterais. Essas garantias possibilitam a criação de códigos com menor incidência de erros e, consequentemente, mais seguros. Isso deve-se ao fato de que o sistema de tipos de Koka indentifica para o programador as funções que alteram valores em memória.

A verificação de tipos e efeitos ocorre durante o processo de compilação de um código, podendo ser enxergado como um mecanismo de geração de garantias. Em geral, compiladores para linguagens funcionais são construídos de forma bastante diferente de compiladores para linguagens imperativas. Além de sistemas de tipos mais consistentes que fornecem mais garantias, as representações intermediárias de código geradas pelos compiladores e as técnicas de otimização adotadas são bastante distintas das utilizadas para linguagens imperativas.

O funcionamento de um compilador é dividido em etapas, e, entre cada uma, gera-se uma representação intermediária. Essas representações são necessárias para o andamento do processo de compilação, recebendo otimizações quando possível. Existem diversos tipos de representações, podendo ser citadas como exemplos: as árvores sintáticas, código de três endereços e os grafos de fluxo de controle, abreviados como CFG (*Control Flow Graph*).

Grafos de fluxo de controle são representações gráficas que facilitam a visualização dos caminhos a serem percorridos dentro de um programa. Os fluxos representam qual bloco básico deve ser alcançado a partir de um determinado bloco. Um CFG, normalmente, é acompanhado de outro modelo de representação intermediária, como o código de três endereços e a forma de atribuição estática única.

A forma de atribuição estática única, referida como SSA (*Static Single As-*

sigment), é uma representação intermediária de código que pode ser associada ao grafo de fluxo de controle. Como sugere sua denominação a forma SSA impõe que para cada ocorrência de variável em um código, seja atribuído um novo nome. Esse denominador não pode ser alterado no decorrer da compilação. Em programas onde ocorrem desvios condicionais é comum a utilização de GFC agregado à forma SSA. Nesses programas existe mais de um fluxo a ser seguido a partir de um bloco. A utilização de SSA permite a inserção de mecanismos para tratar esses casos de modo a simplificar a representação.

A combinação entre CFG e SSA é amplamente utilizada em processos de otimizações, podendo ser encontrada em compiladores, como no GCC (STALLMAN, 2005). Essa forma de Representação Intermediária permite diversas otimizações, além de permitir a conversão em uma representação funcional (APPEL, 1998) e (TORRENS; VASCONCELLOS; GONÇALVES, 2017). A representação funcional gerada pode receber otimizações e garantias comuns de linguagens de programação funcionais.

A ideia deste trabalho é investigar a possibilidade de aplicar o algoritmo de tipos e efeitos, concebido para a linguagem de programação Koka, a esta representação funcional. Verificando se um programa escrito e compilado nestas condições teria capacidade de receber, além das garantias do sistema de tipos e efeitos, otimizações na transformação de linguagem fonte para a representação SSA. Otimizações, estas, que são comuns em linguagens de programação imperativa.

Além do que foi proposto por Leijen (2014), outros trabalhos que sistemas de efeitos. Wadler e Thiemann (2003) mostram que sistemas de efeitos e mônadas comportam-se de forma análoga, propondo a combinação entre estes sistemas. Na mesma linha, Orchard e Petricek (2015) sugerem a incorporação de sistemas de efeitos por meio de mônadas de efeito paramétrico ao compilador de Haskell, o GHC (JONES, 2003).

Recententemente, os trabalhos de Ahman (2017), Hillerström e Lindley (2016), e Kotelnikov (2014) também abordam a disciplina de sistemas de efeitos aplicados em ambientes de linguagens funcionais. Da mesma forma, podem ser encontradas pesquisas para a aplicação de mecanismos para controle de efeitos colaterais em computação concorrente, como no trabalho de Leijen (2017) e de Long e Rajan (2016).

Lindley, McBride e McLaughlin (2017) definem uma linguagem de programação funcional, chamada Frank, e um sistema de efeitos bi-direcional com suporte a efeitos polimórficos. O sistema de efeitos utilizado em Frank, diferencia-se de outras propostas por permitir a utilização de *multihandlers*, extendendo o sistema de tipos de Damas e Milner (1982).

A abordagem apresentada no presente trabalho diferencia-se das demais, pois define o núcleo de uma linguagem de programação com sintaxe imperativa, aplicando a esta linguagem um compilador que possibilita gerar garantias de propriedades relacionadas a tipos e efeitos colaterais. O objetivo principal deste trabalho é investigar em quais situações essa inferência é possível considerando uma sintaxe inspirada em linguagens imperativas, incluindo comandos de repetição e desvio. Esse compilador converte códigos para a representação SSA, utilizando grafos de fluxo de controle, preservando informações sobre o tipo.

Após traduzidas, as representações são convertidas em representações funcionais equivalentes. Um algoritmo de tipos e efeitos foi implementado com base no proposto para linguagem Koka que é aplicado ao código funcional gerado. As contribuições do trabalho são:

- A definição do núcleo de uma linguagem de programação com sintaxe imperativa. Este núcleo contém a estrutura básica de uma linguagem de programação de propósito geral, semelhante à linguagem C.
- A definição e implementação de um algoritmo para a conversão desta linguagem fonte para uma representação intermediária na forma CFG/SSA. Essa transformação pode manter as propriedades do programa fonte e aplicar otimizações comuns em linguagens imperativas.
- A tradução da forma CFG/SSA para uma representação funcional equivalente, utilizando a proposta contida no trabalho de Appel (1998). Essa tradução gera códigos com uma estrutura mais próxima ao cálculo lambda.
- A implementação de um algoritmo para inferência de tipos e efeitos, semelhante ao utilizado no trabalho de Leijen (2014). Esse sistema é agregado à transformação de CFG/SSA para representação funcional em um compilador.
- A criação de uma nova linguagem e de um compilador que facilitem o trabalho do programador, proporcionando a criação de códigos mais seguros e com menor carga de trabalho no desenvolvimento de programas.

Muitos conceitos são necessários para o entendimento e desenvolvimento da proposta deste trabalho. Estes conceitos estão dispostos no Capítulo 2, onde encontram-se informações referentes aos métodos pesquisados e que foram empregados na construção do compilador proposto. Os detalhes da implementação, do desenvolvimento e os resultados preliminares são exibidos no Capítulo 3. No Capítulo 4 são apresentadas as discussões acerca dos resultados e a conclusão do projeto de pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo abordará aspectos teóricos com relação aos conceitos e referenciais bibliográficos utilizados na construção dos algoritmos propostos neste trabalho. Serão abordados temas que tem relação direta com o trabalho, como o sistema de tipos de Damas e Milner (1982) na Seção 2.1. A linguagem funcional Haskell, escolhida para a implementação do protótipo, é debatida na Seção 2.2. O sistema de efeitos do compilador desse trabalho é inspirado na linguagem programação Koka, apresentada nos trabalhos de Leijen (2005), Leijen (2016) e Leijen (2017), por essa razão um resumo dessas publicações pode ser encontrado na Seção 2.3. Por último são abordadas as representações intermediárias de código expostas na Seção 2.4.

2.1 SISTEMA DE TIPOS DE DAMAS MILNER

Conforme Cardelli (1996), o foco essencial de um sistema de tipos consiste na prevenção da ocorrência de erros na execução de um programa. Além de auxiliar o programador a escrever códigos de forma segura, com menor incidência de erros. A utilização de sistema de tipos pode, através da assinatura de tipos, fornecer informações que servem de documentação a respeito da implementação.

Um sistema de tipos estabelece uma relação entre expressões e tipos, sendo essa relação usualmente descrita por um conjunto de regras de inferência. O principal objetivo de um sistema de tipos é determinar, geralmente em tempo de compilação, se um programa é bem comportado, garantindo a ausência de determinados erros. Um sistema capaz de fornecer a garantia de ausência de erros de tipo é dito consistente (*sound*). Inicialmente proposto para a linguagem de programação funcional ML, o sistema de tipos de Damas e Milner (1982) serviu de inspiração para grande maioria das linguagens funcionais atuais. O sistema é apresentado para o núcleo da linguagem ML definido pela gramática a seguir:

$$e ::= x \mid e\ e' \mid \lambda x.e \mid let\ x = e\ in\ e'$$

Nesta sintaxe, os termos e e x representam, respectivamente, uma expressão e uma variável. A produção $let\ x = e\ in\ e'$ introduz a definição de uma função x possivelmente polimórfica, que pode ser usada na expressão e'. Podem-se dividir os tipos em dois grupos: os tipos monomórficos, representados por τ e os tipos polimórficos, que apresentam variáveis quantificadas universalmente, representados por σ. Estes tipos têm a seguinte sintaxe:

$$\tau ::= \alpha \mid \tau \rightarrow \tau'$$
$$\sigma ::= \tau \mid \quad \forall \alpha.\sigma$$

O tipo monomórfico τ pode ser uma variável de tipo α, ou um tipo função, que representa o mapeamento de um dado de tipo τ em τ'. O tipo σ pode ser um tipo τ, ou a quantificação de variáveis de tipo α que ocorrem em σ. Damas e Milner (1982) utilizam o conceito de bem tipado (*well-typed*) para denotar que uma determinada expressão está escrita de forma correta e que seu comportamento é seguro. Para tal, se utiliza um conjunto de inferências, dispostas na Figura 2.

Figura 2 – Definição - Sistema de Tipos de Damas-Milner

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \textbf{ (VAR)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma < \sigma'}{\Gamma \vdash e' : \sigma'} \textbf{ (INST)}$$

$$\frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \tau' \rightarrow \tau} \textbf{ (ABS)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin ftv(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma} \textbf{ (GEN)}$$

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \textbf{ (APP)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma_x \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash (let\ x\ =\ e\ in\ e') : \tau} \textbf{ (LET)}$$

Fonte: Damas e Milner (1982)

A inferência de tipos de uma expressão é feita a partir de um contexto de tipos. Este contexto abriga apenas uma suposição de tipo para cada variável livre da expressão a ser avaliada. Também na Figura 2, a forma $\Gamma \vdash e : \sigma$, significa que uma expressão e apresenta o tipo σ em um contexto Γ. Já o termo Γ_x indica um contexto Γ, em que não há suposições do tipo de x. Cada uma das regras pode ser descrita da seguinte forma:

- **VAR**: Se x for do tipo σ no contexto Γ, então, a partir de Γ, pode-se inferir que x é do tipo σ.
- **ABS**: Representa uma abstração lambda, em um contexto em que x possui tipo τ' pode-se inferir que o tipo de e será τ e, então, deduz-se que o tipo desta abstração e, levando em consideração a variável x, $\lambda x.e$, é $\tau' \rightarrow \tau$.
- **APP**: Aplicação de expressões, supondo que e apresenta o tipo $\tau \rightarrow \tau'$ e que e' é do tipo τ, pode-se inferir que a aplicação de e em e' terá o tipo τ'.
- **INST**: Instanciação de tipos de variáveis, em que, caso e apresente o tipo σ e σ' for uma instanciação de σ, infere-se que e será do tipo σ'. A forma $\sigma < \sigma'$ mostra que o tipo σ' é uma instância do tipo σ (σ' é mais especifico que σ).

- **GEN**: Denota a generalização, se uma variável do tipo α não está livre ($\notin$ *ftv*) no contexto Γ atual, e sendo possível inferir que e possui o tipo σ independente de α, e será do tipo $\forall\alpha.\sigma$.
- **LET**: Na condição de e ter o tipo σ, presumindo que x carrega o tipo σ, e inferindo que e' apresenta o tipo τ, então, pressupõe-se que o tipo da forma $let\ x = e\ in\ e'$ será τ.

Conforme mostrado no trabalho de Damas e Milner (1982), por meio desse conjunto de regras é possível inferir o tipo mais geral de uma expressão. O Algoritmo W, apresentado na Figura 3, infere para uma expressão um tipo que representa todos esses tipos possíveis, chamado tipo principal da expressão.

Figura 3 – Definição - Algoritimo W

```
W(Γ, x) =
    If Γ (x) = ∀α1...α2.τ then (Id, [β/α]τ)
    else fail

W(Γ, e e') =
    let (S1, τ) = W(Γ, e)
    (S2, τ') = W(S1Γ, e')
    S = unify (S2τ', τ' → β) where β is fresh
    in (S ∘ S2 ∘ S1, Sβ)

W(Γ, λx.e) =
    let (S,τ) = W(Γx ∪ {x : β}, e)
    in (S,S(β → τ))

W(Γ, let x = e in e') =
    let (S1, τ) = W(Γ, e)
    (S2, τ') = W(S1(Γx ∪ {x : closure (S1Γ,τ)}), e')
    in (S1 ∘ S2, τ')}
```

Fonte: Damas e Milner (1982)

O Algoritmo W tem como resultado o tipo principal de uma expressão, ou seja, um tipo que representa todos os possíveis tipos que podem ser derivados para a expressão. O algoritmo recebe como entrada uma dupla (Γ, e), sendo Γ um contexto de tipos e e uma expressão. O retorno dessa computação será outra dupla onde estarão dispostos um tipo principal da expressão τ e uma substituição S. Caso ocorra algum problema durante a execução e, por consequência, não se obtenha o tipo principal da expressão, um erro deverá ser emitido. O algoritmo utiliza os conceitos de substituições e de unificações.

Uma substituição pode ser vista como um mapeamento de variáveis de tipo em tipos simples. Dois tipos τ e τ' são unificáveis se existir uma substituição S que os torne iguais ($S(\tau) = S(\tau')$). O unificador S_ν é o mais geral, se para todo unificador S existir umas substituição S' ($S' \circ S_\nu = S$).

A função $unify(\tau_1, \tau_2)$, usada na Figura 3, computa o unificador mais geral para os tipos. A chamada de função *closure*, denominada fechamento de tipo, corresponde à quantificação das variáveis de tipo, sendo definida como:

$$closure\ (\Gamma,\ \tau) = \forall \alpha_1 ... \alpha_n . \tau$$

Os termos $\alpha_1 ... \alpha_n$ correspondem a variáveis de tipo que ocorrem em τ, mas não em Γ. Como anteriormente citado, o sistema de tipos de Damas e Milner (1982) é amplamente utilizado em linguagens de programação funcionais.

2.2 LINGUAGENS DE PROGRAMAÇÃO FUNCIONAIS

O conceito de paradigma de programação refere-se ao modelo em que se é concebido e executado um programa de computador. Tucker (1986) define paradigma de programação como um padrão de resolução de problemas relacionado a um gênero de linguagens e programas. Atualmente, existem vários paradigmas de programação conhecidos, sendo os principais: o paradigma imperativo, o paradigma orientado a objetos, o paradigma funcional e o paradigma lógico. É possível que uma linguagem apresente características de mais de um destes paradigmas. A linguagem Python (ROSSUM et al., 2007), por exemplo, suporta características de vários paradigmas de programação, simultaneamente, como os paradigmas imperativo, orientado a objetos e funcional.

Esses paradigmas, conforme Tucker (1986), diferem-se uns dos outros em vários aspectos. Os conceitos, as abstrações e a sintaxe de cada uma desses estilos podem ser totalmente diferentes. Elementos como funções, variáveis e objetos são representados em cada um destes paradigmas de forma distinta. Além disso, o processo (atribuições, representações intermediárias e otimizações) de compilação, como um todo, é executado de forma diferente.

A Figura 4 representa dois pequenos programas, o código da esquerda está escrito na linguagem C (paradigma imperativo) e o da direita escrito em Haskell (paradigma funcional).

Os códigos contidos na Figura 4 retornam o mesmo valor como resultado, sendo equivalentes. Os dois programas efetuam a inversão de caracteres passados como parâmetros de entrada. Além das perceptíveis diferenças na sintaxe e na forma com a qual estão estruturados, outras diferenças podem ser destacadas. Segundo

Figura 4 – Comparativo - Código imperativo e código funcional

```c
void rev (char *arr) {
    size_t x = strlen(arr);
    for(int i = 0; i < (x/2); i++) {
        char aux   = arr[i];
        arr[i]     = arr[x-i-1];
        arr[x-i-1] = aux;
     }
     printf("%s \n", arr);
}

int main() {
    char *arr;
    arr = malloc(26);
    scanf("%25s", arr);
    reverse(arr);
    free(arr);
    return 0;
}
```

```haskell
main :: IO ()
main = do
         a <- getLine
         print (rev a)

rev :: [a] -> [a]
rev [] = []
rev (x:xs) = (rev xs)++[x]
```

Fonte: O Autor.

Allen e Moronuki (2017), diferentemente do paradigma imperativo, onde os programas têm como base instruções sequenciais, a programação funcional utiliza a definição e aplicação de funções como base para a construção de programas.

O modelo de programação imperativo descreve programas em forma de estados. Isso pode ser entendido como uma aplicação que retém informações em variáveis, as quais são despendidas na memória do computador. A partir dos valores destas posições na memória, em um determinado momento de execução, será definido o estado do programa (ALLEN; MORONUKI, 2017). Os valores contidos nesses estados podem ser alterados durante a execução de um programa.

A programação funcional pode ser vista como a ideia de que programas podem ser desenvolvidos com uma combinação entre expressões. Dentro dessas expressões estão contidas variáveis e funções. Estas funções dão-se da mesma forma que funções matemáticas, seu comportamento é dado pela relação entre as suas entradas e saídas. O resultado de uma computação é obtido exclusivamente de acordo com o que foi passado como entrada. O resultado de um programa não deve sofrer influência de agentes externos. Conforme Allen e Moronuki (2017), além de dependerem de expressões e serem baseadas em cálculo lambda, as linguagens funcionais não são todas igualmente puras. A pureza de uma linguagem de programação tem relação com a ausência de possíveis efeitos colaterais em seu funcionamento, sendo pura, caso não permita tais efeitos, ou impura, caso os permita.

Um efeito colateral, nesse contexto, consiste na alteração de um estado, como por exemplo: a mudança do valor de uma variável, a gravação em disco, ou até a ha-

bilitação de um botão na interface do usuário. Como exemplos de linguagens funcionais impuras, que permitem alterações de estados, pode-se citar Lisp (JR; COMMON, 1984) e ML (MILNER et al., 1997).

Linguagens funcionas puras garantem transparência referencial (*referential transparency*), baseando-se na concepção de que uma função é simplesmente um mapeamento dos argumentos de entrada em seus resultados. Em outras palavras, esta transparência quer dizer que uma função nunca retornará resultados diferentes diante dos mesmos parâmetros.

Por apresentar maior rigor matemático, onde os valores das variáveis não podem ser alterados, essas linguagens possibilitam a verificação de algumas propriedades dos programas. Miranda (TURNER, 1985) e Haskell (JONES, 2003) representam linguagens funcionais puras.

2.2.1 Haskell

Haskell é uma linguagem de programação puramente funcional de propósito geral, não estrita e fortemente tipada, muito influenciada por Miranda (HUDAK et al., 2007). O núcleo e o funcionamento da linguagem herdam muitos aspectos de Miranda, tendo algumas diferenças em relação à sua antecessora.

Miranda é baseada no cálculo lambda de Alonzo Church. Sem apresentar características de linguagens imperativas, Miranda tem seus códigos escritos em forma de conjuntos, contendo definições de funções e objetos de dados. O modelo de programação da linguagem utiliza equações e interações de recursão para a criação de aplicações (TURNER, 1985).

Hudak et al. (2007) expõe que Miranda vem sendo há algum tempo substituída por Haskell em diversos meios. Atualmente Haskell, além de ser muito utilizada pela comunidade científica, pode ser encontrada como parte de grandes aplicações, em diversas áreas. Isso deve-se ao fato de Miranda pertencer a uma compania, enquanto Haskell é continuamente desenvolvida pela comunidade científica e disponibilizada de forma gratuita para uso acadêmico e comercial. Outro porém é Miranda ser nativa apenas para Unix, tornando o seu processo de uso em Windows mais trabalhoso, se comparado a Haskell, que é nativa para ambas as famílias de sistemas operacionais.

Ainda segundo Hudak et al. (2007), existem algumas diferenças na composição de programas entre as duas linguagens. As mais notáveis são: a sintaxe da declaração de tipos de dados, construída de forma diferente; o acúmulo dos construtores de tipos e dados; e o uso de identificadores alfanuméricos para variáveis de tipo, em vez de asteriscos.

Listas são estruturas de dados muito utilizadas em linguagens de programa-

ção funcionais. Uma lista, em Haskell, é composta por colchetes que denotam o começo e o fim da estrutura, contendo, em seu interior, elementos de um determinado tipo, separados por vírgulas. Por exemplo:

```
shapes = ["triangle", "square", "rectangle", "circle"]
```

Neste caso, `shapes` é o nome dado para a lista com os elementos dispostos entre colchetes. A partir desta definição de lista a linguagem permite, por padrão, o uso de alguns operadores para a manipulação dessas estruturas. Como, por exemplo, concatenação de listas, inversão de valores e funções para a fragmentação da lista.

Tuplas também fazem parte da composição do núcleo da linguagem. Diferentemente das listas, que restringem apenas um tipo (todos os elementos devem ser do mesmo tipo) em suas composições, tuplas permitem valores de tipos diferentes em sua estrutura. Seu início e término são marcados por parênteses, tendo seus elementos divididos por vírgulas:

```
band = ("Ramones", 1977, False)
work = (1, 'a', True, "Wadler")
```

Para a manipulação dos dados das tuplas utiliza-se o conceito de casamento de padrões (*pattern matching*). Este e outros conceitos importantes para o funcionamento da linguagem serão abordados nas subseções seguintes.

2.2.1.1 Casamento de Padrões

O casamento de padrões (*pattern matching*) permite a definição de funções com várias alternativas de uso. Consiste em uma estrutura diferente da habitual (para programadores acostumados a linguagens imperativas) ao se desenvolver funções com diversas opções. De modo que, dependendo do argumento de entrada, será escolhida uma das funções disponíveis. Por exemplo, a função `take`, abaixo, recebe como parâmetros: um número qualquer `n` e uma lista, retornando os n primeiros valores desta lista.

```
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

A primeira opção de `take`, ocorre quando o valor para `n` é o de número `0`, retornando assim uma lista vazia, representada por `[]`. Já o segundo caso propõe que, independentemente do valor de `n`, seja passado como parâmetro uma lista vazia, o resultado dessa computação será também uma lista vazia.

A última alternativa trata da situação em que são passados, como parâmetros, um número `n` qualquer, diferente de `0`, e uma lista contendo quaisquer valores.

É importante observar, que Haskell utiliza a forma `(x:xs)` para denotar a composição estrutural de listas, separando a cabeça (primeiro elemento da lista) `x`, do corpo (restante da lista) `xs`, através do operador de dois pontos `:`. O retorno desta alternativa, separa `x`, compondo uma nova lista (através do operador `:`), onde `x` será acompanhado do retorno da função `take`, que terá como argumentos, o valor de `n`, decrescido de 1, (podendo ser `0` e cair no primeiro padrão) e o restante da lista (que pode ser vazia e cair no segundo caso). Se ainda houverem elementos na lista, a terceira alternativa será chamada. Nela se repetirá o processo, recursivamente, até que uma das duas primeiras condições seja satisfeita.

Para se manipular dados de tuplas, o casamento de padrões também é utilizado. Como exemplos clássicos, temos as funções `fst` e `snd`:

```
fst (a, b) = a
snd (a, b) = b
```

A função *fst* recebe, como parâmetro, uma tupla com os valores `a` e `b`, e retorna o valor de `a`. E a função `snd`, segue a mesma estrutura, porém retorna o valor de `b`. Segundo Turner (1985), o casamento de padrões é uma alternativa mais elegante no desenvolvimento de códigos, se comparada ao uso de guardas e múltiplas combinações da composição `if-then-else`.

2.2.1.2 Funções de Ordem Superior

Outra característica importante de Haskell é o fato de que funções são valores de primeira classe, ou seja, funções podem ser passadas como parâmetros para outras funções. Funções que recebem funções como argumento são chamadas de funções de ordem superior. Além disso, funções também podem ser retornadas como resultados. A aplicação de funções é associativa à esquerda, isto é, dada uma função "`foo a b` ", esta será analisada como "`(foo a) b`", onde o resultado da aplicação do termo `foo` em `a`, será uma função aplicada ao termo `b`.

Em outras palavras, toda expressão que tenha dois ou mais argumentos é uma função de ordem superior, isso permite aplicações parciais de funções. A função `foldr`, presente na biblioteca padrão de Haskell, é um exemplo de função de ordem superior:

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

A expressão `foldr` recebe como parâmetros: uma função `op`, um valor qualquer `k` e uma lista `(a:x)`. Aplica-se `op`, utilizando como parâmetros `k` e o último item de `(a:x)`. O valor deste cálculo será passado como parâmetro, juntamente com o penúltimo item da lista, tornando repetitivo esse processo até que o primeiro caso da

função `foldr` seja satisfeito. Ainda utilizando este exemplo, as funções `sum`, `prod` e `reverse`, são funções que respectivamente, somam, multiplicam e invertem os valores de listas. São exemplos de aplicações parciais da função `foldr` e podem ser descritas da seguinte forma:

```
sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
          where postfix a x = x ++ [a]
```

2.2.1.3 Avaliação Preguiçosa

Dentro de um código escrito em Haskell, os valores das variáveis e dos argumentos são avaliados, apenas, quando realmente necessário. Esse procedimento, segundo Hutton (2016), é denominado avaliação preguiçosa (*lazy evaluation*). Como consequência disso, é permitida a definição de funções não estritas[1]. A linguagem permite a possibilidade da criação de estruturas de dados infinitas, como uma lista infinita de números naturais, que também é permitida, graças à avaliação preguiçosa:

```
nats = [0..]
```

Para se representar dados infinitos em listas, pode-se utilizar, no caso de numerais, dois pontos (`..`) não explicitando o número final da lista. Através da recursão e do operador de composição de listas, também pode-se criar listas infinitas:

```
rpt wrd = wrd : rpt wrd
```

A função `rpt` recebe como parâmetro um número ou uma palavra e cria uma lista infinita repetindo estes valores. A utilização de listas infinitas pode ser muito útil para o desenvolvimento de funções de forma mais eficiente e, também, na representação de redes de processos de comunicação.

2.2.1.4 Polimorfismo

Uma função ou expressão em Haskell sempre possuirá um tipo associado à mesma, sendo uma linguagem fortemente tipada. Os tipos das funções são inferidos em tempo de compilação, ocorrendo inconsistências na estrutura da expressão, deverá ser emitida uma mensagem de erro (ALLEN; MORONUKI, 2017).

Além disso, as notações de listas e de tuplas podem ser utilizadas na composição de tipos. Utilizando o tipo `Bool`, por exemplo, presume-se que `[Bool]` é uma lista contendo valores do tipo `Bool`. É possível, também, uma lista possuir como elementos outras listas, da seguinte forma:

[1] Uma função é não estrita (*non-strict*) se ela permitir o retorno do resultado de uma função, mesmo que um de seus argumentos seja indefinido.

```
troo = [[True, False], [False, True], [False, False]]
```

O tipo da lista `troo` será `[[Bool]]`, uma lista de listas contendo valores booleanos. As funções e, por consequência, os programas também possuem tipos associados. Embora não seja necessária para a construção de funções, a linguagem permite a utilização da assinatura de tipos, que é dada pelo operador `::`, conforme o exemplo da função `rpt` a seguir:

```
rpt :: Char -> [Char]
rpt wrd = wrd : rpt wrd
```

A assinatura de tipos da função `rpt`, diz que o tipo da função `rpt` é `Char -> [Char]`, que aceita como entrada argumentos do tipo `Char` gerando um resultado do tipo `[Char]`. Ou seja, uma lista contendo valores do tipo `Char`. Caso não seja informada a assinatura do tipo de uma função, a linguagem, através do sistema de tipos, consegue inferir em tempo de compilação o tipo desta função, caso ele exista.

Haskell trata do caso onde ocorrem tipos polimórficos, representados por variáveis de tipos (`a`, `b`, ...). A função identidade, abaixo, é um exemplo de função que recebe dados de tipos polimórficos:

```
id x = x
```

A função `id` recebe um valor `x`, de qualquer tipo, e retorna o mesmo `x`. Para não se restringir a um determinado tipo específico (somente `Int` ou `Bool`), `id` será do tipo:

```
id :: a -> a
```

Isto garante à função `id` a possibilidade de receber, como parâmetro de entrada, um valor de qualquer tipo, retornando o mesmo valor com o mesmo tipo. Essa característica permite o reaproveitamento de funções. Outros exemplos de funções polimórficas são as funções de projeção sobre duplas:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

Essas funções polimórficas recebem, como argumento de entrada, uma dupla contendo valores de qualquer tipo, e os retornam, conforme a construção da função (`fst` retorna o primeiro, e `snd` o segundo). Vale ressaltar que a assinatura de tipos em Haskell não é necessária, pois o processo de inferência permite atribuir um tipo a cada função.

2.2.1.5 Definição de Tipos

Tipos similares a tipos primitivos da linguagem podem ser definidos, utilizando a ideia de construtores de tipos, da seguinte forma:

```
data Bool     = True | False
data Nat      = Zero | Suc  Nat
data List a   = Nill | Cons a  (List a)
data Pair a b = MakePair a b
```

Isso pode ser concebido por intermédio do construtor "`data`", nome do tipo, "`=`" seguido pela composição deste tipo. Essa construção pode ser um tipo produto, como uma tupla (`Int, Char`), ou uma união disjunta representada pelo símbolo `|`. Estes tipos criados são chamados de tipos de dados algébricos (*Algebraic Datatypes*). Esse mecanismo pode ser usado na criação de novos tipos de dados, por exemplo, na representação de árvores, como neste exemplo de uma árvore binária:

```
data BTree = Nil | Node Int BTree BTree
```

Esse novo tipo, `BTree`, insere 3 identificadores: o próprio `BTree`, representando o nome do tipo, `Nil`, e `Node` que representam os construtores de dados (construtores e tipos sempre devem iniciar com letra maiúscula). O construtor `Node` recebe 3 parâmetros: um inteiro `Int`, duas chamadas de `BTree` e o termo `Nil`, que é um construtor atômico. Utilizando o tipo definido `BTree`, pode-se gerar a seguinte árvore binária:

```
branch = Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)
```

O construtor `Nil` representa a inexistência de mais ramos da árvore a partir daquele momento. Os tipos definidos pelo usuário também podem ser polimórficos, formados pela introdução de tipos mais genéricos nos tipos de variáveis. Utilizando o exemplo anterior `BTree`, pode-se generalizar a definição de árvores, rescrevendo-o da seguinte forma:

```
data BTree a = Nill | Node a (BTree a) (BTree a)
```

Esta adaptação `BTree` permite que sejam criadas árvores que armazenem dados de qualquer tipo `a` em cada um de seus nós, diferente do exemplo anterior que permite apenas elementos inteiros. Além dos tipos algébricos, podem ser introduzidos novos nomes, chamados de sinônimos de tipo (*Type Synonyms*), para os tipos padrões da linguagem, como no exemplo a seguir:

```
type String = [Char]
```

O tipo `String` é um sinônimo de tipo para o tipo `[Char]`. Sinônimos de tipos podem ser introduzidos pela palavra reservada `type`, seguida do nome do novo tipo, "`=`" terminando na construção do tipo já existente.

2.2.1.6 Classes de Tipos (Typeclasses)

De acordo com Hutton (2016), uma classe em Haskell, pode ser vista como uma coleção de tipos que suportam determinadas sobrecargas de funções. Sobre-

carga de função corresponde a ideia de uma determinada função poder ser aplicada a alguns tipos de dados diferentes. A classe `Eq`, pré-definida na biblioteca padrão de Haskell, é um exemplo de classe de tipo para a igualdade de termos:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    b /= c = not ( b == c )
```

A declaração acima denota que um tipo `a` poderá ser uma instância da classe `Eq`, se for fornecida uma definição sobrecarregada para os operadores relacionais `==` e `/=`. As definições sobrecarregadas devem ter como tipo uma instância da assinatura definida na classe. Com base nisso, tem-se a definição da instância de `Eq` para o tipo `Bool`:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

A instância, utilizando tipos booleanos para a classe tipo `Eq`, mostra as combinações de valores `True` e `False`, e seus retornos, quando usado `"=="`. Para que essa função possa ser utilizada, deve existir uma instância, dentro da classe `Eq`, que suporte o tipo do valor a ser passado como entrada. Por padrão, Haskell fornece diversas *classes de tipos*, com suas instâncias definidas.

Pode, o programador, definir suas próprias instâncias de tipos e classes. Como por exemplo, a partir da definição do tipo algébrico `Shapes`, ser definida uma instância para a classe `Eq`, contendo este tipo:

```
data Shapes = Circle | Retangle

instance Eq Shapes where
    Circle   == Circle   = True
    Retangle == Retangle = True
    _        == _        = False
```

Essa instância garante ao operador de igualdade de termos a propriedade de atuar na comparação entre valores dos tipos instanciados. O último caso dessa estrutura, que apresenta um *underscore* ("_") dos dois lados da igualdade, remete ao caso onde os valores são diferentes, retornando `False`. Existe ainda o suporte à hierarquia de classes, como a da classe pré-definida na biblioteca padrão da linguagem, `Ord`:

```
class (Eq a) => Ord a where
    (<)  :: a -> a -> Bool
    (>)  :: a -> a -> Bool
```

```
(<=)  :: a -> a -> Bool
(>=)  :: a -> a -> Bool
```

A descrição dessa classe de tipos, força que as declarações contidas na classe `Ord`, restrinjam todas as instâncias dessa classe a serem, também, instâncias de `Eq`. A definição da linguagem apresenta, por padrão, diversas classes já implementadas. A classe de tipos `Num`, por exemplo, é dada da seguinte forma:

```
class Num a where
    (+), (*), (-)        :: a -> a -> a
    negate, abs, signum  :: a -> a
    fromInteger          :: Integer -> a
```

Para um tipo ser instância da classe `Num` é necessário que seja fornecida uma implementação dessas operações para o tipo. Os tipo numéricos: `Int`, `Integer`, `Float` e `Double` são declarados como instâncias dessa classe na biblioteca padrão. Neste caso, as instâncias contém apenas o identificador da classe a qual pertencem, seguido pelo tipo que será suportado. O conceito de classes de tipos pode ser muito útil na construção de programas em Haskell e também é empregado em outros mecanismos da mesma, como nas mônadas.

2.2.1.7 Mônadas

Como supracitado na Seção 2.2, linguagens funcionais puras não permitem mudanças de valores no estado de memória. Essas modificações poderiam implicar em algum tipo de alteração no resultado de uma computação causando algum tipo de efeito colateral. A ausência de efeitos colaterais pode permitir a garantia de algumas propriedades no desenvolvimento de um programa, gerando maior segurança no desenvolvimento de códigos. Porém isto pode limitar o desenvolvimento de programas, pois podem existir muitas situações em que seja necessário alterar valores de memória, como por exemplo a leitura de arquivos ou a impressão de informações na tela do usuário.

Haskell utiliza meios que simulam efeitos colaterais mutáveis (por necessidade de alterar valor de uma variável, por exemplo). Esses mecanismos proveem recursos para serem utilizados em um maior número de aplicações. Essa simulação de efeito colateral é feita por meio da composição de funções, que impõe uma sequência de execução para estas operações, podendo propagar dados implicitamente.

Wadler (1992) define mônada como um conceito matemático inspirado na teoria das categorias, utilizado em computações que possam requerer algum tipo de efeito colateral. De forma prática, uma mônada é representada, em Haskell, pela sobrecarga dos operadores de composição >>, >>= e `return`, que são definidos na classe de tipos `Monad`.

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
```

A função >>= recebe um argumento do tipo `m a`, e uma função de tipo `(a -> m b)` e retorna a composição dos parâmetros que terá tipo `m b`. O operador >>= é responsável por vincular o segundo argumento ao resultado da primeira computação, sendo também chamado de *bind*. A composição >>, pode ser vista com uma adaptação da função >>=, pois esta não propaga o dado `a`, retornando sempre o mesmo valor (`m b`). Finalmente, a expressão `return` recebe um argumento do tipo `a`, e retorna outro, do tipo `m a`. Essa função é uma versão monádica da função identidade, que recebe um parâmetro e retorna ele mesmo, neste caso, retorna-se um valor monádico. A partir dessa definição da classe `Monad` diversas mônadas podem ser criadas e utilizadas, como é o caso da mônada de entrada e saída.

2.2.1.7.1 Mônada de Entrada e Saída

Entrada e saída de dados envolvem efeitos colaterais. Usualmente, uma entrada de dados afeta um estado de memória que irá afetar computações futuras. Em Haskell estas operações caracterizam-se pela aplicação do tipo monádico `IO a`. Esta abordagem, conforme Thompson (2011), permite a comunicação de programas com o mundo externo. Basicamente, uma Mônada IO controla como são construídos programas que operam com entrada e saída de dados. Esse controle faz com que as operações realizadas dentro da estrutura não afetem as demais funções de uma aplicação. De forma prática, o tipo `IO a` pode, além de retornar um tipo genérico `a`, efetuar operações de entrada e saída, como as funções `getChar` e `putChar`, a seguir:

```
getChar  :: IO Char
putChar  :: Char -> IO ()
```

Estas duas funções, contidas na biblioteca padrão da linguagem, permitem a captura (`getChar`) de caracteres, via teclado, e a impressão (`putChar`) destes, na tela. Somadas a estas funções, mônadas permitem o uso da notação `do` no lugar dos operadores de composição. A notação `do`, muito utilizada em programas em Haskell, pode ser considerada um açúcar sintático (*syntactic sugar*), ou seja, uma forma mais inteligível de se utilizar expressões monádicas na construção de um programa. Considerando o seguinte bloco:

```
foo' = readFile "input.txt" >>= writeFile "output.txt"
```

Esta expressão efetua a leitura de um arquivo `input.txt` e copia o seu conteúdo (uma `String`) para outro arquivo nomeado `output.txt`. Usando `do`, reescreve-se essa função da seguinte forma:

```
foo' = do
          aux <- readFile "input.txt"
          writeFile "output.txt" aux
```

O uso desta sintaxe pode tornar o código mais simples de ser entendido, tornando mais fácil a visualização do que está sendo implementado. Esta forma de escrita exige que cada linha da construção `do`, seja sempre uma função que retorne um valor de tipo monádico. Utilizando outra função padrão `putStrLn` da mônada IO, pode-se escrever a ingênua computação monádica, a seguir:

```
rpt2 wrd = do
             putStrLn wrd
             putStrLn wrd
```

Esta função `rpt2` recebe como entrada uma `String` e imprime na tela duas vezes este mesmo valor, utilizando a função `putStrLn`. O comando `putStrLn` tem o tipo `String -> IO ()`, sendo padrão da linguagem.

O funcionamento de Haskell não exige uma ordem específica na informação e leitura de parâmetros e funções. Porém, para se operar entrada e saída de dados, a ordem de passagem de valores e ações é relevante. Para Thompson (2011), o tipo `IO a` pode ser visto como ações que acontecem em sequência. Podendo ser interpretado como uma espécie de linguagem imperativa, dentro de Haskell, que não afeta o restante do código. Dentro desta estrutura monádica, um valor qualquer de entrada será lido, e com base nele, uma saída pode ser retornada, ou outra entrada pode ser realizada.

Outra forma de se entender o funcionamento de uma mônada de IO é supondo que ela contém um valor do tipo `IO a`. O que pode ser representado por uma função que recebe um estado qualquer como argumento, e que produz um outro valor, que altera esse estado. Portanto, uma mônada IO pode ser vista, também, como uma *mônada de estados*.

2.2.1.7.2 Mônada de Estados

A concepção de estado, conforme Allen e Moronuki (2017), no contexto de linguagens de programação, tem origem na teoria de circuitos e autômatos, pode ser entendido como um interruptor, tendo dois possíveis modos: ligado ou desligado. E a partir da posição deste, no momento, será definido seu estado. Aplicando à realidade

das linguagens de programação, um estado é composto por dados, entradas e saídas, que podem ser alterados a cada avaliação de uma determinada função.

Haskell não permite a alteração de valores de estado, devido a características relacionadas à sua pureza, conforme explanado no início da Seção 2.2. Porém, para diversas aplicações essa alteração pode ser necessária. E com a aplicação de uma *mônada de estados* tais mudanças podem ser efetuadas, permitindo a linguagem operar sobre estados mutáveis.

O transformador de estados, ou simplesmente ST (*state transformer*), segundo Wadler (1992), é uma função aplicada a estados. De forma genérica, uma instância de classe de `Monad` para um tipo `MS a` pode ser descrita conforme a Figura 5.

Figura 5 – Definição - de *bind* e *return*

```
1 data MS a = ME (State -> (a, State))
2
3 type State = Int
4
5 instance Monad MS where
6    return x    = ME (\e -> (x, e))
7    ME m >>= f  = ME (\e -> let (a, e') = m e;
8                                ME fa = f a in fa e')
```

Fonte: O Autor

A Figura 5 representa uma mônada de estados com suas declarações. O tipo algébrico `MS a` define o transformador de estados que recebe o estado atual e retorna: um resultado `a` e um estado alterado. O tipo `State` foi estabelecido como inteiro para facilitar a visualização da troca de estados. Define-se também uma instância para `Monad MS a`. Dentro dessa instância estão definidas as funções *bind* e `return`. A função `return` recebe um parâmetro `x` e retorna o transformador de estados, computando uma dupla com o valor de `x` e o estado `e`. No operador de *bind*, aplica-se o transformador de estados `m` no estado `e`, retornando uma tupla contendo: um valor `a` e um estado `e'`. Em seguida, aplica-se a função `f` no valor `a`. Por fim, o resultado de `fa` será aplicado no estado `e'`. Com a definição da funções da instância, pode-se criar as funções exibidas na Figura 6.

A expressão `runME` recebe uma função monádica, desconstruindo-a em `m`, que recebe o argumento implícito 0. Este valor será utilizado como valor inicial da computação de estados. Em seguida define-se `counter`, que incrementa 1 ao valor de estado `e`, independente do resultado da computação. Este contador pode ser aplicado a diversas funções; neste caso, foi aplicado à função de máximo divisor comum `gcd`. A cada chamada recursiva da função `gcd` o contador será incrementado. Por acionar a função `counter`, que faz uso de um transformador de estados, a função `gcd` retornará

Figura 6 – Exemplo - Cálculo do máximo divisor comum

```
1 runME (ME m) = m 0
2 counter = ME (\e -> ((), e+1))
3
4 gcd a b | a == b     = do {counter; return a}
5         | a < b      = do {counter; gcd a (b-a)}
6         | otherwise = do {counter; gcd (a - b) b}
```

Fonte: O Autor

um valor do tipo `MS a`. O retorno desta computação será um tupla contendo: o resultado do cálculo e o valor do estado ao término da execução da função.

2.2.1.7.3 Mônada Maybe

A mônada `Maybe` é um exemplo de mônada muito utilizada em Haskell. Ela permite, entre outras coisas, manipular computações que potencialmente possam dar errado, buscar um valor em uma lista e não encontrá-lo, por exemplo. A partir da declaração da classe `Monad`, e da definição do tipo `Maybe a`, pode-se definir `Maybe` como sendo uma instância da classe `Monad`, conforme a Figura 7.

Figura 7 – Definição e Exemplo - *Monad Maybe*

```
1 data Maybe a = Just a | Nothing
2
3 instance Monad Maybe where
4     return x       = Just x
5
6     Nothing  >>= _ = Nothing
7     (Just x) >>= f = f x
```

Fonte: O Autor

O tipo `Maybe`, disponível na biblioteca padrão de Haskell, é normalmente usado no retorno de funções que podem computar um valor (`Just a`), ou falhar (`Nothing`). Declarando o tipo `Maybe` como uma instância da classe `Monad` é possível compor funções que retornem dados do tipo `Maybe`. A composição deve propagar o valor retornado para a próxima função, no caso da computação ter resultado em um dado. Se a computação falhar em qualquer uma das funções, reproduz-se `Nothing` até o final da composição. Como exemplos de utilização do tipo `Maybe`, a seguinte função pode ser implementada:

```
search :: Int -> [(Int, String)] -> Maybe String
search _ []              = Nothing
search idt ((x1,x2):xs) | idt == x1 = Just x2
                        | otherwise = search idt xs
```

A função `search` irá receber como parâmetros: um número inteiro (um identificador, por exemplo) e uma lista de duplas, contendo um inteiro e uma palavra, um índice ($identificador,\ palavra$). Ela efetuará a busca do valor `idt` na lista de índices `((x1,x2):xs)`. Caso o valor não seja encontrado, será retornado um tipo `Nothing`, que representa um erro. Do contrário será retornada a `String` correspondente ao índice buscado, acompanhada do Construtor `Just`. A declaração do tipo `Maybe` como uma instância da classe de tipos `Monad` permite o uso da sintaxe `do` o que facilita o tratamento dos casos de exceção (quando alguma das funções retorna `Nothing`):

```
search'' :: Int -> Int -> [(Int, String)] -> Maybe (String, String)
search'' id1 id2 = do
                    r1 <- search id1 lst
                    r2 <- search id2 lst
                    return (r1,r2)
```

Este fragmento de código utiliza a função `search`, definida na subseção anterior, para buscar um valor associado ao seu índice em uma lista. O tipo `Maybe` está implícito no resultado da computação `search`. Se o valor de `r1` não for encontrado, então `Nothing` será propagado, interrompendo a computação, emitindo um erro na forma do tipo `Nothing`. Caso este seja encontrado, aguardará o resultado da computação para `r2`. Se o segundo valor for localizado, retorna-se uma dupla contendo os valores de `r1` e `r2`. Na condição de não ser encontrado o segundo valor, será emitido `Nothing` , que pode ser interpretado como um erro.

Somadas a essas mônadas básicas (entrada e saída, estados e *Maybe*) existem outras, dentro de bibliotecas da linguagem, que podem ser manipuladas de acordo com a necessidade do programador. Um exemplo é a biblioteca `Control.Monad.State`, que permite a captura, a inserção e a modificação do valor de estados. Além dessas características, Haskell apresenta outras vantagens como: suporte a paralelismo e concorrência de forma simplificada, garantias de seu sistema de tipos, elegância no desenvolvimento de códigos, e também, uma comunidade atuante provendo a evolução da linguagem e de seus compiladores.

Porém, especificamente no desenvolvimento de códigos de forma segura, embora apresente garantias de tipos e permita a manipulação de efeitos colaterais, a linguagem não permite, ainda, o que se chama de *tipagem de efeitos*. Esse conceito, encontrado na linguagem de programação funcional Koka, é relativamente novo e visa aumentar a facilidade na detecção de possíveis erros no desenvolvimento de programas de computador.

2.3 KOKA

Linguagens como Haskell e Miranda trazem garantias quanto a segurança na execução e desenvolvimento de um programa. Essas garantias são advindas, principalmente, pela adoção de uma extensão do sistema de tipos de Damas e Milner (1982). Porém, no caso específico de Haskell, não se oferece nenhum tipo de mecanismo que informe algo a respeito de efeitos colaterais, exceto sua simulação em mônadas. Muitas vezes, no desenvolvimento de um programa, pode ser necessária a combinação de efeitos colaterais. É importante que essa combinação ocorra de forma segura, sem erros, a fim de tornar mais eficiente o desenvolvimento de uma aplicação (LEIJEN, 2014).

Diferentemente das linguagens funcionais (puras e impuras) citadas neste trabalho, Koka não está somente focada nos tipos dos dados de entrada e saída de programas. A inferência de tipos destas linguagens não é suficientemente precisa para informar alguns aspectos a respeito do comportamento estático de um programa. Leijen (2014) sugere que seja incluído, a este sistema de tipos, mecanismos que forneçam informações referentes a potenciais efeitos colaterais. Estes conhecimentos podem ser significativos em várias circunstâncias como: paralelismo, otimizações, aplicações de consultas e em sistemas distribuídos.

O modelo de programação proposto por Koka (LEIJEN, 2014), além de tratar inferência de tipos e de efeitos, traz uma maior familiaridade, na sua sintaxe, ao programador habituado a utilizar linguagens de programação imperativas e orientadas a objeto. O bloco abaixo demonstra a sintaxe de uma função escrita em Koka:

```
function sqr (x:int) : total int
{  return x*x  }
```

O código somente retorna o quadrado de um valor `x`. Nota-se, também, após a atribuição de tipos da variável `x`, informação referente à saída de tipo e do tipo de efeito desta função, representado por `total int`. Essas informações representam a assinatura de tipos dessa função (Koka utiliza o operador `:` para informar que a função tem como saída um determinado tipo), podendo ser entendida da seguinte forma:

```
sqr : int -> total int
```

Neste caso, não há efeitos colaterais; nenhum estado é alterado e há apenas uma saída possível para a função. Portanto, o tipo de efeito `total` não é, de fato, um efeito colateral. O tipo de efeito `total` é utilizado para marcar que determinada computação é pura e que termina. Usando um possível componente que cause efeito colateral a valer, esta função pode ser reescrita da seguinte forma:

```
function sqr2 (x:int)
{  print(x*x);  }
```

A função `sqr2` efetua a mesma operação da função `sqr`, porém, imprime na tela o resultado da computação, gerando um efeito colateral de entrada e saída. Nota-se, também, assim como em Haskell e Miranda, que a assinatura de tipo não é necessária:

```
sqr : int -> io int
```

Leijen (2016) explica que um dos objetivos do modelo de programação utilizada em Koka é fazer com que as expressões com possíveis efeitos colaterais sejam combinadas de forma automática por meio da inferência de efeitos. Neste exemplo foi combinado o efeito `total` da multiplicação com o `io` da função `print`, dando a esta função a saída `io int`.

Koka faz a avaliação dos argumentos de uma expressão de forma estrita, isto é, efetua a avaliação de um argumento informado antes da execução da função. Dentro do ecossistema de Koka efeitos colaterais só podem aparecer durante a aplicação de funções. O tipo de uma função, neste sistema, é escrita na forma $\tau \rightarrow \epsilon\ \tau'$, significando que uma função recebe como entrada os argumentos de τ e retorna um valor do tipo τ', com um possível efeito ϵ quando aplicada.

Para seu funcionamento, Koka necessita que as funções estejam totalmente aplicadas e não currificadas. Isso garante que efeitos colaterais sejam imediatamente identificados. Em uma linguagem como ML, uma expressão $f\ x\ y$ pode ter efeitos colaterais em pontos diversos, dependendo da aridade da função f. No modelo proposto, efeitos aparecem imediatamente, de modo que essa função será escrita como $f\ (x,\ y)$ ou $(f\ (x))\ (y)$ (LEIJEN, 2014).

A linguagem permite que o desenvolvedor crie seus próprios tipos de efeitos, de acordo com a sua necessidade. Porém, são fornecidos alguns tipos básicos, de forma padrão. Além dos abordados `total` (funções puras) e `io` (entrada e saída), estão disponíveis: exceções (`exn`), divergências[2] (`div`) e funções não determinísticas (`ndet`). Os efeitos para funções de estados que operam em *heaps*, são: `alloc`$\langle$`h`$\rangle$, `read`$\langle$`h`$\rangle$ e `write`$\langle$`h`$\rangle$. Há, também, a possibilidade da combinação de efeitos, considerando o bloco a seguir:

```
function sqr3 ( x: int ) {error("ERROR"); sqr3(x); x*x}
```

Esta função, assim como as anteriores, efetua o simples cálculo do quadrado de um inteiro, porém, desta vez, há a função `error` que emite na tela qualquer possível erro ocorrido durante a execução da função. A inferência do tipo e do efeito desta computação será:

[2] Entende-se, nesse contexto, que o efeito divergente corresponde a computações que potencialmente possam não terminar. Koka efetua a verificação de terminação dos algoritimos por meio de um mecanismo descrito no trabalho de Abel (1998).

```
sqr3 : int -> ⟨exn, div⟩ int
```

Os efeitos `exn` e `div`, combinados, são chamados de *effect rows*. Esses efeitos são explicitados entre colchetes angulados ⟨ ⟩. Leijen (2014) expõe que essa agregação dos efeitos `exn` e `div` corresponde exatamente à noção de pureza de Haskell, chamando este efeito de `pure`. Ainda em Haskell, não há garantia advinda do sistema de tipos de que uma função sempre terminará e nem que essa função não venha a emitir uma exceção. Isso significa que em Haskell qualquer função pode gerar os efeitos `exn` e `div`.

Para se referir à agregação de efeitos em um construtor, utiliza-se: `alias`, seguido do nome dado, o operador =, e os efeitos a serem agrupados. Além do `alias` para `pure`, podem ser citados outros padrões da linguagem, como:

```
alias st⟨h⟩ = ⟨alloc⟨h⟩, read⟨h⟩, write⟨h⟩⟩
alias io    = ⟨st⟨ioheap⟩, pure, ndet⟩
```

O `alias` para `st` e `io` compõe diversos efeitos que cada um destes suporta. Essa hierarquia é muito semelhante à definição padrão de Mônadas, utilizada em Haskell, podendo servir como base para a criação de efeitos mais eficientes.

2.3.1 Efeitos Polimórficos

Uma das maiores dificuldades encontradas na tipagem de efeitos, segundo Leijen (2014), está contida na inferência de efeitos polimórficos. Normalmente, o efeito de uma função é definido pelos efeitos das funções utilizadas como parâmetro. O mapeamento de uma função para todos os elementos contidos em uma lista, a função `map`, tem seu tipo atribuído da seguinte forma:

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

Exemplificando o que foi citado, o efeito de `map` é determinado pelo tipo de efeitos de seus parâmetros de entrada. Especificamente neste caso, a inferência não se mostra muito complexa, porém, em exemplos maiores podem ser encontradas algumas dificuldades. Para isso, atualmente, Koka utiliza *row-polymorphism*, método utilizado em outras aplicações de inferência para cálculo de registros (*record calculi*).

A notação $\langle\iota|\mu\rangle$ representa a extensão de um *row effect* μ com uma constante de efeitos ι. Estes *row effects* podem ser de duas formas: fechados ($\langle exn, div\rangle$) ou abertos ($\langle exn, div \mid \mu\rangle$). A função `bar`, abaixo, é um exemplo de função que apresentará um efeito polimórfico mais difícil de ser inferido:

```
function bar(f, g) { f(), g(); error("ERROR") }
```

Esta função, utilizando a ideia de *row effect*, apresentará o seguinte efeito aberto:

$$bar : \forall\mu.\ (() \to \langle exn \mid \mu\rangle\ (), \langle exn \mid \mu\rangle\ ()) \to \langle exn \mid \mu\rangle\ ()$$

O método é restritivo e atua impondo que cada argumento desta função seja do mesmo efeito $\langle exn \mid \mu\rangle$. Na avaliação da função `bar` com os parâmetros `(f, g)`, o sistema de efeitos garante a inferência de efeitos abertos para estas duas funções, com base na seguinte suposição de tipos para as funções `f` e `g`:

$$\texttt{f} : () \to \langle exn \mid \mu_1\rangle\ ()$$
$$\texttt{g} : () \to \langle div \mid \mu_2\rangle\ ()$$

Pode-se inferir o tipo de efeito $\langle exn, div \mid \mu_3\rangle$ na função `bar (f, g)`. Para isso, unifica-se os tipos $\langle exn \mid \mu_1\rangle$ e $\langle div \mid \mu_2\rangle$, gerando um tipo $\langle exn, div \mid \mu_3\rangle$ que pode ser aplicado em `bar`.

Uma particularidade desse sistema está em como se abordam os efeitos duplicados, sendo estes, permitidos na proposta de Koka. Segundo Leijen (2014), permitir efeitos duplicados, na forma $\langle exn,\ exn\rangle$, faz com que não sejam necessárias restrições extras. Esse detalhe concede, também, de forma facilitada, a inferência de efeitos para formas de eliminação, como a captura de exceções.

2.3.2 Alterações de Estados

Conforme abordado nas seções anteriores, alterações de estados são muito utilizadas no desenvolvimento de programas, sendo um exemplo de computação que causa efeitos colaterais. Para essas operações, Koka utiliza os intitulados efeitos de *Heap*. A ideia remete ao uso da função `runST` de Haskell, sendo referida, aqui, apenas como `run`, tendo a seguinte atribuição de tipo:

$$run : \forall\mu\alpha.(\forall h.\ () \to \langle st\langle h\rangle \mid \mu\rangle\ \alpha) \to \mu\ \alpha$$

O sistema de efeitos de Koka insere, de forma automática, uma função `run` nas generalizações, quando estas puderem ser aplicadas, inferindo o tipo da expressão. É importante salientar que polimorfismo combinado a estados mutáveis pode causar problemas. Leijen (2014) demonstra que, em Koka, tratar de programas que combinem polimorfismo e mutação de estados não necessita de manobras complexas, como em ML. Usando tipos de efeitos impõe-se que a generalização seja aplicada somente a computações com efeito `total`, ou seja, computações puras. A utilização de `run` acrescenta requisitos para se ter a garantia de que expressões com efeitos colaterais

estejam encapsuladas, comportando-se como funções puras, não contaminando o restante do programa.

O tipo de efeito `div` representa uma divergência e é utilizado em computações que, potencialmente, não tenham terminação. Para fazer isso de forma segura, Koka utiliza um conjunto com três construtores de tipos: indutivos, co-indutivos e construtores de tipos recursivos arbitrários. Qualquer função que utilize tipos de dados recursivos é considerada potencialmente divergente, pois permite a criação de uma função não terminal, sem ser sintaticamente recursiva. Funções recursivas também apresentam o tipo de divergência em geral, porém, se for verificado que cada chamada recursiva diminui o tamanho de um tipo indutivo, então não será atribuído o efeito `div` a esta função.

Ainda conforme Leijen (2014), podem ser encontradas na literatura, diversas propostas para sistemas de efeitos. Porém, estes sistemas, em sua maioria, consistem em aparatos sintáticos ou atuam sobre pequenos conjuntos de efeitos. Inferir efeitos colaterais não é uma tarefa trivial, principalmente quando se envolve polimorfismo, requerendo algumas restrições e abordagens diferentes.

2.3.3 Sistema de Efeitos

O sistema de efeitos proposto por Leijen (2014), para Koka, apresenta algumas particularidades para estar apto a efetuar a inferência de tipos e efeitos colaterais. Um desses detalhes é a sua sintaxe de tipos e *kinds* que, formalmente, pode ser definida conforme a Figura 8. Dentro desta sintaxe, existe a restrição de que constantes de efeitos não podem ser variáveis de tipo.

Uma expressão bem tipada, neste contexto, é dada pelo sistema de *kinds*, onde um *kind* κ está disposto em sobrescrito nos tipos τ, como: τ^{κ}. Há, ainda, a utilização do *kind* $(*)$ e da aplicação de função $\rightarrow$, porém, somente para: *effect rows* e, constantes de efeitos k e *heaps* h. As variáveis de tipo são representadas por α, e o termo μ é utilizado para variáveis do tipo de efeitos e ξ, denota variáveis de *heap*. Os tipos dos efeitos são definidos como um *row* de rótulos de efeitos ι. A Tabela 1 exibe os símbolos, descrições e como são utilizados nesse sistema.

Tabela 1 – Notação - Variáveis de Efeitos

Sìmbolo:	Descrição:	Uso:
μ	Variável de Efeito	α^{e}
ϵ	Tipo de Efeito	τ^{e}
ι	Tipo do Efeito de Constante	τ^{e}
ξ	Variável *Heap*	α^{h}
h	Tipo *Heap*	τ^{h}

Fonte: Leijen (2014)

Figura 8 – Definição - Sintaxe do Sistema de *kinds*

$kinds\ \kappa$	$::=$	$*$	*tipos de valor*
	$\mid$	e	$effect\ rows$
	$\mid$	k	*constantes de efeitos*
	$\mid$	h	*heap*
	$\mid$	$(\kappa_1\ ,\ldots,\kappa_n)\ \rightarrow\ \kappa$	*construtor de tipos*
$tipos\ \tau^\kappa$	$::=$	α^κ	*variável de tipo*
	$\mid$	c^κ	*constante de tipo*
	$\mid$	$c^{\kappa 0}\ \langle\tau_1^{\kappa 1},...,\tau_n^{\kappa n}\rangle$	$\kappa_0 = (\kappa_1,...,\kappa_n) \rightarrow \kappa$
$schemes\ \sigma$	$::=$	$\forall\alpha_1...\alpha_n.\ \tau^*$	

constantes	$()$	$::\ *$	*tipo unitário*
	$(\_ \rightarrow \_\ \_)$	$::\ (*,\ \mathsf{e},\ *) \rightarrow *$	$funções$
	$\langle\rangle$	$::\ \mathsf{e}$	$efeito\ vazio$
	$\langle\_\mid\_\rangle$	$::\ (\mathsf{k},\ \mathsf{e}) \rightarrow \mathsf{e}$	$extensão\ de\ efeito$
	ref	$::\ (\mathsf{h},\ *) \rightarrow *$	$referências$
	exn	$::\ \mathsf{k}$	*parcial*
	div	$::\ \mathsf{k}$	*divergente*
	st	$::\ \mathsf{h} \rightarrow \mathsf{k}$	$stateful$

Fonte: Leijen (2014)

Para simplificar a apresentação formal do funcionamento do sistema de efeitos, Leijen (2014) define que as constantes sejam restringidas a: exceções (exn), divergências (div) e operações em *heaps* (st). Este sistema de *kinds* garante que efeitos sejam sempre fechados $\langle\iota_1,...\iota_n\rangle$ ou abertos $\langle\iota_1,...\iota_n \mid \mu\rangle$. Outra característica importante do funcionamento do sistema é a relação de equivalência entre os tipos de efeitos, exibida na Figura 9.

Figura 9 – Definição - Relação de Equivalência

$\epsilon \equiv \epsilon$	**(EQ-REFL)**
$\dfrac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}$	**(EQ-TRANS)**
$\dfrac{\iota_1 \equiv \iota_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle\iota_1\mid\epsilon_1\rangle \equiv \langle\iota_2\mid\epsilon_2\rangle}$	**(EQ-HEAD)**
$\dfrac{\iota_1 \not\equiv \iota_2}{\langle\iota_1\mid\langle\iota_2\mid\epsilon\rangle\rangle \equiv \langle\iota_2\mid\langle\iota_1\mid\epsilon\rangle\rangle}$	**(EQ-SWAP)**
$c\langle\tau_1,...,\tau_n\rangle \equiv c\langle\tau_1',...,\tau_n'\rangle$	**(EQ-LAB)**

Fonte: Leijen (2014)

Essas relações de equivalência computam os efeitos que podem ser equiva-

lentes, sem levar em consideração a ordem das constantes de efeitos. Em específico, a relação ($EQ - LAB$) define a relação de equivalência sobre constantes de efeitos, onde os argumentos de tipos não são relevantes. As regras de efeitos e tipos, dispostas na Seção 2.3.3.1, são aplicadas sobre o pequeno cálculo de expressões exposto na Figura 10:

Figura 10 – Definição - Cálculo de expressões para o sistema de efeitos

e	::=	x	$(variáveis)$
	|	p	$(primitivas)$
	|	$e_1\ e_2$	$(aplicação)$
	|	$\lambda x.e$	$(função)$
	|	$x \leftarrow e_1;\ e_2$	$(sequência)$
	|	$let\ x = e_1\ in\ e_2$	$(let\ binding)$
	|	$catch\ e_1 e_2$	$(captura\ de\ exceções)$
	|	$run\ e$	$(isolate)$
p	::=	$()$ | fix | $throw$ | new | $(!)$ | $(:=)$	

Fonte: Leijen (2014)

Na gramática exibida na Figura 10, a expressão $x \leftarrow e_1;\ e_2$ representa o vínculo monomórfico de uma variável x a uma expressão e_1, sendo equivalente à função: $(\lambda x.e_2)\ e_1$. As expressões run e $catch$ são simplificações que apresentam onde pode-se ter uma aplicação direta destas regras de tipo. Os termos $throw$ e $catch$ operam somente sobre exceções, limitados apenas ao tipo unitário $()$. A partir da relação de equivalência e das sintaxes expostas, podem ser definidas as regras de tipo para este sistema.

2.3.3.1 Inferência de Efeitos

A inferência de tipos e efeitos, deste sistema, é dada a partir de um contexto Γ. A construção $\Gamma \vdash e : \sigma \mid \epsilon$ é interpretada da seguinte forma: com base em um contexto Γ, a expressão e será do tipo σ contendo um efeito do tipo ϵ. As regras de tipo e efeitos de Koka (Figura 11), seguem o mesmo padrão do que foi proposto no trabalho de Damas e Milner (1982), com a inserção do efeito ϵ em cada uma das cláusulas.

A Figura 11 apresenta as regras de tipos utilizadas, neste sistema, na inferência de tipos e de efeitos. Este conjunto de regras segue o padrão exposto na Seção 2.1, com a inclusão de ϵ em todas as regras para representar efeitos. Ainda é possível notar que as regras *INST* e *GEN* foram incorporadas às regras *VAR*s e *LET*s, respectivamente. Especificamente, durante a utilização da regra *VAR*s sempre será atribuído um efeito ϵ que poderá ser combinado com qualquer outro tipo de efeito. Assume-se que no funcionamento do sistema, as atribuições de variáveis sempre serão computações puras, do tipo `total`.

Figura 11 – Definição - Regras do Sistema de Efeitos

$$\frac{\Gamma(x) = \forall\alpha.\tau}{\Gamma \vdash_s x : [\alpha \mapsto \tau]\ \tau \mid \epsilon} \quad \textbf{(VAR)}_s$$

$$\frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau \mid \epsilon}{\Gamma \vdash (\lambda x.e) : \tau' \to \epsilon'\ \tau \mid \epsilon'} \quad \textbf{(ABS)}$$

$$\frac{\Gamma \vdash e : \tau \to \epsilon\ \tau' \mid \epsilon \qquad \Gamma \vdash e' : \tau \mid \epsilon}{\Gamma \vdash (e\ e') : \tau' \mid \epsilon} \quad \textbf{(APP)}$$

$$\frac{\Gamma \vdash_s e : \sigma \mid \langle\rangle \qquad \alpha \notin free(\Gamma) \qquad \Gamma_x : \forall\alpha.\tau' \vdash_s e' : \tau \mid \epsilon}{\Gamma \vdash (let\ x\ =\ e\ in\ e') : \tau \mid \epsilon} \quad \textbf{(LET)}_s$$

$$\frac{\Gamma \vdash_s e : \tau \mid \langle st\langle\xi\rangle \mid \epsilon\rangle \qquad \xi \notin free(\Gamma, \tau, \epsilon)}{\Gamma \vdash run\ e : \tau \mid \epsilon} \quad \textbf{(RUN)}$$

$$\frac{\Gamma \vdash_s e : \tau \mid \langle exn \mid \epsilon\rangle \qquad \Gamma \vdash_s e' : ()\ \to \epsilon\tau\ |\epsilon}{\Gamma \vdash catch\ (e\ e') : \tau \mid \epsilon} \quad \textbf{(CATCH)}$$

Fonte: Leijen (2014)

São apresentadas, também, as regras *RUN* e *CATCH*, para aplicação em contextos específicos ao ser chamadas durante a inferência de efeitos. O algoritmo de inferência, utilizado por Koka, é semelhante ao algoritmo W, apresentado na Seção 2.1. O algoritmo aplica unificações e substituições. A Figura 12 apresenta o algoritmo disposto como regras formais.

Figura 12 – Definição - Algoritimo de inferência de tipos e efeitos

$$\frac{\Gamma(x) = \forall\overline{\alpha}.\tau}{\varnothing\Gamma \vdash_i x : [\overline{\alpha} \mapsto \overline{\beta}]\ \tau \mid \mu} \quad \textbf{(VAR)}_i$$

$$\frac{\theta\Gamma_x \cup \{x : \alpha\} \vdash_i e : \tau \mid \epsilon}{\theta\Gamma \vdash_i (\lambda x.e) : \theta\alpha \to \epsilon\ \tau \mid \mu} \quad \textbf{(ABS)}_i$$

$$\frac{\theta_0\Gamma \vdash_i e : \tau \mid \epsilon \quad \theta_1(\theta_0\Gamma) \vdash_i e' : \tau' \mid \epsilon' \quad \theta_3\tau \sim (\tau' \to \epsilon'\alpha) : \theta_2 \quad \theta_2\theta_1\epsilon \sim \theta_2\epsilon' : \theta_3}{\theta_3\theta_2\theta_1\theta_0\Gamma \vdash_i (e\ e') : \theta_3\theta_2\alpha \mid \theta_1\theta_2\epsilon} \quad \textbf{(APP)}_i$$

$$\frac{\theta_0\Gamma \vdash_i e : \tau \mid \epsilon \quad \epsilon\ \sim\ \langle\rangle : \theta_1 \quad \sigma = gen(\theta_1\theta_0\Gamma,\ \theta_1\ \tau) \quad \theta_2(\theta_1\theta_0\Gamma, x : \sigma) \vdash_i e' : \tau \mid \epsilon}{\theta_2\theta_1\theta_0\Gamma \vdash_i (let\ x\ =\ e\ in\ e') : \tau \mid \epsilon} \quad \textbf{(LET)}_i$$

$$\frac{\theta_0\Gamma \vdash_i e : \tau\ |\epsilon \quad \epsilon \sim \langle st\langle\xi\rangle \mid \mu\ \rangle; \theta_1 \quad \theta_1\xi \in TVar \quad \theta_1\xi \notin free(\theta_1\theta_0\Gamma,\ \theta_1\tau,\ \theta_1\mu)}{\theta_1\theta_0\Gamma \vdash_i\ run\ e : \theta_1\tau \mid \theta_1\mu} \quad \textbf{(RUN)}_i$$

$$\frac{\theta_0\Gamma \vdash_i e : \tau \mid \epsilon \quad \theta_2(\theta_1\Gamma) \vdash_i e' : \tau' \mid \epsilon' \quad \theta_1\epsilon 1 \sim \langle exn|\epsilon'\rangle : \theta_2 \quad \theta_2\tau' \sim () \to \theta_2\epsilon_1\theta_2\theta_1\tau : \theta_3}{\theta_3\theta_2\theta_1\theta_0\Gamma \vdash catch\ (e\ e') : \theta_3\theta_2\tau \mid \theta_3\theta_2\epsilon} \quad \textbf{(CATCH)}_i$$

Fonte: Leijen (2014)

De forma geral, estas regras podem ser interpretadas como atributos sintáticos, de modo que θ, τ, e ϵ são sintetizados e Γ e e, herdados. O algoritmo utiliza a ideia de unificações: $\tau_1 \sim \tau_2 : \theta$ que unifica τ_1 e τ_2 com a mais geral substituição θ, de forma que $\theta\tau_1 \ = \ \theta\tau_2$. Este algoritmo pode ser visto como uma adaptação do algoritmo W, acrescido de atributos para unificação de efeitos da função *Gen*, que generaliza um tipo com base em um contexto, sendo definida da seguinte forma:

$$gen(\Gamma, \tau) \ = \ \forall(free(\tau) - free(\Gamma)).\tau$$

A Figura 13 exibe o algoritmo de unificações utilizado no sistema de efeitos de Koka. Estas regras consistem em adaptações do algoritmo para registros com rótulos de escopo (*algorithm for records with scoped labels*), descritas no trabalho de Leijen (2005).

Figura 13 – Definição - Algoritmo unificações

$\alpha \sim \alpha : [\,]$	**(UNI-VAR)**
$\dfrac{\alpha \notin free(\tau)}{\alpha^k \sim \tau^k : [\alpha \mapsto \tau]}$	**(UNI-VARL)**
$\dfrac{\alpha \notin free(\tau)}{\tau^k \sim \alpha^k : [\alpha \mapsto \tau]}$	**(UNI-VARR)**
$\dfrac{\forall i \in 1..n. \quad \theta_{i_{-1}}...\theta_1\tau_i \sim \theta_{i_{-1}}...\theta_1 t_i : \theta_i \quad k = (k_1, ..., k_n) \to k'}{c^\kappa\langle\tau_1^{\kappa^1}, ..., \tau_n^{\kappa^n}\rangle \sim c^\kappa\langle t_1^{\kappa^1}, ..., t_n^{\kappa^n}\rangle : \theta_n...\theta_1}$	**(UNI-CON)**
$\dfrac{\epsilon_1 \simeq \iota \mid \epsilon_2 : \theta_0 \quad tl(\epsilon_1) \notin dom(\theta_0) \quad \theta_0\epsilon_0 \sim \theta_0\epsilon_2 : \theta_1}{\langle\iota \mid \epsilon_0\rangle \sim \epsilon_1 : \theta_2\theta_1}$	**(UNI-EFF)**
$\dfrac{\iota \equiv \iota' \quad \iota \sim \iota' : \theta}{\langle\iota' \mid \epsilon\rangle \simeq \iota \mid \epsilon : \theta}$	**(EFF-HEAD)**
$\dfrac{\iota \not\equiv \iota' \quad \epsilon \simeq \iota \mid \epsilon' : \theta}{\langle\iota' \mid \epsilon\rangle \simeq \iota \mid \langle\iota \mid \epsilon'\rangle : \theta}$	**(EFF-SWAP)**
$\dfrac{fresh\ \mu'}{\mu \simeq \iota \mid \mu' : [\mu \mapsto \langle\iota \mid \mu\rangle]}$	**(EFF-TAIL)**

Fonte: Leijen (2014)

O algoritmo (representado na Figura 13), como já citado, unifica dois tipos: τ e τ', gerando uma substituição θ, por meio do operador $\sim$. As quatro primeiras regras deste algoritmo seguem o padrão encontrado na literatura, com algumas pequenas alterações, propostas no trabalho de Gaster e Jones (1996). Essas mudanças visam retornos apenas de substituições mantendo os *kinds* dos tipos (*kind-preserving*).

Dentro da regra de união de *effect rows* $UNI-EFF$, a chamada da verificação $tl(\epsilon_1) \notin dom(\theta_1)$ busca garantir que a unificação tenha terminação. A operação $tl(\epsilon)$ é definida da seguinte forma:

$$tl(\langle \iota_1, ..., \iota_n \mid \mu \rangle) = \mu$$
$$tl(\langle \iota_1, ..., \iota_n \rangle) \quad = \langle\rangle$$

As ultimas três regras correspondem às regras de equivalência, descritas na Figura 9. Este último grupo unifica uma *effect row* com uma *head* específica. Como exemplo, a expressão $\epsilon \simeq \iota \mid \epsilon' : \theta$ mostra que: para uma *effect row* ϵ combinada com a constante de efeito ι, retornará um *effect tail* ϵ' e uma substituição θ ($\theta\epsilon = \langle \theta\iota \mid \theta\epsilon' \rangle$).

Para garantir que programas construídos em Koka possam ter o status de bem tipados, a linguagem define uma semântica mais precisa. A Figura 14 representa uma sintaxe mais apurada para a inferência de tipos e efeitos.

Figura 14 – Definição - Sintaxe de expressões completa

e	::=	v	$(valores)$
	|	$e_1 e_2$	$(aplicação)$
	|	$hp\ \varphi.e$	$(heap\ binding)$
	|	$let\ x = e_1\ in\ e_2$	$(let\ binding)$
	|	$run\ e$	$(isolate)$
v	::=	$\lambda x.e$	$(função)$
	|	$catch\ e$	$(captura\ parcial\ de\ exceções)$
	|	b	$(valores\ básicos, sem\ vínculos)$
b	::=	x	$(variável)$
	|	c	$(constante)$
	|	fix	$(ponto\ fixo)$
	|	$throw$	$(emite\ uma\ exceção)$
	|	$catch$	$(captura\ de\ exceções)$
	|	r	$(variável\ de\ referência)$
	|	ref	$(referência\ nova)$
	|	$(!)$	$(dereference)$
	|	$(::=)$	$(atribuição)$
	|	$(r\ :=)$	$(atribuição\ parcial)$
w	::=	$b \mid throw\ v$	$(valor\ básico\ ou\ exceção)$
a	::=	$v \mid throw\ v \mid hp\ \varphi.v \mid hp\ \varphi.throw\ v$	$(answers)$
φ	::=	$\langle r_1 \mapsto v_1 \rangle ... \langle r_n \mapsto v_n \rangle$	$(heap\ bindings)$

Fonte: Leijen (2014)

Dentro dessa gramática, a sintaxe de expressões foi modificada separando os valores de expressões v. Foi definido como b todo valor que não é uma expressão. Os *heap bindings* foram acrescentados à sintaxe por meio da construção ($hp\ \varphi.e$). A

Figura 15 – Definição - Regras de tipos para expressões de *heap*

$$\frac{typeof(c) \;=\; \sigma}{\Gamma \vdash c : \sigma \mid \epsilon} \qquad \textbf{(CONST)}$$

$$\frac{\Gamma \vdash e : \tau \mid \epsilon}{\Gamma \vdash e : \tau \mid \langle\, \iota \mid \epsilon \,\rangle} \qquad \textbf{(EXTEND)}$$

$$\frac{\forall \langle r_i \mapsto v_i \rangle \in \Gamma, \overline{\varphi}_h \vdash v_i : \tau_i \mid \langle\rangle \qquad \Gamma, \overline{\varphi}_h \vdash e : \tau \mid \langle st\langle h \rangle \mid \epsilon \,\rangle}{\Gamma \vdash hp\ \varphi.e : \tau \mid \langle st\langle h \rangle \mid \epsilon \,\rangle} \qquad \textbf{(HEAP)}$$

Fonte: Leijen (2014)

aplicação parcial de captura de exceções ($catch\ e$), a atribuição ($v :=$), e constantes em geral (c) também foram incluídas. Variáveis utilizadas em *heaps* recebem o indicador r. Uma função $hp\langle r_1 \mapsto v_1 \rangle, ..., \langle r_n \mapsto v_n \rangle.e$, vincula r_1 até r_n em $v_1, ..., v_n$ e em e.

A construção $hp\ \varphi.e$ nunca será exposta diretamente para o desenvolvedor, de modo que, durante a avaliação das reduções, será criado e utilizado um *heap*. Seu objetivo é o de dar um tipo para cada expressão, sendo necessária uma regra específica para tratar disso. A regra extra será aplicada nas ações dos *heap bindings*. A regra denominada $HEAP$ e mais duas regras necessárias para o funcionamento do sistema, estão definidas na Figura 15.

Cada valor de *heap* deve ter seu tipo avaliado utilizando, como premissa, um contexto de tipos para cada um dos vínculos criados. Um *heap binding* leva sempre a um efeito *statefull* $st\langle h \rangle$. A regra $EXTEND$, também exposta na Figura 15, garante a atribuição de um efeito, sendo necessária para se definir computações de estado. Por último, a regra $CONST$, aplicada em constantes, tem a função $typeof(c)$ e o objetivo de retornar um tipo fechado σ para cada constante. Com isso, pode-se dar, para qualquer expressão de valor, um efeito de tipo, inclusive um efeito vazio (LEIJEN, 2014).

Assim como a maioria das linguagens funcionais, Koka é baseada em cálculo lambda (*lambda calculus*) utilizando, também, o conceito de reduções. Define-se, então, algumas reduções primitivas (Figura 16) a serem aplicadas a expressões da linguagem. As quatro primeiras regras seguem o padrão do cálculo lambda proposto no trabalho de Church (1941).

A Figura 16 mostra as regras de redução, onde se utiliza δ para representar a função que recebe constantes e funções de valores fechados para valores fechados, separando-os de outros conjuntos de constantes. Leijen (2014) propõe a atribuição de um δ-*typability* para cada constante, de modo que: se $typeof(c) = \forall\overline{\alpha}.\tau_1 \rightarrow \epsilon\tau_2$, usa a substituição $\theta = [\overline{\alpha} \mapsto \overline{\tau}]$ e $\vdash v : \theta\ \tau_1 \mid \langle\ \rangle$, então retorna-se $\delta(c, v)$ e $\vdash \delta(c, v) : \theta_2 \mid \theta\epsilon$. Na redução β, e na regra LET, ocorre a substituição da variável x pelo valor inferido de

Figura 16 – Definição - Reduções e contextos de avaliação

(δ)	$c\ v$	$\longrightarrow$	$\delta\ (c, v)$ if $\delta\ (c, v)$ is def.
(β)	$(\lambda x.e)\ v$	$\longrightarrow$	$[x \mapsto v]e$
(LET)	let x = v in e	$\longrightarrow$	$[x \mapsto v]e$
(FIX)	fix v	$\longrightarrow$	$v\ (\lambda x.$(fix v) $x)$
(THROW)	X [throw v]	$\longrightarrow$	throw v if $X \neq$ []
(CATCHT)	catch(throw v)	$\longrightarrow$	$e\ v$
(CATCHV)	catch $v\ e$	$\longrightarrow$	v
(ALLOC)	ref v	$\longrightarrow$	hp $\langle\ r\ \mapsto v \rangle$. r
(READ)	hp $\varphi\ \langle\ r\ \mapsto v\ \rangle$. R[!r]	$\longrightarrow$	hp $\varphi \langle\ r\ \mapsto v\ \rangle$. $R[v]$
(WRITE)	hp $\varphi\ \langle\ r\ \mapsto v_1 \rangle$. $R[r$:= $v_2]$	$\longrightarrow$	hp $\varphi \langle\ r\ \mapsto v_2 \rangle$. R[()]
(MERGE)	hp φ_1 . hp φ_2 . e	$\longrightarrow$	hp $\varphi_1\ \varphi_2$. e
(LIFT)	R[hp φ . e]	$\longrightarrow$	hp φ. $R[e]$ if $R \neq$ []
(RUNL)	run[hp φ.] $\lambda x.e$	$\longrightarrow$	$\lambda x.$run ([hp φ.] e)
(RUNH)	run[hp φ.] catch e	$\longrightarrow$	catch (run ([hp φ.] e)
(RUNC)	run[hp φ.] w	$\longrightarrow$	w if frv(w) $\cap$ dom(ϕ) = $\emptyset$

Contextos de Avaliação:

X	::=	[] | $X\ e$ | $v\ X$ | let x = X in e
R	::=	[] | $R\ e$ | $v\ R$ | let x = R in e | catch $R\ e$
E	::=	[] | $E\ e$ | $v\ E$ | let x = E in e | catch $E\ e$ | hp φ. E | run E

Fonte: Leijen (2014)

v dentro na expressão e. A regra FIX introduz a recursão, utilizando um combinador de ponto-fixo.

O termo $THROW$ gera uma exceção, com base em um contexto X, que será propagada até o manipulador de exceções ou bloco de estado mais próximo. Porém, há de se obedecer à restrição de que este contexto não pode abrigar as funções: `catch` $e_1 e_2$, `hp` $\varphi.e$ ou `run` e. $CATCHT$ captura uma exceção e envia para o manipulador e caso ocorra uma exceção, esta deverá ser replicada para o próximo manipulador de exceções. A regra $CATCHV$ recebe como parâmetros um valor v (captura parcial de erros) e uma expressão e, retornando apenas a captura parcial v.

Leijen (2014) define as cinco regras seguintes para operar sobre reduções de *heaps*. $ALLOC$ cria um *heap*, $WRITE$ e $READ$ efetuam escrita e leitura de valores em um *heap*. $LIFT$ e $MERGE$ permitem a transformação de *heaps* em expressões, mantendo suas referências, podendo ser ligadas ao *heap* mais próximo. O isolamento de estado, durante a execução de uma função, é dado pelas três funções finais. $RUNL$ e $RUNC$ reduzem uma operação de execução em uma expressão lambda ou uma captura parcial de exceção. Já a regra $RUNH$ efetua a captura do isolamento de estado, reduzindo-o a um novo valor ou exceção e removendo o *heap* φ. A condição

$\cap dom(\phi) = \emptyset$ garante que durante redução não se perca a referência de seu vínculo. A partir das regras de redução, utilizando o contexto de avaliação E (Figura 16), pode-se definir uma função para avaliação de expressões:

$$E[e] \longmapsto E[e'] \quad iff \quad e \longrightarrow e'$$

A partir desse contexto de avaliação, somente a redução *leftmost-outermost* pode ser aplicada em uma expressão. No trabalho de Leijen (2014) encontra-se exemplos de implementações, de médio porte, utilizando a linguagem de programação Koka com sucesso. O sistema de efeitos da linguagem tem se mostrado seguro e tem facilitado o desenvolvimento de códigos com menor incidência de erros. Também são apresentadas neste mesmo trabalho, as provas das regras e sistemas descritos até aqui, bem como provas externas, utilizadas para a construção do sistema de efeitos.

2.4 REPRESENTAÇÕES INTERMEDIÁRIAS DE CÓDIGO

Compiladores para linguagens de programação têm evoluído, no sentido de dar mais eficiência, otimização, segurança e menor incidência de erros, no desenvolvimento de programas. Segundo Muchnick (1997), o funcionamento de um compilador é composto de um conjunto de etapas, onde cada uma destas fases tem uma missão específica. Ao se efetuar a mudança de uma etapa para a etapa seguinte, ao se compilar um código, pode ser necessária uma representação intermediária (IR – *intermediate representation*).

Uma IR é criada na transição entre cada fase no processo de compilação. Contidas nela, estão informações necessárias para o andamento da tradução de um código. Existem vários modelos de IR, e cada um desses pode ser aplicado em uma parte específica do procedimento. Essas representações podem ser: gráficas, lineares ou híbridas, cabendo ao projetista do compilador, escolher qual é a mais eficiente para a sua necessidade (COOPER; TORCZON, 2011).

IRs gráficas estabelecem grafos, por intermédio de informações advindas da etapa na qual o processo de compilação se encontra. As árvores sintáticas, geradas durante a análise sintática, são exemplos destas IRs. Enquanto as representações lineares se assemelham a pseudocódigos para máquinas abstratas. Esta codificação executa operações lineares em sequência; o código de três endereços, é uma destas representações.

Como sua denominação sugere, uma IR Híbrida combina atributos das IRs Gráficas e Lineares. Cooper e Torczon (2011) expõem que esse tipo de representação, tenta explorar os pontos fortes de cada um destes modelos, utilizando uma IR Linear

de baixo nível, para representar blocos de códigos em linha, e uma representação do fluxo de controle destes blocos, em forma de grafo.

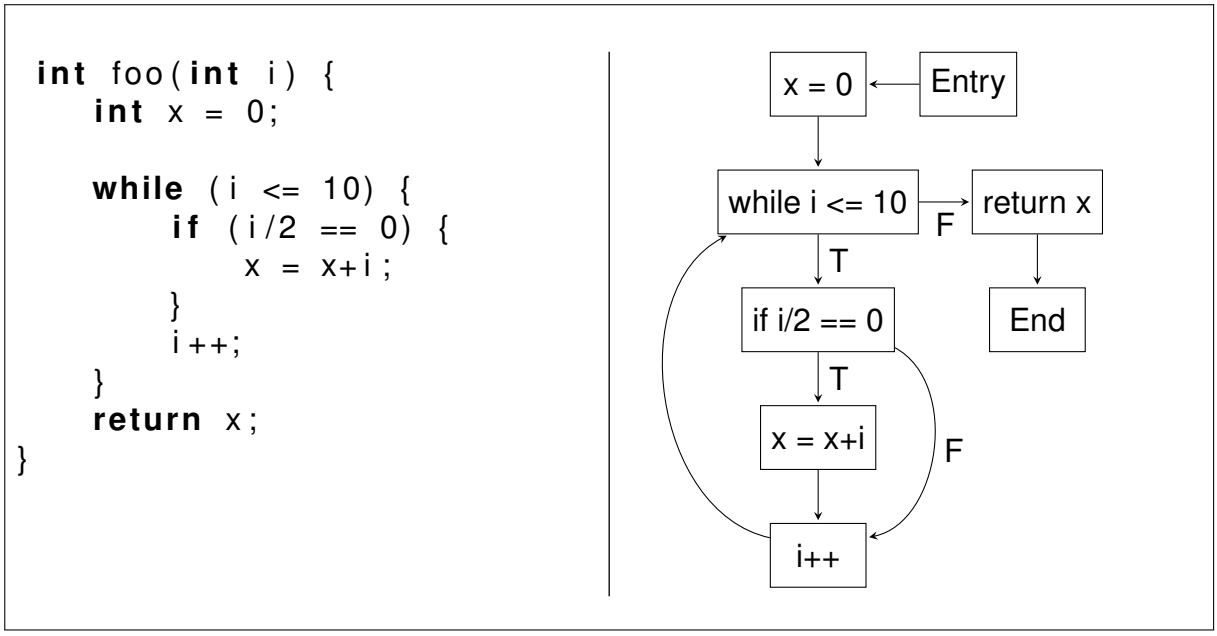
Em linguagens de programação modernas, IRs híbridas são as mais utilizadas, pois podem permitir um maior número de otimizações, além de possibilitar uma maior eficiência. Os grafos de fluxo de controle (CFG — *Control-Flow Graph*) correspondem a este grupo de IRs híbridas.

2.4.1 Grafos de Fluxo de Controle

Um CFG pode ser entendido como um modelo que representa todos os caminhos possíveis, de um código, durante o processo de tradução de um programa. Formalmente, Cooper e Torczon (2011) definem CFG como uma IR que manipula o fluxo de controle entre os blocos básicos de um programa. Sendo este um grafo dirigido $G = (N, E)$, em que cada nó $n \in N$ representa um bloco básico, e cada aresta $e = (n_i, n_j) \in E$ corresponde a uma transferência de controle, de um bloco n_i para um bloco n_j.

Para esta representação gráfica, exige-se que cada grafo apresente somente um nó de entrada. A Figura 17 apresenta um pequeno código na linguagem C utilizando condições `while` e `if` e o seu CFG correspondente.

Figura 17 – Exemplo - Conversão para CFG



Fonte: O Autor

Um bloco básico é composto por uma sequência de comandos consecutivos. Cada um destes blocos pode ter uma operação qualquer e encaminhar fluxo para o bloco seguinte, ou efetuar a verificação de uma condição. Dependendo do resultado

desta consulta, o fluxo poderá seguir um ou outro caminho, além disso, um bloco pode receber informações advindas de mais de um bloco. Cada linguagem de programação utiliza essa representação de forma diferente e, normalmente, está associada a outro tipo de representação intermediária. Um exemplo de IR que pode ser utilizada em conjunto com CFG, é a forma de atribuição única estática (SSA – *Single Static Assingment*).

2.4.2 Forma de Atribuição Única Estática

A forma SSA consiste em um modo de IR onde se atribui um denominador para um valor de variável, impondo que este nome seja único dentro do programa. Para Muchnick (1997), esta forma de representação permite muitas otimizações de código, sendo alcançadas com a separação dos valores contidos em um programa, e usando como base, os locais em que estes estão localizados.

Cooper e Torczon (2011) definem que, para estar na forma SSA, um programa deve obedecer a restrição de ter apenas um nome exclusivo para cada definição e cada ação deve ser ligada a uma única definição. Em programas onde não ocorram condições de desvios (*if* por exemplo), a conversão para SSA é trivial. Cada uma das instruções atribuirá, sempre, um novo nome de variável. Ao se definir a variável x, no escopo do programa, esta será renomeada como x_1 e a cada uso desta variável, será modificada afim de se utilizar a última definição desta. A Figura 19, a seguir, demonstra a conversão simples de atribuições para a forma SSA.

Figura 18 – Exemplo - Conversão de atribuições para SSA

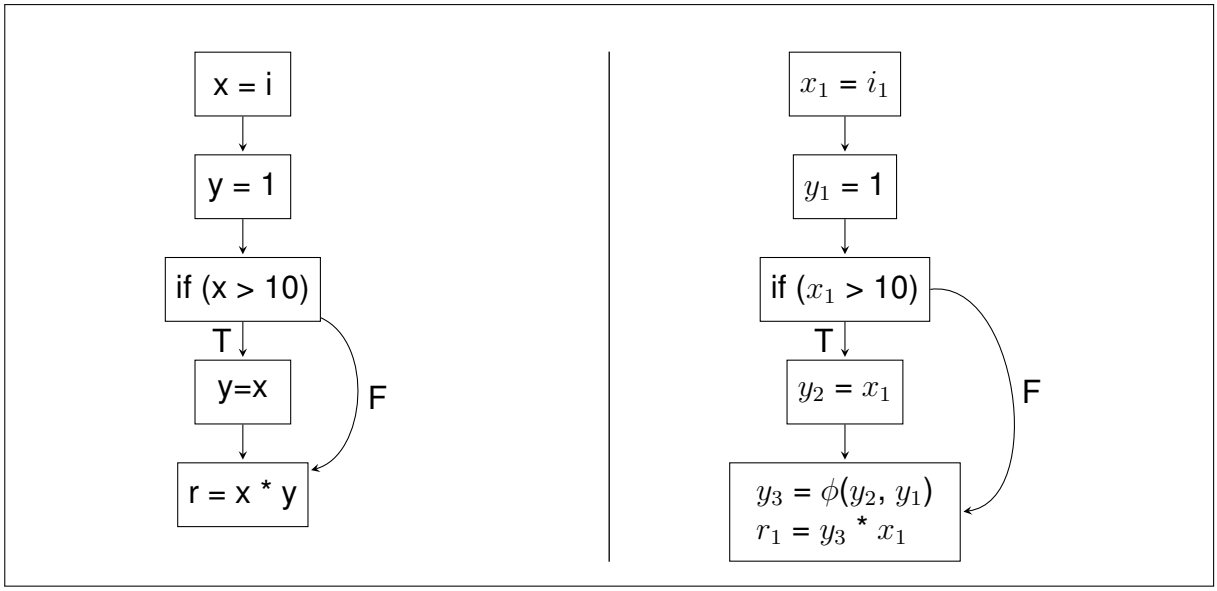
a = 10	$\longrightarrow$	a_1 = 10
b = 15	$\longrightarrow$	b_1 = 15
c = a − b	$\longrightarrow$	$c_1 = a_1 - b_1$
a = b * 15	$\longrightarrow$	$a_2 = b_1$ * 15
b = c + 1	$\longrightarrow$	$b_2 = c_1$ + 1
c = a + 3	$\longrightarrow$	$c_2 = a_2$ + 3

Fonte: O Autor

Na prática, programas fazem uso de mais do que apenas atribuições e cálculos simples, tendo de haver a tomada de decisão por qual caminho o fluxo do código deve seguir. Um CFG, abordado na Seção 2.4.1, é utilizado como IR em linguagens de programação, pois permite trabalhar em cima de blocos que contenham condições. Para isso, ao se converter um grafo destes para a forma SSA deve-se acrescentar funções ϕ em locais onde os fluxos de controle se encontram. Após o encontro destes caminhos, é efetuada a renomeação de variáveis, buscando manter a propriedade de um único nome para cada valor.

Utilizando uma função ϕ pode-se agrupar as variáveis oriundas de dois caminhos distintos, durante o processo de funcionamento do programa. A Figura 19 apresenta o CFG de um programa e sua transformação para a forma SSA. Os blocos básicos podem receber, além da informação referente a ação que está sendo executada, os novos nomes exclusivos das variáveis.

Figura 19 – Exemplo - Conversão de CFG para SSA



Fonte: O Autor

O grafo com a nomeação estática para as variáveis, à direita, segue o processo normal de um CFG. Porém, no último bloco básico, pode-se encontrar uma informação a mais, gerada de acordo com o caminho que foi percorrido, até chegar ao final da representação. De forma que, no local que consta a função ϕ, será atribuído o valor para y_3. Caso este parâmetro satisfaça condição if, receberá o valor de y_2, do contrário, reterá y_1. Assim, será computado o valor final desse bloco atribuído a r_1.

A função ϕ pode ser pensada como um ponto do programa onde se faz a bifurcação de caminhos. De acordo com o valor inserido como entrada para o programa, ou para o bloco em questão, será escolhido um dos parâmetros contidos na função. Esse argumento representará qual caminho deverá ser seguido na continuidade da aplicação. Porém, em sua representação, deve conter todos os caminhos que podem ser seguidos durante a execução de um código.

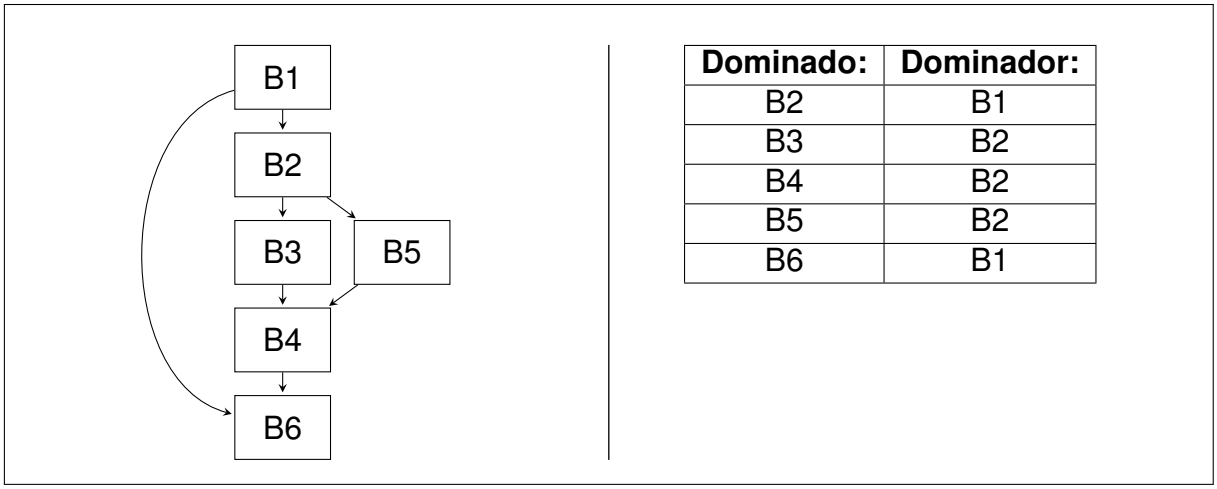
Em programas de médio porte, com muitas condições em seu funcionamento, pode-se gerar um número grande de funções ϕ e também de nomes únicos para variáveis. Segundo Muchnick (1997), para se traduzir um programa para uma forma SSA mais eficiente, com menor número de funções ϕ, pode-se utilizar o conceito de dominâncias (*dominance frontiers*). Dado um nó x de um CFG, a dominância de x,

escrita $DF(x)$ é um conjunto de todos os nós y no grafo, de modo que x domina um predecessor intermediário ($pred$) de y, mas não domina estritamente y. Isso pode ser escrito da seguinte forma:

$$DF(x) = [y|(\exists_z \ \in \ pred(y) \text{ tal que } x \ dom \ z) \text{ e } x \ !sdom \ y]$$

O domínio estrito, neste caso representado pela expressão $x \ !sdom \ y$, diz que, x domina estritamente y se x dominar y e que ambos são diferentes ($x \neq y$). A dominância de um bloco é a coleção de todos os blocos que sucedem imediadamente os blocos dominados pelo bloco dominador. Isso deve ser executado atendendo a restrição de dominância estrita. De forma simplificada, dominadores podem ser entendidos como uma forma de se representar quais blocos básicos devem ser executados antes dos demais blocos. A Figura 20 apresenta um CFG genérico e sua tabela de dominância correspondente.

Figura 20 – Exemplo - Tabela de Dominância de CFG



Dominado:	Dominador:
B2	B1
B3	B2
B4	B2
B5	B2
B6	B1

Fonte: O Autor

Com essa relação de dominância estabelecida pode-se, então, elaborar algoritmos eficientes para a otimização da transformação de IR em um SSA *minimal* utilizando CFG. Na literatura pode-se encontrar implementações eficientes de algoritmos, utilizando *dominance frontiers* para a conversão para SSA, como no trabalho de Cytron et al. (1991). Pode-se utilizar o SSA, juntamente das informações de dominância, para a aplicação de transformações de códigos.

2.4.3 Conversão de SSA para Representação Funcional

Conforme anteriormente abordado, representações intermediárias são amplamente utilizadas em otimizações e no funcionamento do processo de compilação. Segundo Appel (1998) desde os anos 80 a composição SSA/CFG é utilizada em lingua-

gens imperativas. Um dos mais famosos compiladores da linguagem imperativa C, o GCC (STALLMAN, 2005), faz uso de SSA/CFG no processo de tradução de códigos.

Compiladores de linguagens imperativas e funcionais operam de forma diferente em diversos aspectos. Esse cenário pode ser notado em diversas etapas do processo de tradução de código, como na geração de IRs e na avaliação de argumentos. Embora existam essas discrepâncias, Appel (1998) cita que existe uma relação direta entre o cálculo lambda, utilizado por linguagens funcionais, e a forma de representação intermediária SSA, utilizada em linguagens imperativas. Isso remete ao fato de que pode-se interpretar o CFG, com o seu SSA, para uma representação funcional em forma de cálculo lambda. Ele cita ainda que cada função phi encontrada na forma SSA é equivalente a uma função λ no cálculo funcional. Em programas que não fazem uso de condicionais, essa tradução é simples, porém em códigos mais rebuscados, o processo se torna mais complexo.

O trabalho de Torrens, Vasconcellos e Gonçalves (2017) sugere uma linguagem intermediaria que pode ser interpretada, simultaneamente, como SSA ou uma representação funcional. Essa linguagem permite a aplicação de otimizações e procedimentos de garantias de ambos os paradigmas.

Para a geração desta linguagem intermediária, denominada λ^{U}_{SSA} sugere-se a conversão da representação intermediária SSA/CFG em uma representação funcional. Segundo Appel (1998), a representação SSA/CFG pode ser vista como uma linguagem funcional. Isso pode permitir essa conversão sem maiores problemas, utilizando a relação de dominância. A sintaxe da linguagem λ^{U}_{SSA} é mostrada na Figura 21.

Figura 21 – Definição - Gramática λ^{U}_{SSA}

(toplevel function)	F	::=	$\texttt{func}(x*)\ B$
(block)	B	::=	$\texttt{let}\ x \Leftarrow E\ \texttt{in}\ B$
		|	$\texttt{let}\ x \Leftarrow y(z*)\ \texttt{in}\ B$
		|	$\texttt{goto}\ x_n\ D^?\ y(z*)$
		|	$\texttt{goto}\ x\ ?\ y_n : (z_m)\ D^?$
		|	$\texttt{halt}\ x$
(dominator tree)	D	::=	$\texttt{where}\ L^+$
(labeled block)	L	::=	$x\texttt{:}\ I^*\ B$
(ϕ-node)	I	::=	$\texttt{let}\ x \Leftarrow \phi(y+)\ \texttt{in}$
(expression)	E	::=	$n \mid x + y \mid x - y...$

Fonte: Torrens, Vasconcellos e Gonçalves (2017)

Nesta sintaxe, segundo Torrens, Vasconcellos e Gonçalves (2017), identificadores são representados pelas variáveis x, y e z. Constantes numéricas são representadas por n e m. Os termos sobre-escritos correspondem a zero ou mais ($*$), um ou

mais ($+$) e zero ou um ($?$). Blocos são as principais estruturas de controle, contendo definições de variáveis, sendo sucedidas por uma chamada. A construção $\texttt{func}(x^*)$, define uma função de nível superior, utilizada para otimizações e interpretada como uma função definida no contexto global do código de origem, como a definição de uma variável global.

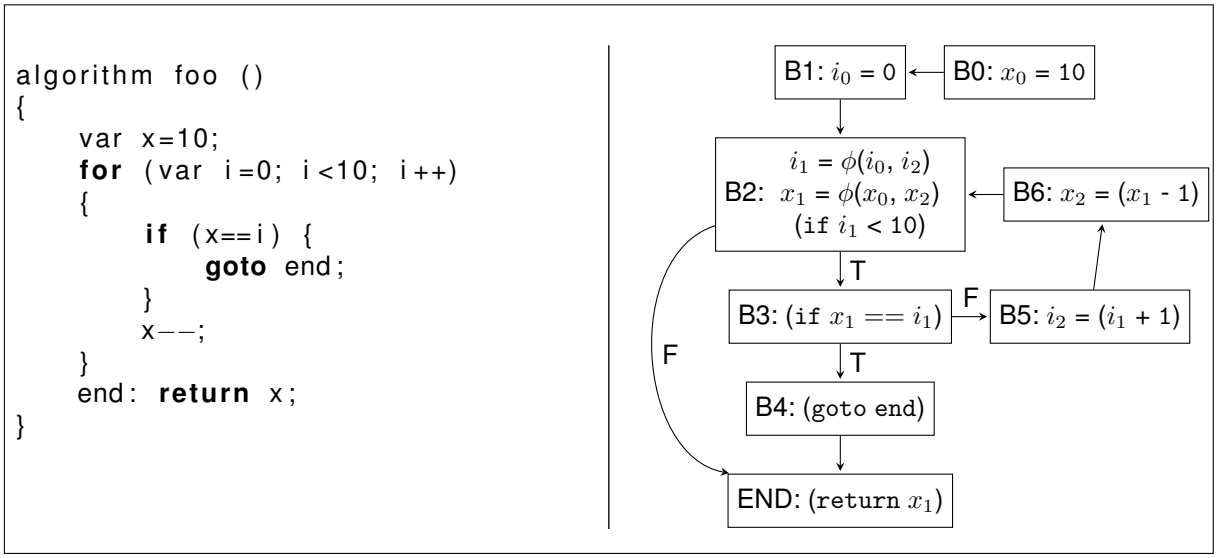
Os termos $\texttt{let}\ x \Leftarrow E\ \texttt{in}\ B$ fixam uma expressão E no bloco B. A produção $\texttt{let}\ x \Leftarrow y(z*)\ \texttt{in}\ B$ é utilizada na chamada de outra função de nível superior. A construção $\texttt{goto}\ x_n\ D^?\ y(z*)$ chama, de forma incondicional, o bloco rotulado x com o conjunto dos atuais parâmetros n. D representará a relação de dominância entre blocos.

Já, $\texttt{goto}\ x\ ?\ y_n : (z_m)\ D^?$ chama, condicionalmente, um bloco rotulado y com um conjunto de parâmetros n, ou um bloco rotulado z com um conjunto de parâmetros m, de acordo com o valor de x (verdadeiro ou falso). Por último, $\texttt{halt}\ x$ faz a chamada da atual continuação de retorno com a variável vinculada x.

Ainda no trabalho de Torrens, Vasconcellos e Gonçalves (2017), pode-se encontrar um exemplo prático desta conversão, bem como maiores informações a respeito da linguagem intermediária proposta. Esta conversão, na prática, não apresenta muitos problemas, permitindo a geração de um código funcional semelhante à linguagem Haskell.

Para executar a transformação da forma SSA para uma representação funcional, primeiramente é necessária a conversão do código de um programa para um CFG. Nos blocos básicos deste grafo efetua-se a renomeação de variáveis e a inserção das funções ϕ, nos locais de encontro de fluxos.

Figura 22 – Exemplo - Algoritmo e sua Representação em SSA



Fonte: O Autor

A Figura 22 exibe um código, contendo elementos comuns em programas, e sua representação na forma de SSA/CFG. Pode-se observar, no bloco $B2$, o encontro de fluxos originários de $B1$ e $B6$. Neste local será inserida uma função ϕ que recebe como parâmetros, os valores contidos nas variáveis i_0 (oriundo de $B1$) e i_2 (recebido de $B5$, utilizando $B6$ como caminho).

Para se efetuar a conversão da forma SSA para uma representação funcional, são necessárias informações obtidas por intermédio da relação de dominância. A Tabela 2 apresenta a relação de dominadores e dominados, do algoritmo exposto na Figura 22.

Tabela 2 – Tabela de Dominância

Dominado:	**Dominador:**
B0	–
B1	B0
B2	B1
B3	B2
B4	B3
B5	B3
B6	B5
END	B2

Fonte: O Autor

O bloco básico $B2$ domina os blocos $B3$ e END, isso quer dizer que, para o fluxo acessar estes dois blocos, primeiro deve-se passar por $B2$. O mesmo ocorre com os blocos que têm como dominador o bloco $B3$, estes somente serão atingidos por informações vindas de $B3$. Essas informações são importantes no processo de conversão, pois, a partir delas, pode-se visualizar e rotular blocos alinhados mais facilmente. A Figura 23 apresenta o código convertido para uma representação funcional.

Nesta representação funcional, proposta no trabalho de (APPEL, 1998) cada bloco vira uma `function`. A variável `x0` recebe 10 e é propagada para o restante da função. O mesmo ocorre com todas as chamadas de `let`, onde se recebe uma variável ou uma `function`. Esse código, se forem removidas as *keywords* `function`, é válido para Haskell.

A informação de dominância, além de auxiliar a transformação do SSA/CFG para uma representação funcional, permite a aplicação de algoritmos de otimizações, como citado na Subseção 2.4.2. Essas otimizações visam minimizar o número de funções ϕ durante a conversão da linguagem objeto para a representação SSA/CFG . Quanto menor for o número de funções ϕ, mais eficiente será a tradução.

A partir deste código funcional, otimizações e processos de inferências e garantias de tipo, comuns em linguagens funcionais, podem ser aplicados. Isto, somado a otimizações de compiladores para linguagens imperativas aplicadas durante a trans-

Figura 23 – Exemplo - Representação funcional

```
1  let x0 = 10 in
2  let function b1() =
3      let i0 = 0 in
4      let function b2(i1, x1) =
5            let function end() =
6                  x1
7                function b3() =
8                  let function b4() =
9                        end()
10                     function b5() =
11                       let i2 = i1 + 1 in
12                       let function b6() =
13                             let x2 = x1 - 1 in
14                             b2(i2, x2)
15                       in b6()
16                 in if x1 == i1 then
17                   b4()
18                 else
19                   b5()
20           in if i1 < 10 then
21             in b3()
22           else
23             end()
24     in b2(i0, x0)
25 in b1()
```

Fonte: O Autor

formação do código fonte para SSA, podem garantir a geração de programas mais eficientes e com menor incidência de erros.

Com base no que foi levantado na fundamentação teórica, esse trabalho irá explorar, nos capítulos seguintes, a correspondência entre SSA/CFG e o cálculo lambda. Aplicando, após concluída a tradução, o algoritimo de tipos e efeitos proposto na linguagem Koka.

3 DESENVOLVIMENTO

Para se dar início à construção do projeto foi definida a sintaxe de uma linguagem de programação de propósito geral. O corpo da gramática da linguagem fonte é baseada em conceitos básicos de linguagens de programação imperativas. A sintaxe adotada para a representação dos manipuladores e construtores de efeito foi adaptada ao dialeto para uma melhor leitura e utilização. Há também a necessidade de se salientar que, neste momento, as funções dos programas criados não são valores de primeira classe, ou seja, funções não podem ser utilizadas como parâmetros e nem retornadas em computações. Nesse capítulo serão abordados os detalhes de implementação dos algorítimos utilizados na construção do compilador e, ao final, na Seção 3.4, serão exibidos os resultados preliminares da presente pesquisa.

3.1 DEFINIÇÃO DA GRAMÁTICA PROPOSTA

As principais construções sintáticas foram definidas de forma que fiquem familiares à programadores habituados a linguagens imperativas. Escolheu-se o padrão da linguagem C como inspiração por representar bem o paradigma imperativo e apresenta uma sintaxe simples e fácil de ser entendida. Além de ser uma linguagem utilizada no ensino básico de programação.

A gramática criada está disposta utilizando o formalismo de *Backus-Naur* (BNF). Para simplificar a especificação, foram acrescentados os símbolos: de `"+"` (um ou mais parâmetros), `"*"` (zero ou mais parâmetros) e `"?"` (um ou nenhum parâmetro). O núcleo básico da linguagem é apresentado na Figura 24. Nesta definição pode-se observar que um programa escrito com essa sintaxe será dado pela declaração de $\langle top - level \rangle$ podendo ser um $\langle handler \rangle$, um $\langle algorithm \rangle$ ou um $\langle effect \rangle$. A definição completa da gramática da linguagem proposta encontra-se no Apêndice A.

Um $\langle algorithm \rangle$ é descrito pela utilização da palavra reservada `algorithm`, seguido pelo terminal $\langle id \rangle$, representando o nome da função, uma lista contendo nenhum ou mais parâmetros ($\langle param - list \rangle$), e de um $\langle body \rangle$. Esses parâmetros podem ser declarados com os tipos pré-definidos na linguagem (`int` `bool` e `string`) ou pela construção `var` e do identificador da variável. A palavra reservada `var` indica que o tipo do argumento será inferido pelo compilador da linguagem. O corpo da função $\langle body \rangle$ é definido por uma sequência de *statements* entre chaves. Diferentemente de linguagens funcionais como Haskell, a linguagem aqui definida não faz uso de expressões em seu corpo. Os *statements* definidos são similares aos utilizados pelo padrão da linguagem C, sendo removidas algumas regras para simplificar a sintaxe.

Figura 24 – Definição - Trecho da gramática linguagem proposta

⟨*program*⟩ ::= ⟨*top-level*⟩+

⟨*top-level*⟩ ::= ⟨*algorithm*⟩ | ⟨*effect*⟩ | ⟨*handler*⟩

⟨*algorithm*⟩ ::= ‘`algorithm`’ ⟨*id*⟩ ⟨*param-list*⟩ ⟨*body*⟩

⟨*param-list*⟩ ::= ‘`(`’ ⟨*declarator*⟩* ‘`)`’

⟨*declarator*⟩ ::= ‘`var`’ ⟨*id*⟩ | ⟨*type*⟩ ⟨*id*⟩

⟨*type*⟩ ::= ‘`int`’ | ‘`bool`’ | ‘`string`’

⟨*body*⟩ ::= ‘`{`’ ⟨*statement*⟩+ ‘`}`’

⟨*statement*⟩ ::= ⟨*assignment-stmt*⟩
| ⟨*call-stmt*⟩
| ⟨*if-stmt*⟩
| ⟨*while-stmt*⟩
| ⟨*for-stmt*⟩
| ⟨*goto-stmt*⟩
| ⟨*return-stmt*⟩
| ⟨*label-stmt*⟩

⟨*effect*⟩ ::= ‘`effect`’ ⟨*id*⟩ ‘`{`’ ⟨*effect-decl*⟩* ‘`}`’

⟨*effect-decl*⟩ ::= ‘`function`’ ⟨*id*⟩ ‘`(`’ ⟨*type*⟩+ ‘`)`’ ⟨*effect-return*⟩?

⟨*effect-return*⟩ ::= ‘`:`’ ⟨*type*⟩

⟨*handler*⟩ ::= ‘`handler`’ ⟨*id*⟩ ‘`(`’ ‘`)`’ ‘`{`’ ⟨*handler-body*⟩+ ‘`}`’

⟨*handler-body*⟩ ::= ‘`pure`’ ‘`:`’ ⟨*body*⟩ | ‘`case`’ ⟨*id*⟩ ‘`(`’ ‘`)`’ ‘`:`’ ⟨*body*⟩

Fonte: O Autor

A produção $\langle effect \rangle$ representa o construtor de efeitos. Esse comando é realizado por meio da declaração de *token* `effect`, da definição de seu identificador ($\langle id \rangle$) e de uma ou mais declarações de efeito ($\langle effect-decl \rangle$). Essa declaração de efeito é obtida pela chamada de `function`, terminal de identificação ($\langle id \rangle$), uma ou mais declarações de tipos ($\langle type \rangle$) e de zero ou mais retornos de efeito ($\langle effect-return \rangle$). Para se utilizar o $\langle effect-return \rangle$, é necessário o operador $\langle : \rangle$ e uma declaração de tipos ($\langle type \rangle$). Esse mecanismo permite ao programador definir efeitos customizados a serem utilizados em seu programa.

A declaração $\langle handler \rangle$ remete aos manipuladores de efeitos colaterais definidos para a linguagem. Após `handler` é definido o identificador do manipulador e da definição, entre chaves, de um ou mais $\langle handler-body \rangle$. Um $\langle handler-body \rangle$ pode ser definido por `pure` e por um $\langle body \rangle$ ou pela construção `case` e por um $\langle id \rangle$, `():` e de um $\langle body \rangle$. Como citado anteriormente, um $\langle body \rangle$ é composto por uma sequencia de *statements*. Caso a cláusula $\langle pure \rangle$, que recebe o resultado final da computação, não seja informada, uma função de identidade é assumida. Simplesmente retornando o resultado da computação inalterado (como implementado em Koka).

O corpo das funções também é formado por uma sequencia de *statments* ($\langle body \rangle$). Como mencionado anteriormente esses *statements* são similares aos encontrados em linguagens imperativas. A Figura 25 exibe a gramática da chamada da estrutura de controle *if*.

Figura 25 – Definição - $\langle if-stmt \rangle$

⟨*if-stmt*⟩ ::= ⟨*if*⟩ ⟨*else*⟩?

⟨*if*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*body*⟩
| ‘`if`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*statement*⟩

⟨*else*⟩ ::= ‘`else`’ ⟨*body*⟩ | ‘`else`’ ⟨*statment*⟩

Fonte: O Autor

O *statement* $\langle if-stmt \rangle$ segue o padrão da linguagem C, recebendo um $\langle if \rangle$ e um ou nenhum $\langle else \rangle$. A regra $\langle if \rangle$ é composta de `if` uma expressão ($\langle expr \rangle$) entre parênteses e de um $\langle body \rangle$ ou por um $\langle statement \rangle$. A produção iniciada por $\langle else \rangle$, tem o uso do *token* `else` seguido por um $\langle body \rangle$ ou por um $\langle statement \rangle$. Uma $\langle expr \rangle$ nessa gramática é dada por expressões matemáticas em geral incluindo operadores relacionais e lógicos. A Figura 26 apresenta um código simples da estrutura de controle if, um código equivalente escrito na linguagem proposta neste trabalho e por último, a mesma sentença codificada em Haskell.

Pode-se notar que os códigos escritos em C e na linguagem aqui proposta são semelhantes em suas construções. Mas estas semelhanças são apenas sintáticas, pois a representação intermediária gerada a partir dessa sintaxe é puramente funcional como Haskell. Devido a isso, e por não ter o corpo das funções construído através de expressões, adotou-se o termo *pseudoimperativa* para se referir ao dialeto aqui proposto. Outro detalhe a se salientar é que, se comparada a Haskell, a linguagem aqui desenvolvida tende a ser mais legível para um programador imperativo, mesmo podendo ser traduzida para um código puramente funcional. A partir desta gramática,

Figura 26 – Exemplo - Estruturas de controle if

```c
#include <stdlib.h>                          // Exemplo da estrutura de controle if em C.
#include <stdio.h>

int main(void) {
    int a;
    scanf("%d", &a);

    if(a == 10) {
        printf("Hello, earth!");
    } else {
        printf("Hello, mars!");
    }

    return EXIT_SUCCESS;
}
```

```
algorithm main() {                           // Exemplo da estrutura de controle if na linguagem
    var a;                                   // proposta neste trabalho.
    scan(a);

    if(*a == 10) {
        print("Hello, earth!");
    } else {
        print("Hello, mars!");
    }
}
```

```haskell
main :: forall s. STT s IO ()                // Exemplo da estrutura de controle if em Haskell.
main =
    a <- newSTRef
    liftIO $ scan a
    if (a == 10)
        then do
            liftIO $ print "Hello, earth!"
        else do
            liftIO $ print "Hello, mars!"
```

Fonte: O Autor

é possível foi construído o compilador.

3.2 PROCESSO DE COMPILAÇÃO DE CÓDIGO

A linguagem escolhida para a construção do compilador foi Haskell pois é uma linguagem puramente funcional com boa documentação. Além disso, ela dispõe de ferramentas auxiliares para se fazer as análises léxica e sintática. Essas análises

são base de todo o processo de compilação, compondo, juntamente com a análise semântica, o *front-end* do compilador.

3.2.1 Análises Léxica e Sintática

A análise léxica é a primeira parte do processo de tradução de um código. Dentro dessa verificação é apurado se os caracteres informados são válidos para a gramática definida. O analisador léxico também é responsável pela conversão dos caracteres em uma sequências de *tokens*. O processo de análise léxica, atualmente, pode ser construído pelo uso de ferramentas que necessitam apenas da definição dos *tokens* de uma linguagem. A ferramenta Alex (DORNAN; JONES, 2003), permite a definição de tais palavras, e também a verificação das mesmas tornando mais eficiente o desenvolvimento do compilador. Outra vantagem do uso dessa ferramenta é que ela dispõe de alternativas para a geração de notificações de erros durante o processo de compilação. Portanto, utilizou-se a ferramenta Alex para gerar o analisador léxico na construção do compilador.

Em seguida é feita a análise sintática que avalia se a expressão passada a ela, pelo analisador léxico, é uma expressão válida na sintaxe proposta. A construção de analisadores sintáticos pode ser muito custosa e gerar muitos problemas. Assim como os analisadores léxicos, os analisadores sintáticos podem ser construídos apenas com a definição da gramática proposta. O código validado pela ferramenta, retornará a árvore sintática abstrata (AST) gerada a partir da linguagem fonte. O analisador sintático Happy (MARLOW; GILL, 1997) provê mecanismos para a validação de árvores sintáticas e também para a definição das mesmas através dos construtores de tipos da linguagem Haskell. O Happy funciona em perfeita harmonia com o Alex, fato que ajudou a evitar problemas e economizou tempo na definição e validação da primeira parte do processo de compilação. Essas ferramentas apresentam maior rigor teórico se comparados à outros mecanismos para o mesmo fim.

A gramática definida na Seção 2.4 foi implementada e validada pelas ferramentas acima descritas. Além disso, como citado anteriormente, foi possível, por meio destas ferramentas, fornecer mecanismos para auxiliar no tratamento e indicação de erros na execução do compilador. A partir dessa validação é gerada uma árvore sintática abstrata. Esta árvore é uma representação que indica a estrutura semântica do código fonte. Após gerada, a AST pode ser transformada em outras representações intermediárias de código.

3.2.2 Análise Semântica

O objetivo da análise semântica é diminuir o conjunto de programas aceitos ao subconjunto sintaticamente válido que apresenta comportamento bem-definido pela

semântica da linguagem. Em geral, a analise semântica está relacionada apenas a verificação do contexto, por exemplo, verificação de tipos e escopo. Nesse projeto a análise semântica foi postergada, sendo executada apenas na representação intermediária funcional gerada nas fases descritas a seguir.

3.2.2.1 Geração dos Grafos de Fluxo de Controle

Com base na AST gerada, cria-se um grafo de fluxo de controle. Por definição, a transformação em CFG consiste na transcrição dos *statements* em blocos básicos. Cada bloco conterá informações diferentes de acordo com a sua localização no fluxo de controle. A geração dos blocos básicos, e posteriormente do CFG, segue o que é proposto na literatura. Um bloco básico contém uma instrução, podendo agregar, no caso das atribuições de variáveis, mais de uma instrução em seu corpo. Utilizando os construtores de tipos de Haskell, um bloco básico agregará informações e se tornará um nó do grafo. O grafo é definido por uma lista de nós que contém, além das informações do bloco básico, informações relativas a qual momento do fluxo este se encontra e também para quais outros blocos ele aponta.

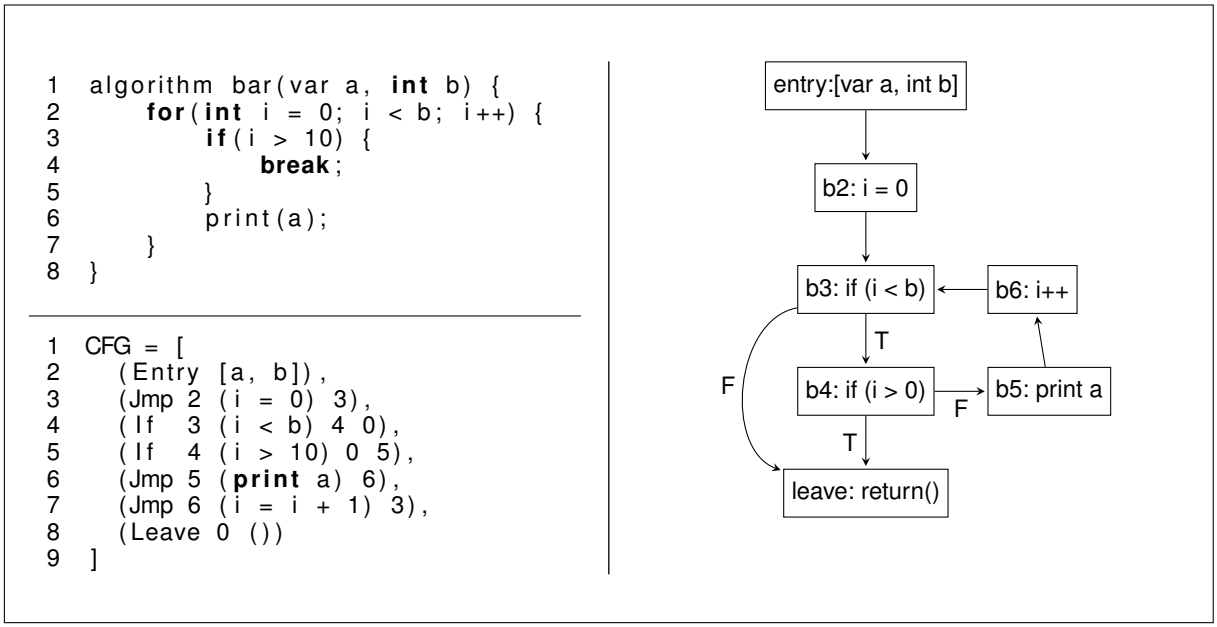
Os nós foram definidos através do construtor `Node`. Cada `Node` corresponde a um tipo de bloco básico, podendo também ser um nó ϕ (será apresentado adiante nessa mesma subseção). A representação dos blocos básicos, que serão os nós do grafo de fluxo de controle, foram definidos da seguinte forma:

```
data Node = Entry  Parameters Label
          | Jump   Label Command Label
          | IF     Label Expression Label Label
          | Leave  Label Expression
```

Cada nó, representado pela definição do tipo `Node`, será disposto em uma lista. O construtor `Entry` sempre marca o início do grafo, podendo receber `Parameters` (variáveis e seus tipos) e um valor `Label` que indica para onde este nó aponta. A alternativa iniciada com `Jump` representa um bloco básico comum sendo composto de seu endereço, representado pelo primeiro `Label`, o *statement* em si (`Command`) e o `Label` para qual este bloco aponta. A chamada de `If`, por sua vez, não executa nenhum comando, apenas faz a verificação da expressão e, conforme seu resultado, aponta para um ou outro endereço. O bloco que se iniciar pelo construtor `Leave` representará o retorno da computação, podendo delimitar o fim do grafo. Conterá apenas o seu endereço na forma de uma `Label` seguido de uma expressão da linguagem.

A Figura 27 exemplifica a tradução do código fonte para um grafo de fluxo de controle, a lista contendo os nós do grafo gerado e o desenho do grafo demonstrando o fluxo de controle do programa.

Figura 27 – Exemplo - Conversão de código para CFG



Fonte: O Autor

O código descrito Figura 27 representa um programa que recebe uma variável `a`, sem tipo definido, e um inteiro `b` que ao executar as operações contidas na instrução *for* irá imprimir na tela o valor de a. Logo abaixo, está a representação intermediária, em forma de lista, gerada utilizando o construtor `Node`, remetendo ao Grafo de Fluxo de controle. Ao lado direito encontra-se o grafo desenhado de forma a representar todos os caminhos percorrido pelo código.

Na lista dos nós criados, observa-se que o primeiro bloco básico representa o início do código com os parâmetros `var a` e `int b`. A estrutura de controle `for` foi dividida em 3 partes. A primeira delas é a atribuição do bloco 2, `Jump 2 (i = 0) 3`. Aqui é importante notar que o tipo de `i`, embora informado pelo programador, é desconsiderado. O tipo de `i` será inferido pelo compilador ainda na análise semântica, sua declaração serve apenas como anotação. A segunda parte do `for` é obtida pela construção IF 3 (`i < b`) 4 0, onde 4 representa a *label* do próximo bloco caso a condição seja verdadeira e 0 caso ela seja falsa. Seguindo o fluxo, o bloco representado por IF 4 (`i > 10`) 0 5, assim como seu predecessor, apresenta 2 caminhos, 0 se a condição for satisfeita e 5 caso ela seja falsa. O bloco Jmp 5 (`print a`) 6, irá executar a função de `print` tendo como parâmetro a variável `a` e enviará o fluxo para o bloco Jmp 6 (`i = i + 1`) 3. A terceira e última parte do `for`, o operador de incremeto, irá enviar o fluxo de volta para o bloco 3, que poderá apontar, se a condição não for atendida, para o bloco `Leave 0` () que representa o final do código. Este bloco terá como rótulo o número 0 denotando um `return` implícito no grafo. Este bloco também poderá ser alcançado a partir do bloco 4, caso o *if* satisfaça a condição e caia no `break`.

Se houver um `return` explícito dentro do código, este bloco básico irá receber uma `Label` própria. Durante a geração dos nós do grafo, assumiu-se que sempre haverá um *return* implício no fim da função mesmo que não exista um *return* declarado. Ao término da computação sem *return* ela irá cair em um bloco com rótulo 0. Para uma melhor visualização da estrutura gerada, os blocos antepostos por `Entry` recebem como endereço interno o número 1. Haverá somente um bloco rotulado como `Entry`, mas podem haver *n* blocos `Leave`.

A forma com a qual se geram os grafos de fluxo de controle facilitam a leitura e o repasse dos dados para a próxima fase da tradução de código. A partir destas informações pode-se dar continuidade ao processo de compilação. A próxima etapa utiliza os fluxos de controle criados para calcular a dominância dos blocos e assim gerar a forma de atribuição estática única, o SSA.

3.2.2.2 Forma de Atribuição Estática Única

Calcular a dominância do CFG é necessária para a geração de forma otimizada do SSA. Essa relação irá permitir a substituição das variáveis repetidas do código por variáveis com nomes novos. O algoritimo de dominância canônico, descrito na literatura, é projetado para linguagens imperativas, sendo necessária uma considerável adaptação para que funcione em Haskell.

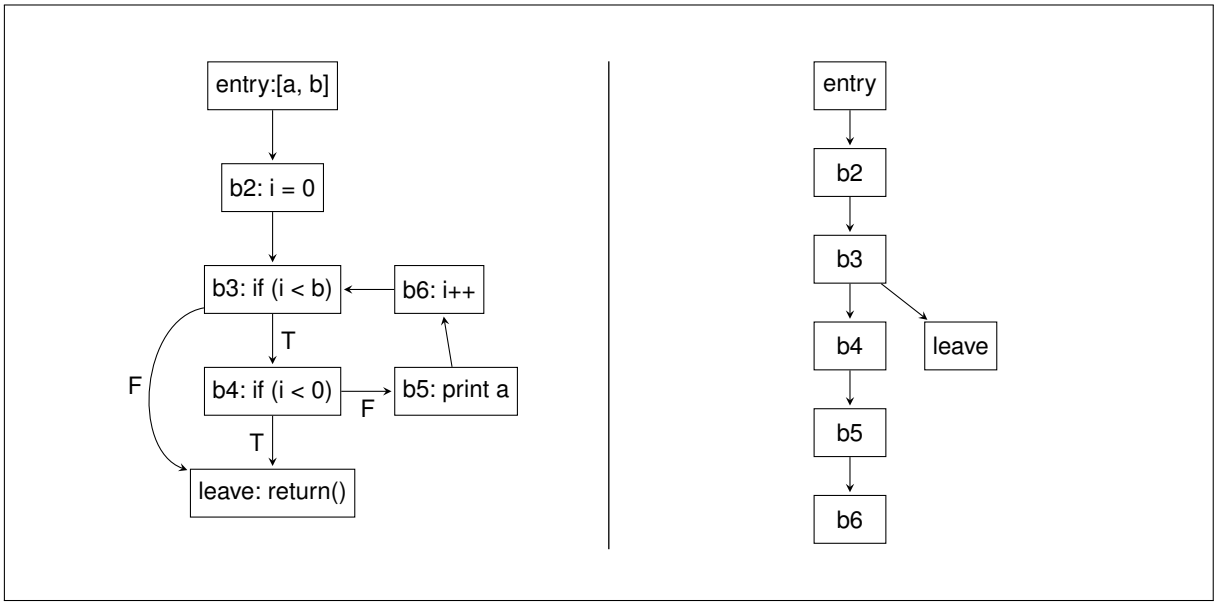
Para facilitar essa etapa da implementação foi utilizada uma adaptação do algoritimo de dominância encontrado na literatura vindo do projeto *cao* [1]. Esse algoritmo utiliza particionamento de grafos a fim de encontrar a árvore de dominância, da qual são extraídas as informações de dominância. De forma simplificada, a árvore de dominância representa o nó mais próximo que será necessariamente percorrido para se gerar um caminho do nó inicial do grafo ao nó desejado. Além disso, com este algoritimo é possível gerar a informação da fronteira de dominância que é necessária para calcular o número mínimo de nós ϕ.

Utilizando o mesmo exemplo de algoritimo da Figura 27, a árvore de dominância gerada para o grafo da função `bar` é exibido na Figura 28. Essa árvore mostra os blocos dominantes e a partir dela, pode-se gerar a informação de fronteira de dominância que é de onde se delimita a posição dos nós ϕ no grafo.

A árvore mostra que o bloco *Leave* somente poderá ser alcançado pelo bloco *b3*. Isso quer dizer que o bloco *b3* domina estritamente o bloco *Leave*. Os demais blocos são dominados pelos seus predecessores de forma que só é possível se chegar a um bloco dominado através de seu dominante. Com essa informação pode-se calcular a *dominance frontier* deste grafo. Primeiramente gera-se o conjunto de nós que são

[1] http://hackage.haskell.org/package/cao-0.1

Figura 28 – Exemplo - Geração da árvore de dominância



Fonte: O Autor

dominados, também chamada de região de dominância, pelo bloco *b3* (*b4*, *b5*, *b6*, e *Leave*). Após a informação de dominância ser gerada, calcula-se a fronteira de dominância para esse bloco com base no CFG. A fronteira de dominância diz respeito à quais nós alcançados por *b3* estão fora da região que este domina. Observa-se que o bloco que excede a esse limite é o próprio bloco *b3*, pois ele está além de sua própria fronteira de dominância e é alcançado por si mesmo.

Esse algoritimo foi implementado seguindo a especificação da literatura e permite retornar a fronteira de dominância, parte do que é necessário para a inserção do número mínimo de funções ϕ no grafo. A implementação deste, e dos demais algoritimos utilizados na construção do compilador estão dispostas no repositório TE-Infer-SSA[2].

Além da informação de fronteira de dominância, para a inserção das funções ϕ no CFG corrente, é necessário listar quais são as atribuições de variáveis contidas no código. Para a obtenção dessa lista de declaração de variáveis, executa-se uma varredura no grafo buscando-as. Em seguida é obtida a informação dos escopos aos quais estas variáveis respondem, estabelecendo uma lista de duplas com estas informações. Para o CFG da função `bar`, gera-se a seguinte lista:

```
knowDeclVariables = [("a",0),("b",0),("i",2)]
```

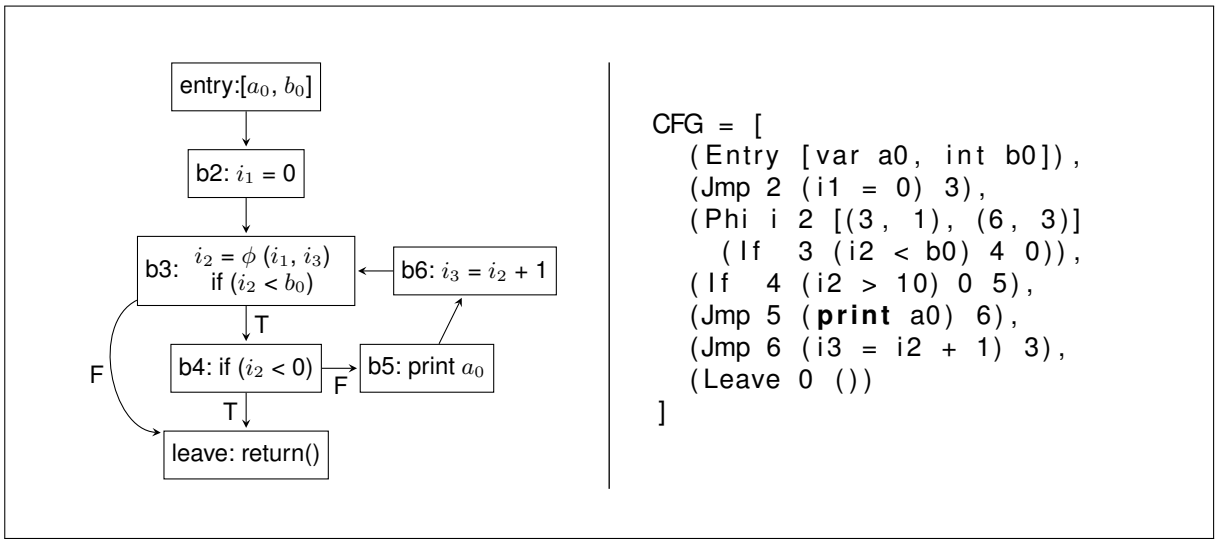
Essa lista apresenta a variável declarada e o escopo ao qual ela está atrelada. As variáveis `a` e `b`, por serem parâmetros da função, estão no escopo 0, já a variável

[2] https://github.com/leorigon/TE-Infer-SSA

`i` está no escopo 2 pois é declarada no *statement* `for` que corresponde ao segundo bloco básico. Se houvesse uma declaração de variável anterior à chamada do `for`, essa variável pertenceria ao escopo 1.

Após o cálculo da fronteira de dominância e com a listagem de variáveis conhecidas e seus escopos, são inseridos os nós ϕ no grafo e também renomeiam-se as demais variáveis de forma que só exista uma variável com o mesmo rótulo. A Figura 29 mostra então o CFG com as variáveis renomeadas e o nó ϕ.

Figura 29 – Exemplo - Atribuição estática única gerada



Fonte: O Autor

As variáveis oriundas de atribuições também recebem nomes únicos, como no caso do operador de incremento do bloco *b6*. A variável *i1* definida no bloco *b2* é passada como argumento de ϕ, sendo representada a partir de agora pela variável única *i2* sendo propagada para o restante do grafo. No bloco *b6* ocorre a soma de *i2* com 1, gerando a variável *i3* que será enviada ao bloco *b3* sendo o último parâmetro de ϕ. A função ϕ está contida no bloco *b3* pois a variável *i*, seguindo o algoritimo, pode assumir valores distintos dependendo de onde vem o fluxo de controle, mas deve ser atribuída apenas uma única vez.

O CFG criado durante o processo de compilação exibido do lado direito da Figura 29 utiliza o tipo `Node` definido para os blocos básicos e mostrado na subseção 3.2.2.1. Neste tipo, agrega-se um construtor para representar os nós ϕ, conforme a definição abaixo:

```
data Node = Entry Parameters Label
          | Jump  Label Command Label
          | IF    Label Expression Label Label
          | Leave Label Expression
          | Phi   String Int [(Label, Int)] Node
```

O construtor `Phi` tem como parâmetros uma string que representa uma variável e um inteiro representando seu índice. Em seguida, compondo o restante do construtor, uma lista de duplas contendo um rótulo de bloco básico `Label` e um índice. Fechando o construtor, é apresentado um `Node` que poderá ser qualquer um dos blocos básicos, exceto se o bloco em questão comece por `Entry`.

O exemplo da lista gerada para o algoritmo bar (Figura 27) recebe o construtor `Phi` no bloco 3 é apresentado ainda na direita da Figura 29. Neste momento essa é a única mudança no CFG, juntamente com as renomeações de variáveis, gerando assim a representação intermediária SSA. Com o SSA gerado, pode-se dar continuidade ao processo de tradução de código e gerar uma representação funcional equivalente.

3.2.2.3 Geração da representação funcional

A representação funcional aqui proposta consiste em um cálculo lambda estendido com alguns tipos primitivos, expressões *let* (como de costume na literatura), e expressões *where*, sendo a penúltima fase do processo de análise semântica proposto. Sua geração fará uso das informações obtidas por meio do SSA e também da dominância do grafo que gerou a representação. A primeira parte desta etapa consiste em transformar a lista de `Node` em um tipo que represente o cálculo lambda. Para isso foi criado o tipo `Expr` representado na Figura 30.

Figura 30 – Definição - Tipo Expr

```
1  data Expr = Free String
2            | Number Int
3            | Text String
4            | UnitValue
5            | TrueValue
6            | FalseValue
7            | Let String Expr Expr
8            | Lambda String Expr
9            | If Expr Expr Expr
10           | Application Expr Expr
11           | Operation Operator Expr Expr
12           | Where [(String, Expr)] Expr
```

Fonte: O Autor

O tipo `Expr` pode assumir qualquer um dos valores do lado direito da igualdade, ele representa o *calculus* que será utilizado na inferência de tipos e efeitos posteriormente. Podendo ser uma variável, representada pelo construtor `Free` e uma `String`, um literal numeral (`Number Int`), uma string (`Text` String). Há também os construtores que representam os valores booleanos, `TrueValue` e `FalseValue`, além do tipo que representa *unit* `UnitValue`. Até aqui, esses construtores representam os valores

que são aceitos na conversão da linguagem fonte para uma IR em forma de cálculo lambda.

Dando continuidade aos construtores do tipo `Expr`, tem se o construtor `Let`. Esse construtor corresponde ao let utilizado em linguagens funcionais. Ele é seguido de uma *string* que representa a variável que receberá uma `Expr` e outra expressão que pode representar qualquer um dos construtores do tipo `Expr`. Já o construtor `Lambda` recebe uma string, em forma de variável e uma `Expr`. A condicional *if* recebe 3 `Expr`, a primeira representa a condição, a segunda o que vem após o *then*, e por último o que viria após, o *else*.

O construtor `Application` representa a aplicação de função. Ele recebe duas expressões como parâmetros. O construtor `Operation` também representa a aplicação de funções, porém ele é mais específico, sendo utilizado apenas para operadores matemáticos sendo representados por `Operator` e duas expressões, igual ao construtor de aplicação. Já o último construtor, `Where`, é um pouco diferente pois recebe uma lista de duplas contendo uma string, que é uma variável, e uma `Expr` e por último uma expressão para dar continuidade ao cálculo lambda. Seu funcionamento é análogo ao *where* utilizado em Haskell. Para exemplificar a conversão, temos um pequeno algoritimo da função identidade escrito na linguagem deste trabalho:

```
algorithm id(var x) { return x; }
```

A função `id` é convertida para uma representação funcional, preservando suas propriedades, que faz uso do tipo `Expr`, conforme a 31.

Figura 31 – Exemplo - *Algorithm id* convertido para uma representação funcional

```
Lambda "x0" (
    Where
        [("B2", Lambda "_" (Free "x0"))]
        (Application
            (Free "B2")
            UnitValue
        )
)
```

Fonte: O Autor

A aplicação da função B2 representa o corpo da função lambda declarada, onde o construtor `Where` delimita o escopo da função `B2`, visível dentro da aplicação. A variável `x0` aparece livre no bloco `B2`, e é então ligada à função lambda à qual pertence. Em outras palavras: como a função B2 está contida em um escopo interno da função lambda, ela é capaz de ver seus parâmetros.

A lista de duplas contém o próprio `B2`, uma nova `Lambda` que recebe um *wild-card* e a variável `x0` como argumento. O *underscore* é necessário pois a linguagem aqui descrita e implementada, diferentemente de Haskell, utiliza avaliação estrita. Ou seja, os argumentos de uma computação serão sempre totalmente avaliados antes de sua aplicação. Portanto, para o funcionamento do compilador, é necessário atrasá-los para que se possa compilar o código corretamente. Outro detalhe importante é que como será utilizado o algoritimo do trabalho de Leijen (2014) para inferência de tipos e efeitos, se faz necessária essa adaptação, pois o algoritimo requer que todas as funções e argumentos sejam puros.

Para finalizar, há a aplicação de função, onde se aplica o bloco e uma *unit*, que forçará a execução dos efeitos contidos dentro do escopo do bloco. Isso corresponde ao retorno de função. Para este trecho pequeno de código, a tradução ainda é legível, porém em códigos maiores a tradução se torna de difícil leitura. Para isso, foi implementada uma *instance* que torna o cálculo lambda mais intelegível, conforme o exemplo a baixo da função `id`:

```
\x0 -> let b2 _ = x0 in (b2 ())
```

Para elucidar melhor a tradução, a estrutura explicitada na Figura 29, para a função `bar`, na Subseção 3.2.2.3, a lista de nós `CFG` foi convertida para essa última apresentação de cálculo lambda. A conversão da lista de `[Node]` gerada com o construtor `Expr` é exibida na Figura 32.

De acordo com essa representação intermediária, o bloco inicial `Entry` com os parâmetros `a0` e `b0` são convertidos para `\ a0 b0 ->` sendo representados na primeira linha da figura. Esse valores são passados adiante para o segundo bloco identificado pela declaração de `let B2 _`, este bloco irá receber o valor de `i1` e o repassará para o bloco `B3` e retornará, ao término do deste fluxo, o resultado da computação (`in` (`B2 ()`). O bloco b3 é convertido em `B3` recebendo como parâmetro o valor `i2`, assim como ocorre em seu bloco antecessor, quando concluída a computação repassada à `B0` e `B4`, seu valor será obtido através de (`in` (`B3 i1`) `()`) .

O `let B0 _ = ()` é repassado à chamada da condicional representada por `if i2 < b0`, assim como o retorno da computação iniciada em `let B4`. O resultado oriundo desta condição se verdadeira será o cálculo executado em `B4`, caso falsa, assumirá o valor de `b0`. Já a chamada `let B4 _ =` recebe `let B4 _ =` e efetua a verificação `if i2 < 10`, fazendo processo semelhante ao ocorrido no no bloco b3. Porém aqui, se a condição for satisfeita, assumirá o valor de (`B0 ()`), se for falsa assumirá a computação de (`B5 ()`). O bloco b5, representado por `let B5 _ =`, recebe a função `print a0` inserindo-a na chamada de `B6` que efetuara o operador de incremento, representado pela operação `let i3 = i2 + 1` e irá inserir este valor como `i3` sendo passado como parâmetro para

Figura 32 – Exemplo - *Algorithm bar* convertido para uma representação funcional

```
\a0 b0 ->
  let B2 _ =
        let i1 = 0 in
        let B3 i2 _ =
              let B0 _ =
                    ()
                  B4 _ =
                    let B5 _ =
                          let _ = (print a0) in
                          let B6 _ =
                                let i3 = i2 + 1 in
                                ((B3 i3) ())
                          in (B6 ())
                    in if i2 > 10
                         then (B0 ())
                         else (B5 ())
              in if i2 < b0
                   then (B4 ())
                   else (B0 ())
        in ((B3 i1) ())
  in (B2 ())
```

Fonte: O Autor

`B3`. Ainda na Figura 32 pode se fazer uma analogia à relação de dominância citada no trabalho de (APPEL, 1998), onde os blocos dominados representam escopos internos da representação funcional.

O processo até aqui descrito permite transformar qualquer código escrito na linguagem proposta neste trabalho em uma extensão do cálculo lambda com tipos primitivos e as chamadas de *let* e *where*. Podendo ainda, essa representação funcional, ser executada e compilada pelo compilador de Haskell, o GHC, caso seja uma computação pura. Lembrando que a alteração de variáveis locais não altera a pureza da função compilada, pode-se livremente alterar variáveis locais, pois isso não consiste em um efeito colateral. Após essa tradução pode-se efetuar a aplicação de um sistema de tipos para se verificar as propriedades do programa.

3.2.2.4 Aplicação do sistema de tipos e efeitos

O algoritimo de sistema de tipos e efeitos empregado no escopo desse projeto é semelhante, com algumas pequenas adaptações, ao utilizado e apresentado no trabalho de Leijen (2014) abordado no Capítulo 2. A adequação mais contundente, consiste na forma com a qual se adaptou o algoritimo de inferência para atuar no cálculo gerado a partir do SSA. O cálculo considerado aqui é representado pelo tipo `Expr`, mostrado na Figura 30 na Seção anterior. Foi adicionada uma nova regra para

se poder efetuar a inferência utilizando o construtor `Where`.

Para dar continuidade ao processo de compilação e iniciar a inferência de tipos e efeitos, foi definido um tipo chamado `Type` que abriga os construtores representando os tipos válidos da linguagem. A Figura 33 exibe os construtores, escritos em Haskell, de tipo aceitos na inferência do cálculo.

Figura 33 – Definição - Tipos válidos na linguagem

```
1  data Type = Int
2            | Bool
3            | String
4            | Unit
5            | Console
6            | Pure
7            | Const String
8            | Generic String
9            | Arrow Type Effect Type
10           | Ref String Type
11           | State String
12           | Row Effect Effect
13
14 data Effect = Type
```

Fonte: O Autor

Nessa figura, além dos tipos primitivos da linguagem (`Int`, `Bool`, `String` e `Unit`), são exibidos as construções de tipos que podem ser obtidos por meio da inferência. O tipo de efeito `Console` que é inserido no contexto quando há a chamada da palavra reservada `print` e o tipo `Pure` que representa uma computação pura, que não tem efeitos colaterais. O tipo `Const` representa o construtor de efeitos declarados pelo programador, recebendo uma string como efeito. As variáveis de tipo são representadas por `Generic` seguido de uma variável. O construtor `Arrow Type Effect Type` representa a aplicação de funções, recebendo 3 tipos como parâmetros. Foi adicionado um tipo que efetua a passagem de valor por referência (`Ref Type`) recebendo como parâmetro um tipo qualquer. O tipo `State` denota uma computação de estados. Por último, o construtor `Row` que tem como parâmetros dois efeitos e serve para representar as *effect rows*. Cada um desses efeitos representa um tipo. Utilizando a função identidade, definida na Seção anterior e exibida na Figura 31, pode-se inferir o tipo dela utilizando o tipo `Type`:

```
Arrow (Generic "b") (Generic "a") (Generic "b)
```

Diferentemente do Damas-Milner tradicional, aqui o construtor da funções `Arrow` recebe 3 parâmetros, o segundo parâmetro, `Generic "a"` receberá uma variável que representa o efeito desta função. Neste caso, ela recebe um efeito algébrico do tipo `"a"` representando uma computação pura. A primeira etapa do processo de inferência

consiste na definição das regras a serem utilizadas para a tipagem das expressões. A adição de termos para a tipagem de efeitos é agregada de forma que não venha a interferir na inferência de tipos. A formalização das regras do algoritimo adaptado utiliza a gramática demonstrada na figura 34:

Figura 34 – Definição - Gramática da formalização algoritimo de inferência de tipos e efeitos adaptado

$$
\begin{array}{lcll}
e & ::= & x & (variável) \\
 & | & \lambda x.e & (função\ lambda) \\
 & | & e_1 e_2 & (aplicação) \\
 & | & let\ x = e_1\ in\ e_2 & (let\ binding) \\
 & | & where\ \{x_1 = e_1; ...; x_n = e_n\} & (where)
\end{array}
$$

Fonte: O Autor

O cálculo utilizado é uma extensão do cálculo lambda que receberá além das variáveis e das principais construções (aplicação, função lambda, let) o $where$ adaptado. O $where$ recebe como parâmetros uma lista de atribuições de expressões e_i em variáveis x_i. Uma nova regra necessária para tratar blocos foi agregada ao algorítimo de Leijen (2014), conforme explicitado na Figura 35. Cada bloco básico irá ser transformado na chamada de `where`, contendo os seus blocos dominados.

Figura 35 – Definição - Algoritimo de inferência de tipos e efeitos adaptado

$$
\frac{\Gamma(x) = \forall \overline{\alpha}.\tau}{\varnothing\Gamma \vdash_i x : [\overline{\alpha} \mapsto \overline{\beta}]\ \tau \mid \mu} \quad \textbf{(VAR)}_i
$$

$$
\frac{\theta\Gamma_x \cup \{x : \alpha\} \vdash_i e : \tau \mid \epsilon}{\theta\Gamma \vdash_i (\lambda x.e) : \theta\alpha \to \epsilon\ \tau \mid \mu} \quad \textbf{(ABS)}_i
$$

$$
\frac{\theta_0\Gamma \vdash_i e : \tau \mid \epsilon \quad \theta_1(\theta_0\Gamma) \vdash_i e' : \tau' \mid \epsilon' \quad \theta_3\tau \sim (\tau' \to \epsilon'\alpha) : \theta_2 \quad \theta_2\theta_1\epsilon \sim \theta_2\epsilon' : \theta_3}{\theta_3\theta_2\theta_1\theta_0\Gamma \vdash_i (e\ e') : \theta_3\theta_2\alpha \mid \theta_1\theta_2\epsilon} \quad \textbf{(APP)}_i
$$

$$
\frac{\theta_0\Gamma \vdash_i e : \tau \mid \epsilon \quad \epsilon \sim \langle\rangle : \theta_1 \quad \sigma = gen(\theta_1\theta_0\Gamma,\ \theta_1\ \tau) \quad \theta_2(\theta_1\theta_0\Gamma, x : \sigma) \vdash_i e' : \tau \mid \epsilon}{\theta_2\theta_1\theta_0\Gamma \vdash_i (let\ x\ =\ e\ in\ e') : \tau \mid \epsilon} \quad \textbf{(LET)}_i
$$

$$
\frac{\begin{array}{c}\Gamma_2 = \Gamma_1, x_1 : \alpha_1,\ ...\ , x_n : \alpha_n \\ \forall i \in 1..n \begin{cases} \theta_{3i-2}(\theta_{3i-3}...\theta_1\Gamma_2) \vdash_i e_i : \tau_i|\epsilon_i \\ \theta_{3i-2}...\theta_1\alpha_i \sim \tau_i : \theta_{3i-1} \\ \theta_{3i-1}\ \epsilon_i \sim \langle\rangle : \theta_{3i} \end{cases} \\ \theta_{3n+1}(\theta_{3n}...\theta_1\Gamma_2) \vdash_i e :\ \tau|\epsilon \end{array}}{\theta_{3n+1}...\theta_1\Gamma_1 \vdash_i e\ where\ \{x_1 = e_1;\ ...\ ; x_n = e_n\}\ : \tau|\epsilon} \quad \textbf{(WHERE)}_i
$$

Fonte: O Autor

As quatro primeiras regras da formalização são o que o Leijen (2014) propõe para sua linguagem. Vale ressaltar que as regras de inferência de generalização e instanciação foram absorvidas pelas regras VAR_i e LET_i. A regra $WHERE_i$ é utilizada

para a inferência de um conjunto de termos mutuamente recursivos, similar a expressões *let* em algumas linguagens funcionais. Na extensão do cálculo lambda utilizada, expressões *where* são utilizadas para funções locais (que surgem a partir de blocos estritamente dominados no grafo de fluxo de controle), e por isso são monomórficos (não há generalização após a inferência). Cada variável de tipo é unificada ao corpo de cada declaração de forma padrão, levando em conta que a mesma pode ter sido alterada internamente (visto que chamadas recursivas são permitidas). Além disso, similar à expressões *let*, um efeito não pode ser executado ao criar o bloco, e quaisquer efeitos desejados devem estar devidamente abstraídos por uma função lambda.

A nova regra ($WHERE_i$) é necessária para que se possa efetuar a inferência do cálculo funcional aqui utilizado. Ela não se encontra na literatura e portanto, pode ser considerada também uma contribuição desse trabalho. Na prática ela funciona como um acumulador de substituições de tipos por meio de um *fold*. Não foram feitas alterações significativas nas demais regras de inferência.

Conforme a formalização, as regras foram implementadas utilizando *pattern matching* que recebe como entradas um contexto, a expressão lambda em si e retorna um tipo ou um erro caso não seja possível tipar a expressão. A Figura 36 exibe a implementação da função `infer` para os principais casos, alguns casos foram omitidos para simplificar a exibição no corpo do trabalho.

Todos os casos de `infer` receberão um contexto como primeiro argumento. O contexto abriga as substituições de tipos que serão utilizadas na inferência. Esse conjunto é composto de tuplas contendo uma variável e um tipo da linguagem que o representa. Algumas *keywords* da linguagem contidas no projeto são nativas do contexto do inferidor, como é o caso da função `print`:

```
("print", Forall ["a", "u"] $ Arrow (Generic "a") (Row Console $ Generic "u") Unit)
```

O tipo de `print` será acionado e passado para a inferência caso o código contenha um *statement* de impressão. Vale ressaltar que Haskell trata funções como variáveis, então é possível assumir que `print` é uma variável nessa situação. Variáveis, constantes e demais funções durante o processo de inferência são inseridas ao contexto quando acionadas.

Ainda na Figura 36 é possível observar a regra **(VAR)**$_i$ na primeira ocorrência de `infer`. Onde é recebido, além do contexto, uma variável qualquer. Executando um `lookup` [3] da variável no contexto, se encontrada, armazenando o resultado em `sigma`. Em seguida este termo é instanciado gerando um tipo `t`. Ao término dessa compu-

[3] A função `lookup` faz a busca de um valor verificando o primeiro elemento de duplas contidas em uma lista, se encontrá-lo, retornará o segundo elemento como parâmetro do construtor `Just` https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html

Figura 36 – Implementação - Algoritimo de inferência de Tipos e Efeitos

```
infer (Environment env) (Free var) =
    case M.lookup var env of
        Just sigma -> do
            t <- instantiate sigma
            mi <- newTypeVar
            return (empty, t, mi)
        Nothing ->
            throwError (UnboundVariable var)

infer env (C.Lambda x e) = do
    alpha <- newTypeVar
    (theta, tau2, epsilon2) <- infer (extend env x alpha) e
    mi <- newTypeVar
    return (theta, Arrow (subst theta alpha) epsilon2 tau2, mi)

infer env (Application e1 e2) = do
    (theta1, tau1, epsilon1) <- infer env e1
    (theta2, tau2, epsilon2) <- infer (ubst theta1 env) e2
    alpha <- newTypeVar
    theta3 <- unify (subst theta2 tau1) (Arrow tau2 epsilon2 alpha)
    theta4 <- unify (subst (theta3 @@ theta2) epsilon1) (subst theta3 epsilon2)
    return (theta4 @@ theta3 @@ theta2 @@ theta1,
                C.subst (theta4 @@ theta3) alpha,
                    C.subst (theta4 @@ theta3) epsilon2)

infer env (Let x e1 e2) = do
    (theta1, tau1, epsilon1) <- infer env e1
    let Environment env1 = remove env x
    let tau2 = generalize (subst theta1 env) tau1
    let env2 = Environment (insert x tau2 env1)
    theta2 <- unify epsilon Pure
    (theta3, tau3, epsilon2) <- infer (subst (theta2 @@ theta1) env2) e2
    return (theta3 @@ theta2 @@ theta1, tau2, epsilon2)
```

Fonte: O Autor

tação será gerada uma tripla contendo um conjunto vazio, um tipo `t` e um efeito `mi`. Se o `lookup` falhar, e por consequência retornar o construtor `Nothing`, será retornado um erro dizendo que essa variável não está presente no contexto e portanto não é possível inferir o tipo dela.

Se `infer` receber, como segundo parâmetro, um número, uma *unit*, um valor booleano ou uma *string*, retornará uma tripla contendo uma lista vazia, o tipo correspondente ao valor e uma variável *fresh*. Esse terceiro valor da tripla corresponde ao efeito da expressão neste momento da inferência.

Quando o segundo parâmetro passado para `infer` tiver o construtor `Lambda`, será criada uma nova variável de tipo (`alpha`) e seus dois parâmetros `x` e `e` serão repassadas como parâmetros para a nova chamada de `infer`. Na chamada recursiva de `infer` serão repassados a extensão do contexto `env` abrigando `x` e `alpha` e a expressão `e`. O resultado dessa computação será uma tripla contendo uma substituição `theta`, um tipo `tau2` e um efeito `epsilon`. Gera-se uma nova variável (`mi`) que será o terceiro argumento do retorno da função que conterá uma substituição `theta`, uma

aplicação de função representada pelo construtor `Arrow` que terá como argumentos o retorno da substituição aplicada a `theta` e `alpha`, o efeito `epsilon2` e o tipo `tau2`. Essa etapa corresponde à regra**(ABS)**$_i$ formalizada no algoritimo.

O terceiro caso do casamento de padrões de `infer` é acionado se o segundo parâmetro de sua entrada for iniciado pelo construtor `Application` representando a regra **(APP)**$_i$. A função `infer` é chamada recursivamente 2 vezes, uma vez para cada um dos argumentos da aplicação (`e1` e `e2`). Armazenando seus resultados em duas triplas contendo cada uma delas uma lista de substituição (`theta1` e `theta2`), um tipo (`tau1` e `tau2`), e um efeito (`epsilon1` e `epsilon2`). Uma nova variável `alpha` é criada, em seguida são executadas as unificações que serão armazenadas em `theta3` e `theta4` respectivamente. A primeira unificação recebe como parâmetro o resultado da substituição do tipo `tau1` utilizando a lista de substituições `theta2` e como segundo parâmetro um tipo `Arrow tau2 epsilon2 alpha`. O valor obtido de `theta4` é criado por meio da unificação do resultado da substituição de `epsilon1` na composição (`@@`) das substituições `theta3` e `theta2` e do resultado da substituição de `epsilon2` em `theta3`. O retorno dessa chamada de `infer` é composto pela composição de todas as substituições θ geradas na mônada, do resultado da substituição de `alpha` na composição de `theta4` e `theta3` e da substituição de `epsilon2` na mesma composição anterior (`theta4 @@ theta3`).

Uma composição (`@@`) é equivalente à aplicação sequencial das substituições que recebem o operador. As unificações que criam `theta3` e `theta4` são dadas pela função `unify`, essa função, assim como boa parte do algoritimo, segue o padrão da literatura e será abordada mais adiante nessa mesma sessão.

A última chamada de `infer`, na linha 26 da Figura 36, diz respeito à regra formalizada **(LET)**$_i$. Ela é acionada quando é recebido como segundo parâmetro de entrada o construtor `Let`. Primeiramente, a função chama recursivamente `infer` passando como argumentos o contexto `env` e `e1`. Em seguida cria um novo contexto a partir da remoção da substituição que tem como chave a variável `x` do contexto `env`. Em seguida gera-se um tipo `tau2` a partir da generalização do resultado da substituição de `theta1` e `env` e do tipo `tau1`. Após isso é criado um novo contexto `env2` que será composto da inserção do valor de `x` com o tipo `tau1` no contexto `env1`. Se já existir uma variável ou função igual à `x`, será substituído pelo tipo `tau1`. A substituição `theta2` é dada pela unificação do efeito `epsilon` com o efeito `Pure`. As substituições `theta2` e `theta1` serão, juntamente com o contexto `env''` passadas como argumento para a função `subst` gerando um novo contexto que será argumento de `infer` juntamente de `e2` gerando uma tripla contendo uma substituição `theta3`, um tipo `tau2` e um efeito `epsilon2`. E por fim, como retorno dessa computação, será retornada uma tripla com a composição das 3 substituições θ geradas, do tipo `tau2` e do efeito `epsilon2`.

A última regra da formalização, o **Where**$_i$, tem sua implementação apresentada na Figura 37, sendo executado quando o *pattern matching* recebe o construtor `Where` como segundo argumento após o contexto `env`.

Figura 37 – Implementação - Regra Where

```
1  infer env (Where bindings e) = do
2      alpha <- mapM (const newTypeVar) bindings
3      let env' = foldl (\acc (var, (block, _)) ->
4                            extend acc block var) env (zip alpha bindings)
5      (env'', theta) <- foldlM inferBlock (env', empty) (zip alpha bindings)
6
7      if (subst theta env') /= env''
8          then
9              error "internal compiler error"
10         else
11             return ()
12     (theta2, tau2, epsilon2) <- infer env'' e
13     return (theta2 @@ theta, tau2, epsilon2)
14
15     where
16         inferBlock (env', theta_i) (alpha, (var, block)) = do
17             (theta_i', tau, epsilon) <- infer env' block
18             let alpha' = subst (theta_i' @@ theta_i) alpha
19             theta_i'' <- unify alpha' tau
20             let env'' = subst (theta_i'' @@ theta_i') env'
21             return (env'', theta_i'' @@ theta_i' @@ theta_i)
```

Fonte: O Autor

Primeiramente é gerado uma nova variável de tipo para cada uma das variáveis contidas em `bindings` gerando o valor de `alpha`. Em seguida é estendido o contexto, gerando um novo contexto (`env'`) onde o contexto recebido como entrada da função é estendido com uma nova variável de tipo para cada um dos blocos presentes nesse *where* (isso equivale ao Γ_2 da Figura 35). A próxima tarefa consiste na geração de uma dupla contendo o novo contexto `env''` e uma substituição `theta`, essa geração é dada pela acumulação, através de `foldM`, das substituições. No código, é feita também a acumulação do contexto para facilitar a implementação, diferente da formalização, onde substituições são reaplicadas ao contexto ao término da acumulação.

A função `inferBlock` recebe 2 parâmetros, um contendo um contexto `env''` e uma substituição `theta_i`, e o outro contendo uma variável de tipo `alpha` e mais uma tupla contendo o nome do bloco e o bloco em si. Dentro dessa chamada são efetuadas substituições e unificações de modo que ao término dessa computação se retorna um contexto `env''`, e uma composição de substituições θ.

Se a substituição, aplicada a `theta` e `env'` for diferente do contexto `env''`, uma exceção será emitida, do contrário, será retornado uma tripla contendo o resultado da

aplicação recursiva de `infer` no contexto `env'` e `e`. Uma tripla, contendo composição das substituições `theta2` e `theta`, um tipo `tau2` e um efeito `epsilon2`.

A união de tipos segue o que é proposto na literatura e não sofreu alterações significantes em sua adaptação. Porém foi necessária a adição de novos casos no algoritimo para tratar construções específicas da linguagem, conforme a Figura 38.

Figura 38 – Implementação - Algoritimo de Unificação para casos adicionais

```
1  unify (Row l epsilon1) epsilon2 = do
2      (epsilon3, theta1) <- unifyEffect epsilon2 l
3
4      when (member (tl epsilon1) theta1) $ do
5          throwError $ EffectTailCheck (tl epsilon1)
6      theta2 <- unify (subst theta1 epsilon1) (subst theta1 epsilon3)
7      return (theta2 @@ theta1)
8      where
9          tl (Row _ tail) = tl tail
10         tl  C.Pure = "___"
11         tl (Generic a) = a
```

Fonte: O Autor

O `unify` da Figura 38 tem seu comportamento acionado quando se recebe como parâmetro um construtor `Row` e um outro construtor qualquer. Primeiramente se aciona a função `unifyEffect`, abordada na Figura 39, passando como parâmetros o valor de `epsilon2` e um efeito `l`, gerando uma dupla (`epsilon3`, `theta1`). A chamada da função `when` é utilizada para garantir que o algoritimo de unificação sempre termine. Essa função foi transcrita de forma igual ao que foi proposto no artigo de Leijen (2016). A Figura 39 exibe o trecho do código da adaptação do algoritimo de união de efeitos da aplicação de funções:

Figura 39 – Implementação - Algoritimo de Unificação de efeitos

```
1  unifyEffect (Row l' epsilon) l =
2      if l == l'
3          then do
4              theta <- unify l l'
5              return (epsilon, theta)
6          else do
7              (epsilon', theta) <- unifyEffect epsilon l
8              return (Row l' epsilon', theta)
9
10 unifyEffect (Generic mu) l = do
11     mu' <- newTypeVar
12     return (mu', singleton mu (Row l mu'))
13
14 unifyEffect a b = do
15     throwError $ CannotUnify a b
```

Fonte: O Autor

A função `unifyEffect` é acionada quando a unificação, em seu *pattern mat-*

ching recebe como primeiro parâmetro o construtor `Row` passando como argumento para `unifyEffect` o segundo argumento da união (`epsilon2`) e o primeiro parâmetro da `Row`, `l`:

```
unify (Row l epsilon1) epsilon2
```

Se o `epsilon2` for uma nova `Row`, cairá no primeiro caso, e efetuará a comparação do primeiro argumento desta `Row` com o parâmetro `l`, advindo da função `unify`. Se estes valores forem iguais, irá executar a unificação dos mesmos, retornando-o como segundo parâmetro de uma tupla que será iniciada pelo valor do segundo parâmetro (`epsilon`) da `Row` passada para `uniffyEffect`. Caso os valores de `l` e `l'` sejam diferentes, será chamada, recursivamente, a função `uniffyEffect` passando como argumento `epsilon` e `l`, o retorno desta computação será uma tupla contendo `epsilon'` e `theta`. O primeiro argumento dessa tupla receberá o construtor `Row` e o parâmetro `l'` e juntamente com `theta`, em forma de tupla, será retornada.

Se `unifyEffect` receber o construtor `Generic`, que representa uma variável de tipo como primeiro argumento, irá ser gerada uma variável de tipo (`mu'`) sendo passada como primeiro argumento da tupla de retorno desta computação. O segundo argumento será dado pela chamada de `singleton`, do valor recebido `mu` e de um construtor `Row` com o valor do segundo argumento recebido por `unifyEffect`, `l` e de um `mu'`.

Se a função `unifyEffect` receber como primeiro parâmetro qualquer coisa diferente de `Row` ou `Generic`, a mesma irá retornar um erro dizendo que é impossível unificar os tipos recebidos. O primeiro caso desse casamento de padrões, em sua condicional, representa a relação de equivalência **EFF-HEAD** se verdadeira e **EFF-SWAP**, se falsa.

Para mostrar o funcionamento do inferidor de tipos e efeitos, será utilizada a mesma função `bar` da seção anterior. A Figura 40 a seguir, exibe a entrada e a saída do compilador.

Figura 40 – Exemplo - Tipagem *algortihm bar*

```
1 algorithm bar(var x, int y) {                bar :: ∀ a b c.a → c int → <console, b> unit
2     for(int i = 0; i < y; i++) {
3         if(i > 10) {
4             break;
5         }
6         print(x);
7     }
8 }
```

Fonte: O Autor

O tipo exibido no lado direito da Figura 40 corresponde ao tipo mais geral da função `bar`, recebendo como entrada um tipo polimórfico `a` e um `int` executando um efeito polimórfico `c` (que, efetivamente, marca que nenhum efeito é introduzido no ponto onde se forneceu o primeiro parâmetro). A construção `<console,b>` diz respeito à *effect row* inferida para essa função, sendo um efeito aberto contendo o tipo de efeito `console` por causa da chamada de `print` e uma variável de tipo de efeito `b`. Em seguida é exibido o tipo do retorno da função, uma `unit`, semelhante ao que ocorre em Haskell quando se utiliza a mônada de IO para imprimir alguma computação na tela. São trazidas também, as informações das variáveis livres de tipo que foram utilizados durante a inferência antes do ponto precedidas por $\forall$ para ilustrar o que foi gerado durante a tipagem da função.

O compilador aceita como entrada arquivos de texto compilando-os para a representação funcional através da transformação do código fonte em uma representação intermediária em forma de grafos de fluxo de controle com a atribuição estática única. Após a compilação para o cálculo de expressões proposto é aplicado um algoritimo de inferência de tipos e efeitos que tem como saída o tipo do algoritimo ou um erro, caso exista alguma inconsistência. Conforme abordado anteriormente, nesta versão do compilador, as anotações de tipos serão meramente informativas, servindo como documentação do código. Todos os argumentos serão inferidos e validados por meio do sistema de tipos da representação intermediária.

3.3 EFEITOS ALGÉBRICOS

Para aumentar o poder da linguagem, foram adicionados construtores de efeitos algébricos ($effect$) e *handlers* ($handler$) para esses efeitos, conforme mostrado na Seção 3.1. A implementação seguiu o que está proposto no trabalho de Leijen (2016), porém foram necessárias algumas adaptações para que se pudesse agregar esses elementos ao cálculo que foi gerado para a linguagem.

Formalmente, efeitos algébricos são equivalentes a mônadas livres, onde uma mônada livre é uma mônada derivada diretamente de um endofuntor. Dentro do ecossistema da linguagem desse trabalho, foram definidos construtores para os tipos de efeitos algébricos. Os construtores de tipos de efeitos funcionam de forma semelhante aos tipos algébricos existentes em Haskell e ML. Para isso foi definida uma sintaxe que permita a declaração desses efeitos. Essa sintaxe é inspirada pela proposta em Koka e interfaces em linguagens como TypeScript e Java, conforme o exemplo do efeito `Exception`:

```
effect Exception {
  function throw(string): unit;
}
```

O tipo da função `throw` é:

```
∀ a.string → <Exception, a> unit
```

Conforme essa declaração, o efeito `Exception` é composto de um operação de efeito (*effectful operation*), nominalmente o `throw`. Essa operação recebe uma *string* qualquer como entrada e retorna `unit` executando o efeito `Exception` que está sendo declarado. Esse efeito pode ser utilizado em computações que possam causar algum tipo de problema, que tenham como parâmetros valores que não devem ser aceitos, por exemplo. A Figura 41 exemplifica esse cenário.

Figura 41 – Exemplo - Função com uso de efeito algébrico

```
1 algorithm safeDiv(int x, int y) {
2     if(y == 0)
3         throw("can't divide by zero");
4     return x / y;
5 }
```

Fonte: O Autor

A função `safeDiv` apenas verifica se o valor de `y` é igual a 0, e se sim irá emitir uma *exception*, e do contrário irá retornar o resultado. A inferência de tipos nesse caso irá retornar:

```
∀ a b.int → a int → <Exception, b> int
```

Serão recebidos 2 inteiros e retorna-se um inteiro. Quando essa função for aplicada, irá ser executado o efeito `Exception`, cujo significado ainda não foi definido. Com esses construtores, precedidos pela palavra reservada `effect`, é possível criar tipos de efeitos customizados e que tenham seu próprios comportamentos. A semântica da atuação dessas operações de efeito devem ser dadas por um *handler* de efeitos.

3.3.1 Effect Handlers

Um *handler* de efeitos é utilizado para se trabalhar com efeitos colaterais de modo que seja possível definir um procedimento para eles, efetivamente dando significado a uma declaração de efeitos. É possível, através da definição de um *handler*, purificar uma computação que carregue efeitos colaterais, similar a forma que mônadas são eliminadas em Haskell. O exemplo mais comum desse cenário envolve o tratamento de exceções, conforme exemplificado na subseção anterior. A forma mais habitual de se remover a exceção de uma computação é utilizando as estruturas de *try/catch*. A Figura 42 apresenta um *handler* para o efeito `Exception` análoga ao uso de *catch*.

Figura 42 – Exemplo - Handler para o efeito *Exception*

```
1 handler catchExc() {
2   pure x: {
3     return just(x);
4   }
5   case throw(str): {
6     return nothing();
7   }
8 }
```

Fonte: O Autor

O *handler* `catchExc` define um comportamento que permite purificar uma computação que contenha como tipo de efeito uma `Exception`. Conforme abordado na fundamentação teórica, o tipo *Maybe* (Subseção 2.2.1.7.3) é aplicado, entre outras situações, para se lidar com tratamento de exceções. O resultado da computação original, chamado de `x` no exemplo acima, é encapsulado dentro do tipo `Maybe` por meio da função `just`. Tal código só irá ser chamado caso a função `throw` não for chamada no código original; se ela for, conforme o código acima, o código original será interrompido e `nothing` irá retornar no seu lugar. O tipo do *handler* `catchExc` é:

```
∀ a b.(unit → <Exception, b> a) → b Maybe<a>
```

Em outras palavras, a computação original é convertida para o tipo *Maybe*, purificando a computação (removendo o efeito `Exception` dela). Efeitos algébricos permitem que apenas seja definida a interface de operação, o tipo de efeito em si, a retirada do efeito colateral é delimitada pela definição de seu *handler*. Há de se salientar também, que é possível existir mais de um *handler* para um efeito, como no caso do efeito `Exception`. A Figura 43 exibe um segundo *handler* para `Exception`.

Figura 43 – Exemplo - 2º Handler para o efeito *Exception*

```
1 handler ignoreException() {
2   pure x: {
3     return x;
4   }
5   case throw(str): {
6     return resume();
7   }
8 }
```

Fonte: O Autor

Esse segundo *handler* executa o restante da computação, retornando para o ponto onde `thrown` é chamado. O tipo extraído de `ignoreException` é dado por:

```
∀ a b.(unit → <Exception, a> b) → a b
```

A informação do tipo para o *handler* `ignoreException` mostra que essa função, permite a remoção do efeito `Exception`. A computação é continuada através da chamada de *keyword* `resume`.

A regra para a eliminação de efeitos é fornecida no artigo Leijen (2016) pela regra de checagem de tipos. Conforme o próprio artigo, uma regra de inferência pode ser facilmente derivada a partir da regra de checagem de tipos fornecida (o artigo focado na inferência Leijen (2014), por exemplo, não apresenta tal regra). A implementação se provou simples, embora trabalhosa, para inferir tais tipos; o ponto chave foi a necessidade da utilização de unificações para cada ponto da regra de checagem de tipos onde dois tipos (ou efeitos) iguais foram utilizados.

3.4 RESULTADOS DA IMPLEMENTAÇÃO

O principal resultado obtido por meio dessa pesquisa é a demonstração de que é possível aplicar um algorítimo de tipos e efeitos em uma linguagem com sintaxe imperativa. Para elucidar de forma mais assertiva esses desfechos, o programa que calcula o enésimo número da sequncia Fibonacci, exibido na Figura 44, é um exemplo de função pura escrita com sintaxe imperativa.

Figura 44 – Exemplo - Cálculo do enésimo número da sequência de Fibonacci

```
1  algorithm fibonnaci(int n) {
2      int x = 0;
3      int y = 1;
4
5      for(int z = 3; z < n; z++) {
6          int aux = x + y;
7          x = y;
8          y = aux;
9      }
10
11     return y;
12 }
13
14 algorithm main(int n) {
15     print(fibonnaci(n));
16 }
```

Fonte: O Autor

De forma simplificada, a função `fibonnaci` recebe como parâmetro um número qualquer, efetua o cálculo e retorna o resultado. A função `main` chama a função `fibonnaci` executando-a e imprimindo o resultado na tela. O tipo inferido da função `main` é:

```
∀ a.int → <console, a> unit
```

A função recebe um inteiro qualquer e retorna uma `unit` que executa o efeito `console`. É importante salientar que diferentemente de Koka, que insere implicitamente uma operação de desencapsulamento para remover o efeito (equivalente ao `runST` de Haskell), purificando o efeito de estado (mônada `ST` do Haskell) da computação, aqui o código é de fato puro. A mutabilidade local se transforma em passagem de parâmetros na geração do grafo (lembrando que as funções ϕ são equivalentes a funções lambda), e por isso não pode ser classificada como efeito colateral. O termo lambda gerado para esse programa pode ser interpretado como código Haskell válido, e é aceito por um compilador de Haskell, como é o exemplo da função `fibonnaci`, conforme visto na Figura 45.

Figura 45 – Exemplo - Fibonnaci convertido para representação funcional

```
\n0 ->
  let b2 _ =
      let x1 = 0 in
      let b3 _ =
            let y1 = 1 in
            let b4 _ =
                  let aux1 = 0 in
                  let b5 _ =
                        let z1 = 3 in
                        let b6 z2 y2 x2 aux2 _ =
                              let b7 _ =
                                    let aux3 = x2 + y2 in
                                    let b8 _ =
                                          let x3 = y2 in
                                          let b9 _ =
                                                let y3 = aux3 in
                                                let b10 _ =
                                                      let z3 = z2 + 1 in
                                                      (((((b6 z3) y3) x3) aux3)())
                                                in (b10 ())
                                          in (b9 ())
                                    in (b8 ())
                              b11 _ = y2
                        in if z2 < n0
                              then (b7 ())
                              else (b11 ())
                  in (((((b6 z1) y1) x1) aux1) ())
            in (b5 ())
      in (b4 ())
  in (b3 ())
in (b2 ())
```

Fonte: O Autor

O código gerado pelo compilador poderá ser estendido, com algumas adaptações, para poder utilizar como parâmetros outras funções, podendo construir programas com funções de ordem superior. Outra característica implementada na linguagem é a utilização de passagem de referências por valor utilizando o efeito `ST` e valores do tipo `ref`. A Figura 46 mostra uma operação que faz a troca de 2 valores.

A função `swap` recebe 2 valores e efetua a troca da posição deles. A referência

Figura 46 – Exemplo - Função swap

```
1  algorithm swap(var y, var x) {
2    var aux = *y;
3    *y = *x;
4    *x = aux;
5  }
6
7  algorithm main() {
8    var x = ref 10;
9    var y = ref 20;
10   swap(x, y);
11   print(*x);
12   print(*y);
13 }
```

Fonte: O Autor

da variável `y` é passada para `aux` e em seguida recebe a referência da variável `x`. Por último, a referência `x` recebe o valor advindo de `aux`. Dentro de `main`, são definidas as variáveis de `x` e `y` que receberão as novas referências 10 e 20 respectivamente. Após isso será feita a troca na chamada de `swap` imprimindo, ao término da computação, os valores contidos nas referências `*x` e `*y`. Os tipos inferidos para esses fragmentos de código são:

```
swap :: ∀a b c.ref<b> → c ref<b> → <st, a> unit
main :: ∀a b.a → <st, console, b> unit
```

Nota-se que conforme mencionado, Koka iria remover o efeito `st` da função `main` pois adiciona nos pontos de generalização uma chamada implícita ao *handler* de referências. Por simplicidade o algorítimo no protótipo não insere um mecanismo semelhante automaticamente, e por isso a função `main` também está executando o efeito `st`.

4 CONCLUSÃO

Dentre os diversos paradigmas de programação existentes, o paradigma imperativo é, discutivelmente, o mais utilizado comercialmente. Nesse cenário é possível afirmar que as linguagens abrigadas por esse paradigma são mais conhecidas e populares, fazendo delas, normalmente, a porta de entrada para a programação como um todo. Embora exista essa popularidade entre os programadores, linguagens imperativas podem apresentar empecilhos em sua utilização. Alguns desses problemas tem relação com a mutabilidade e computações que envolvam efeitos colaterais, como a alteração de estados, por exemplo.

Linguagens de programação funcionais, apesar de menos populares, se comparadas às imperativas, trazem algumas facilidades no que diz respeito a garantias e otimizações. Porém a sintaxe dessas linguagens é normalmente muito diferente de linguagens de programação imperativa. Isso pode fazer com que sejam preteridas a linguagens imperativas em geral. Haskell, por exemplo, não possui estruturas de controle para laços, o que pode causar uma má impressão ao programador habituado a linguagens imperativas.

Linguagens de programação com sintaxe imperativa e que compilam para um cálculo funcional podem ser encontradas na literatura. Um exemplo é a linguagem de programação SAC (SCHOLZ, 1994) que tem sua sintaxe baseada no padrão da linguagem C e é uma linguagem puramente funcional, mas não apresenta recursos para inferência de tipos e efeitos. Esses mecanismos podem ser muito úteis para auxiliar o programador a codificar com menor incidência de erros em tempo de execução. Por isso, a ideia de ser aplicar algoritimos de garantias de tipos e efeitos se torna interessante. Pode-se também abrir a possibilidade de se aplicar outras otimizações de compiladores imperativos e funcionais a essa representação funcional.

Embora sejam métodos existentes na literatura, os algoritimos empregados na presente pesquisa, e abordados no Capítulo 3, não são encontrados utilizados em conjunto no mesmo escopo, na literatura. As traduções e análises executadas no processo de compilação foram implementadas em um ecossistema totalmente funcional. Isso trouxe alguns problemas no desenvolvimento do compilador, gerando um atraso no cronograma de implementação.

Esses problemas se concentraram principalmente na extração das informações de dominância dos grafos de fluxo de controle. Isso deve-se ao fato de que o algorítimo para os cálculos envolvidos nessa retirada são projetados para linguagens imperativas, havendo a necessidade de muitas adaptações. Foi despendido muito es-

força para fazer uma adaptação fiel, sem muito sucesso, havendo a necessidade de se recorrer a pacotes externos para concluir a etapa.

Além das dificuldades que envolvem a adaptação de um algorítimo imperativo para um paradigma funcional, a tradução do SSA/CFG para o cálculo lambda proposto também teve alguns obstáculos. Sendo necessárias algumas adaptações ao que se encontra na literatura. A mais gritante, que também pode ser considerada uma contribuição dessa pesquisa, foi a adição do *where*, algo que não ocorre no trabalho de Appel (1998). Isso faz com que o cálculo lambda gerado a partir do SSA se torne mais legível e próximo do que se encontra em Haskell. Atualmente, programas, que não tenham efeitos colaterais em sua estrutura, escritos e compilados pela proposta desse trabalho geram códigos Haskell válidos. Como trabalho futuro, pode-se utilizar mecanismos propostos em Leijen (2014) para se transformar qualquer código escrito na linguagem proposta em código Haskell válido.

A implementação do sistema de tipos e efeitos para a linguagem proposta encontrou alguns problemas também. O maior deles ocorreu por conta de um erro de digitação em uma das definições de algoritimos (Figura 13) propostas no trabalho de Leijen (2014). Isso fez com que fosse necessária uma série de testes e verificações extras na implementação do algoritmo de inferência de tipos. De modo geral, a implementação seguiu exatamente o que foi proposto na literatura e permitiu um grande aprendizado sobre *effect rows*.

A inserção de meios para tratar e combinar efeitos em sistemas de tipos, permite a criação de códigos mais seguros. Outro detalhe relevante é que, se comparada a Haskell, a combinação de efeitos de Koka, e da linguagem proposta, é muito menos verbosa, o que torna os códigos mais legíveis e mais fáceis de receberem manutenção. A proposta desse projeto limitou-se a tratar a combinação de efeitos; como trabalho futuro, fica a investigação da possibilidade de se combinar outras mônadas compiladas a partir de uma linguagem de programação imperativa.

Foram escolhidos efeitos algébricos para o trabalho pois a combinação destes ocorre de forma natural. Isso remete ao fato de que código é capaz de inferir diversos efeitos, baseado no uso, pra mesma função. Esse cenário já pode ser encontrado na literatura, porém pode ser viável a construção de uma espécie de *framework* geral para trabalhar com códigos monádicos.

Outro fator a ser apurado está contido na verificação dos impactos, na prática, da escrita de códigos em estilo imperativo dentro de uma linguagem funcional; isso realmente ajuda no processo de criação de programas? O desenvolvimento de aplicações com essa abordagem é realmente eficiente? Auxilia o programador a se habituar com uma linguagem de programação puramente funcional? A pesquisa conteve-se a

apenas demonstrar que é possível inferir efeitos colaterais de linguagens que façam uso de fluxos de controle em sua compilação. Ficando como trabalho futuro averiguar esse cenários em situações reais e com uma boa amostragem.

O objetivo desse trabalho foi demonstrar empiricamente que é possível traduzir construções comuns em linguagens imperativas para um código funcional puro, explorando a correspondência entre a representação intermediária de código SSA e linguagens funcionais. Também foi demonstrado que o algoritmo de inferência de tipos e efeitos proposto para a linguagem Koka pode ser usado, com algumas modificações, para inferir tipos de funções escritas em sintaxe imperativa. Essas alterações podem ser vistas também como contribuições da pesquisa.

Foi constatado que a mutabilidade local de variáveis não altera a pureza do programa, visto que estruturas de fluxo de controle podem ser convertidas em funções puras mutuamente recursivas. Sendo possível tratar como código funcional estruturas de repetição como *while* e *for* ou, até mesmo, comandos de desvio como *goto* que, até onde o autor está ciente, não é encontrado em nenhuma linguagem de programação puramente funcional. A inferência de tipos na representação intermediária mostrou-se capaz de detectar os possíveis efeitos gerados por uma função e as situações em que uma função é pura, ou seja, não propaga efeitos colaterais além de seu escopo, espera-se que esse resultado seja um passo na direção de fornecer maiores garantias em linguagens imperativas.

Diversos trabalhos futuros, além dos listados no capítulo anterior, podem ser desenvolvidos a partir da pesquisa proposta: A especialização do algoritimo Damas-Milner para atuar diretamente nos blocos, sem ter que traduzi-los para uma representação funcional e suas as provas de consistência e completude. Também, como trabalho futuro, é interessante compilar o código para ser aceito pela *LLVM* (ou outra forma de *bytecode*) tornando-o portável. Por último, acrescentar mais estruturas à linguagem (*switch*, tipos algébricos, mais estruturas de dados, sobrecarga, etc.).

REFERÊNCIAS

ABEL, A. foetus–termination checker for simple functional programs. **Programming Lab Report**, v. 474, 1998.

AHMAN, D. Handling fibred algebraic effects. **Proceedings of the ACM on Programming Languages**, ACM, v. 2, n. POPL, p. 7, 2017.

ALLEN, C.; MORONUKI, J. Haskell programming from first principles. **Gumroad (ebook)**, 2017.

APPEL, A. W. Ssa is functional programming. **ACM SIGPLAN Notices**, ACM, v. 33, n. 4, p. 17–20, 1998.

CARDELLI, L. Type systems. **ACM Computing Surveys**, v. 28, n. 1, p. 263–264, 1996.

CHURCH, A. **The calculi of lambda-conversion**. [S.l.]: Princeton University Press, 1941.

COOPER, K.; TORCZON, L. **Engineering a compiler**. [S.l.]: Elsevier, 2011.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 13, n. 4, p. 451–490, 1991.

DAMAS, L.; MILNER, R. Principal type-schemes for functional programs. In: ACM. **Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.], 1982. p. 207–212.

DORNAN, C.; JONES, I. **Alex User Guide**. 2003.

GASTER, B. R.; JONES, M. P. A polymorphic type system for extensible records and variants. **Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham**, 1996.

HILLERSTRÖM, D.; LINDLEY, S. Liberating effects with rows and handlers. In: ACM. **Proceedings of the 1st International Workshop on Type-Driven Development**. [S.l.], 2016. p. 15–27.

HUDAK, P. et al. A history of haskell: being lazy with class. In: ACM. **Proceedings of the third ACM SIGPLAN conference on History of programming languages**. [S.l.], 2007. p. 12–1.

HUTTON, G. **Programming in Haskell**. [S.l.]: Cambridge University Press, 2016.

JONES, S. P. **Haskell 98 language and libraries: the revised report**. [S.l.]: Cambridge University Press, 2003.

JR, G. L. S.; COMMON, L. the language. **Digital Press**, v. 20, p. 124, 1984.

KOTELNIKOV, E. Type-directed language extension for effectful computations. In: ACM. **Proceedings of the Fifth Annual Scala Workshop**. [S.l.], 2014. p. 35–43.

LEIJEN, D. Extensible records with scoped labels. **Trends in Functional Programming**, v. 5, p. 297–312, 2005.

LEIJEN, D. Koka: Programming with row polymorphic effect types. **arXiv preprint arXiv:1406.2061**, 2014.

LEIJEN, D. **Algebraic Effects for Functional Programming**. [S.l.], 2016.

LEIJEN, D. Structured asynchrony with algebraic effects. In: ACM. **Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development**. [S.l.], 2017. p. 16–29.

LINDLEY, S.; MCBRIDE, C.; MCLAUGHLIN, C. Do be do be do. In: **POPL 2017 Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages**. [S.l.]: ACM, 2017. v. 4, n. 1, p. 500–514.

LONG, Y.; RAJAN, H. A type-and-effect system for asynchronous, typed events. In: ACM. **Proceedings of the 15th International Conference on Modularity**. [S.l.], 2016. p. 42–53.

MARLOW, S.; GILL, A. Happy user guide. **Glasgow University, December**, 1997.

MILNER, R. et al. **The definition of standard ML: revised**. [S.l.]: MIT press, 1997.

MUCHNICK, S. S. **Advanced compiler design implementation**. [S.l.]: Morgan Kaufmann, 1997.

ORCHARD, D.; PETRICEK, T. Embedding effect systems in haskell. **ACM SIGPLAN Notices**, ACM, v. 49, n. 12, p. 13–24, 2015.

ROSSUM, G. V. et al. Python programming language. In: **USENIX Annual Technical Conference**. [S.l.: s.n.], 2007. v. 41, p. 36.

SCHOLZ, S.-B. Single assignment c-functional programming using imperative style. In: CITESEER. **In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia**. [S.l.], 1994.

STALLMAN, R. M. Gnu compiler. **collection internals**, GNU, 2005.

THOMPSON, S. **Haskell: the craft of functional programming**. [S.l.]: Addison-Wesley, 2011. v. 2.

TORRENS, P.; VASCONCELLOS, C.; GONÇALVES, J. A hybrid intermediate language between ssa and cps. In: ACM. **Proceedings of the 21st Brazilian Symposium on Programming Languages**. [S.l.], 2017. p. 1.

TUCKER, A. B. **Programming languages**. [S.l.]: Tata McGraw-Hill Education, 1986.

TURNER, D. A. Miranda: A non-strict functional language with polymorphic types. In: SPRINGER. **Conference on Functional Programming Languages and Computer Architecture**. [S.l.], 1985. p. 1–16.

WADLER, P. The essence of functional programming. In: ACM. **Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.], 1992. p. 1–14.

WADLER, P.; THIEMANN, P. The marriage of effects and monads. **ACM Transactions on Computational Logic (TOCL)**, ACM, v. 4, n. 1, p. 1–32, 2003.

APÊNDICE A – DEFINIÇÃO DA GRAMÁTICA DA LINGUAGEM PROPOSTA

⟨*program*⟩ ::= ⟨*top-level*⟩+

⟨*top-level*⟩ ::= ⟨*algorithm*⟩ | ⟨*effect*⟩ | ⟨*handler*⟩

⟨*algorithm*⟩ ::= '`algorithm`' ⟨*id*⟩ ⟨*param-list*⟩ ⟨*body*⟩

⟨*param-list*⟩ ::= '`(`' ⟨*declarator*⟩* '`)`'

⟨*declarator*⟩ ::= '`var`' ⟨*id*⟩ | ⟨*type*⟩ ⟨*id*⟩

⟨*type*⟩ ::= '`int`' | '`bool`' | '`string`'

⟨*body*⟩ ::= '`{`' ⟨*statement*⟩* '`}`'

⟨*statement*⟩ ::= ⟨*assignment-stmt*⟩
| ⟨*call-stmt*⟩
| ⟨*if-stmt*⟩
| ⟨*while-stmt*⟩
| ⟨*for-stmt*⟩
| ⟨*goto-stmt*⟩
| ⟨*return-stmt*⟩
| ⟨*label-stmt*⟩
| ⟨*break-stmt*⟩
| ⟨*continue-stmt*⟩
| ⟨*decl-stmt*⟩

⟨*if-stmt*⟩ ::= ⟨*if*⟩ ⟨*else*⟩?

⟨*if*⟩ ::= '`if`' '`(`' ⟨*expr*⟩ '`)`' ⟨*body*⟩
| '`íf`' '`(`' ⟨*expr*⟩ '`)`' ⟨*statement*⟩

⟨*else*⟩ ::= '`else`' ⟨*body*⟩ | '`else`' ⟨*statment*⟩

⟨*assignment-stmt*⟩ ::= ⟨*assignment*⟩ ';' | ⟨*assignment-ref*⟩ ';'

⟨*assignment*⟩ ::= ⟨*id*⟩ '=' ⟨*id*⟩ | ⟨*id*⟩ '++' | ⟨*id*⟩ '–'

⟨*call-stmt*⟩ ::= ⟨*id*⟩ '(' ⟨*expr-list*⟩* ')'

⟨*expr-list*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ',' ⟨*expr-list*⟩

⟨*while-stmt*⟩ ::= 'while' '(' ⟨*expr*⟩ ')' ⟨*body*⟩
| 'while' '(' ⟨*expr*⟩ ')' ⟨*statement*⟩

⟨*for-stmt*⟩ ::= 'for' '(' ⟨*decl-stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*assignment-stmt*⟩ ')' ⟨*body*⟩
| 'for' '(' ⟨*assignment-stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*assignment-stmt*⟩ ')'
| 'for' '(' ⟨*decl-stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*assignment-stmt*⟩ ')' ⟨*statement*⟩
| 'for' '(' ⟨*assignment-stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*assignment-stmt*⟩ ')' ⟨*statement*⟩

⟨*return-stmt*⟩ ::= 'return' ⟨*expr*⟩ ';' | 'return' ';'

⟨*goto-stmt*⟩ ::= 'goto' ⟨*id*⟩ ';'

⟨*break-stmt*⟩ ::= 'break' ';'

⟨*continue-stmt*⟩ ::= 'continue' ';'

⟨*decl-stmt*⟩ ::= ⟨*declarator*⟩ ';' | ⟨*declarator*⟩ '=' ⟨*declarator*⟩

⟨*label-stmt*⟩ ::= ⟨*id*⟩ ':' ⟨*statement*⟩
| ⟨*case*⟩ ⟨*number*⟩ ':' ⟨*statement*⟩
| ⟨*default*⟩ ':' ⟨*statement*⟩

⟨*expr*⟩ ::= ⟨*id*⟩
| ⟨*number*⟩
| ⟨*string*⟩
| ⟨*true*⟩
| ⟨*false*⟩
| ⟨*id*⟩ ⟨*expr-list*⟩*
| ⟨*expr*⟩ '+' ⟨*expr*⟩
| ⟨*expr*⟩ '-' ⟨*expr*⟩
| ⟨*expr*⟩ '*' ⟨*expr*⟩
| ⟨*expr*⟩ '/' ⟨*expr*⟩
| ⟨*expr*⟩ '==' ⟨*expr*⟩
| ⟨*expr*⟩ '!=' ⟨*expr*⟩
| 'ref' ⟨*expr*⟩
| '*' ⟨*expr*⟩
| '-' ⟨*expr*⟩
| '(' ')'
| '(' ⟨*expr*⟩ ')'

⟨*effect*⟩ ::= '`effect`' ⟨*id*⟩ '{' ⟨*effect-decl*⟩+ '}'

⟨*effect-decl*⟩ ::= '`function`' ⟨*id*⟩ '(' ⟨*type*⟩+ ')' ⟨*effect-return*⟩?

⟨*effect-return*⟩ ::= ‘`:`’ ⟨*type*⟩

⟨*handler*⟩ ::= ‘`handler`’ ⟨*id*⟩ ‘`(`’ ‘`)`’ ‘`{`’ ⟨*handler-body*⟩+ ‘`}`’

⟨*handler-body*⟩ ::= ‘`pure`’ ‘`:`’ ⟨*body*⟩ | ‘`case`’ ⟨*id*⟩ ‘`(`’ ‘`)`’ ‘`:`’ ⟨*body*⟩