Interpretando Efeitos Algébricos por Meio de Mônadas

Karla A. de S. Joriatti
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brasil
kjoriatti@gmail.com

Paulo H. Torrens
Universidade de Kent
Canterbury, Reino Unido
paulotorrens@gnu.org

Rodrigo Geraldo Ribeiro
Departamento de Computação, Universidade Federal de Ouro Preto
Ouro Preto, Brasil
rodrigo.ribeiro@ufop.edu.br

Cristiano Damiani Vasconcellos
Departamento de Ciência da Computação, Universidade do Estado de Santa Catarina
Joinville, Brasil
cristiano.vasconcellos@udesc.br

RESUMO

Kelsey demonstrou que a forma de atribuição estática única, comumente utilizada como uma linguagem intermediária para a implementação de otimizadores de código em compiladores de linguagens imperativas, pode ser caracterizada como uma linguagem funcional. Isso ocorre porque a mutabilidade local é eliminada por meio da criação de novas variáveis, e o fluxo de controle pode ser traduzido em abstrações lambda. A tradução dessa representação intermediária para a forma A-normal, frequentemente utilizada em compiladores para linguagens funcionais, é relativamente direta. Rigon et al. exploram essa característica, propondo um sistema que infere tipos e efeitos para uma linguagem com mecanismos de controle de fluxo similares aos de linguagens imperativas. Este trabalho expande essa proposta ao introduzir a tradução da representação intermediária na forma A-normal com efeitos algébricos para uma representação que lida com efeitos por meio de mônadas, onde a combinação de vários efeitos em uma mônada é representada por mônadas livres. Como resultado, é demonstrada a geração de código executável em linguagem *Haskell*, oferecendo indiretamente um meio simples para a definição de uma semântica para a forma de atribuição estática única.

PALAVRAS-CHAVE

efeitos algébricos, mônadas livres, ANF, linguagens de programação

1 Introdução

Sistemas de tipos e efeitos foram propostos por Gifford and Lucassen [5] com o objetivo de identificar, de forma estática, as regiões do código em que ocorrem efeitos colaterais. Essa identificação permite determinar expressões livres de efeitos e tinha como motivação a possibilidade de auxiliar na tomada de decisões relacionadas à paralelização do código [12]. Com o crescente interesse por linguagens funcionais, tais sistemas vêm ganhando relevância no projeto de linguagens que buscam garantir a pureza de funções. Um exemplo é a linguagem *Koka* [9], uma linguagem funcional pura com avaliação estrita e sintaxe inspirada em *JavaScript*, que possui um sistema de tipos capaz de inferir tipos e efeitos.

Inspirado por tais sistemas e explorando o fato de que a representação intermediária de código na forma de atribuição estática única (SSA[1]) corresponde a uma linguagem funcional [1, 6], Rigon et al. [17] propõe a inferência de tipos e efeitos para código em SSA apresentando um sistema de tipos inspirado na linguagem *Koka*.

O SSA [3] é uma linguagem intermediária, comumente usada na fase de otimização de compiladores para linguagens imperativas, que impõe a restrição de que as variáveis sejam atribuídas uma única vez e que suas definições devam preceder o seu uso. O código é representado por um grafo de fluxo de controle que especifica todos os caminhos de execução possíveis.

Rigon et al. [17] define a tradução de código em SSA para a forma A-normal (ANF[2]) e um sistema de tipos e efeitos, bastante similar ao de *Koka*, capaz de inferir tipos e efeitos para o código em ANF, mostrando empiricamente que é possível inferir tipos e efeitos para código em SSA. A ideia deste trabalho é apresentar um algoritmo capaz de representar os efeitos inferidos para o código em ANF por meio de mônadas, o que indiretamente fornece um meio simples para a definição de uma semântica para a linguagem em SSA.

A passagem de código que utiliza efeitos algébricos para uma abordagem monádica já foi realizada anteriormente no trabalho de Vazou and Leijen [20], onde a composição entre efeitos ocorre de forma semelhante ao que é observado na biblioteca de transformadores de mônadas em *Haskell*. Contudo, a composição de mônadas pode ser uma tarefa complexa, especialmente ao lidar com muitos efeitos [13]. Dessa forma, este trabalho propõe a utilização do conceito de mônadas livres [19] por meio da biblioteca *extensible-effects* para simplificar a tradução da composição de efeitos.

2 Overview

Com o objetivo de apresentar a tradução de um código em ANF utilizando um sistema de tipos e efeitos para uma versão em cálculo lambda com instanciação de mônadas livres (que representa *Haskell*), esta seção discute os conceitos fundamentais de ANF, efeitos algébricos e de mônadas.

2.1 Forma A-Normal

A ANF é uma variante do cálculo lambda por valor, o qual foi proposto por Plotkin [16], frequentemente utilizada como representação intermediária em compiladores. Nessa forma, não é mais possível realizar reduções A, conforme definido por Sabry and Felleisen [18].

[1] Do inglês, *static single assignment.*

[2] Do inglês, A-normal form

Para caracterizar o conjunto de reduções A, é necessário introduzir o conceito de contexto, cuja definição é a usual. Um contexto é um termo com um "buraco", representado por "[]", que pode ser preenchido por uma subexpressão. Esse preenchimento, denominado *filling*, ocorre entre um contexto e um termo, resultando em um novo termo. O processo pode capturar variáveis livres, por exemplo, o preenchimento do contexto $(\lambda x.y[])$ com o termo $(\lambda z.x)$ resulta em $(\lambda x.y(\lambda z.x))$, onde a variável x foi ligada.

Com base nessa definição, Sabry and Felleisen [18] propõem o conjunto de reduções que compõem as chamadas reduções A, representadas na Figura 1.

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

Figura 1: Conjunto de Reduções A, de Sabry and Felleisen [18].

Esse conjunto é correspondente às reduções administrativas aplicadas em programas escritos em CPS (*Continuation-Passing Style*). A primeira regra corresponde à redução η por valor; a segunda eleva o contexto ao corpo da função (*lifting*); a terceira reestrutura aplicações aninhadas, permitindo o *bind* de (MN) antes de sua aplicação; a quarta redução elimina aplicações de funções identidade; e a última é uma adaptação da tradicional redução β do cálculo lambda, utilizando contextos.

A linguagem ANF utilizada em compiladores é obtida a partir da restrição da gramática do cálculo lambda a termos que já se encontram na forma A-Normal. Essa gramática pode ser descrita como:

$$
\begin{array}{l}
e ::= v \\
\quad \mid \textbf{let}\ x\ = v\ \textbf{in}\ e \\
\quad \mid \textbf{let}\ x\ = v\ v\ \textbf{in}\ e \\
\\
v ::= x \\
\quad \mid \lambda x.e
\end{array}
$$

Além disso, Flanagan et al. [4] demonstram a equivalência entre máquinas abstratas baseadas em CPS e ANF, enquanto Sabry and Felleisen [18] mostram a correspondência entre ANF e o cálculo lambda computacional de Moggi [14].

2.2 Efeitos Algébricos

A utilização de efeitos algébricos tem ganhado destaque como uma abordagem puramente funcional para modelar efeitos colaterais de forma modular [10], promovendo uma separação explícita entre operações e seus respectivos tratadores (ou *handlers*). Nesse contexto, linguagens como *Eff* [2], desenvolvida com o propósito de investigar efeitos algébricos, e *Koka* [9], que incorpora efeitos diretamente no sistema de tipos, representam alternativas à abordagem tradicional baseada em mônadas. Adicionalmente, efeitos algébricos semelhantes aos utilizados em *Koka* podem ser empregados como base para representações intermediárias funcionais derivadas de representações na forma SSA [17].

Em *Koka*, o sistema de tipos é estendido para incluir a inferência de efeitos adotando sentenças da forma $\Gamma \vdash e : \sigma|\epsilon$, em que uma expressão e tem tipo σ e produz um efeito ϵ em um dado contexto Γ. O sistema também admite efeitos polimórficos, onde o efeito resultante de uma função pode ser determinado pelos efeitos de seus parâmetros. Um exemplo clássico é a função *map*, responsável por aplicar uma função a cada elemento de uma lista. Dado que α e β são tipos arbitrários e μ representa um efeito genérico, o tipo de *map* é expresso como:

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

Nesse tipo é possível observar que *map* recebe uma lista de elementos do tipo α e uma função com entrada α e saída β, sujeita ao efeito μ. Assim, o efeito da função mapeada é propagado pela aplicação de *map*, resultando em um valor do tipo $\mu\ list\langle\beta\rangle$.

Outra característica central do sistema de tipos e efeitos de Leijen [9] é o uso de *effect-rows*, que representam a união disjunta de múltiplos efeitos em uma computação. Por exemplo, uma computação que envolva efeitos de ambiguidade (*amb*) e exceção (*exn*) terá seu efeito inferido como o *effect-row* $\langle amb,\ exn\rangle$ (equivalente, também, a $\langle exn,\ amb\rangle$).

Além dos efeitos nativos da linguagem, *Koka* permite a definição de novos efeitos por meio de operações. O efeito de ambiguidade, por exemplo, pode ser especificado conforme segue:

```
effect amb {
    ctl flip () : bool
}
```

A operação de controle *flip* é responsável por retornar todas as possibilidades de valores booleanos (*True* e *False*). Essa operação é definida como parte do efeito *amb* e possibilita a implementação de funções como *xor*, cujo tipo inferido é *amb bool*, conforme mostrado a seguir:

```
fun xor () {
    val p = flip ()
    val q = flip ()
    (p || q) && not(p && q)
}
```

Funções com efeitos definidos dessa forma devem ser tratadas dinamicamente por meio de *handlers*, os quais, no entanto, não são abordados neste trabalho.

Desse modo, *Koka* permite a criação modular de efeitos, *handlers* e funções com múltiplos efeitos, evidenciando a expressividade e flexibilidade dessa abordagem. Uma alternativa conceitual à mesma problemática pode ser encontrada no uso de mônadas livres.

2.3 Mônadas

Introduzido ao contexto da programação por Wadler [21], o conceito de mônada tem origem na teoria das categorias. Em linguagens funcionais, uma mônada fornece um mecanismo sistemático para representar sequências de computações, propagando implicitamente as alterações de estado e contexto em um determinado tipo de dado.

Embora o uso de mônadas contribua para modularizar a estrutura de programas em linguagens funcionais puras, tornando o

fluxo de dados menos explícito e verboso, a composição de mônadas apresenta desafios práticos. Lüth and Ghani [13] discutem as dificuldades na composição de mônadas e sugerem abordar o problema por meio de coprodutos [3]. As mônadas livres (*free monads*), que propõem uma abordagem mais flexível para representação e combinação de efeitos, são inspiradas nessa abordagem.

Outra alternativa, adotada na linguagem *Haskell* para lidar com a composição de efeitos representados por meio de mônadas, são os transformadores de mônadas (*monad transformers*), que possibilitam empilhar diferentes efeitos monádicos.

Nesta seção, será apresentada uma visão geral dos transformadores de mônadas, bem como uma introdução ao uso de mônadas livres como modelo alternativo para a composição modular de efeitos computacionais.

2.3.1 Transformadores de Mônadas. Uma das abordagens mais consolidadas para a combinação de efeitos em linguagens funcionais puras é o uso de transformadores de mônadas. Introduzido por Liang et al. [11], os transformadores de mônadas permitem a composição de efeitos computacionais clássicos, como estado, erro e entrada/saída, por meio da definição de estruturas de combinação em camadas.

A ideia central consiste na criação de novos tipos de dados, como *MaybeT*, *StateT* ou *EitherT*, que encapsulam um efeito específico e permitem sua combinação com qualquer outra mônada. Para viabilizar essa combinação, a classe *MonadTrans* define a função *lift*, cuja responsabilidade é elevar uma computação monádica simples ou inferior para um contexto de combinação com mais de um efeito. O tipo de *lift* é apresentado a seguir.

$$lift :: (MonadTrans\ t,\ Monad\ m) \Rightarrow m\ a \rightarrow t\ m\ a$$

No entanto, essa abordagem apresenta algumas limitações práticas. A combinação de múltiplos efeitos exige empilhar transformadores, o que pode resultar em estruturas complexas e na necessidade de múltiplos *lifts* para acessar dados mais internos na pilha. Além disso, a ordem de composição dos transformadores é fixa, reduzindo a flexibilidade na manipulação de efeitos.

Apesar dessas limitações, a biblioteca *mtl* continua sendo amplamente utilizada, especialmente para combinar efeitos padrão. Por outro lado, propostas mais recentes, como *polysemy*, *freer-simple* e *extensible-effects*, oferecem modelos baseados em efeitos extensíveis, proporcionando maior flexibilidade e modularidade na composição de efeitos [8].

2.3.2 Mônadas Livres. Uma mônada livre é definida como a construção adjunta à esquerda de um *forgetful functor* de mônadas para seus respectivos funtores adjacentes [19]. Como funtores adjuntos à esquerda preservam coproduto, computar o coproduto de mônadas livres é reduzido a computar a mônada livre sobre o coproduto dos funtores adjacentes. Além disso, a assinatura de operações de efeitos algébricos gera uma álgebra livre, que, por sua vez, origina uma mônada livre [10].

Swierstra [19] descreve uma estrutura que, sendo instância da classe *Functor* e *Monad*, é capaz de elevar um funtor à condição de mônada. A estrutura é representada pelo tipo algébrico *Free*, inspirada na definição dada por Swierstra [19], declarada a seguir com a nomenclatura adotada atualmente [7].

```
data Free f a = Pure a
              | Free (f (Free f a))
```

Nessa definição, o tipo *Free* recebe um funtor f e um tipo a, e retorna uma computação pura (*Pure a*) ou uma computação impura (*Free (f (Free f a))*). Para ilustrar o funcionamento de uma mônada livre, alguns exemplos de código foram adaptados de Leijen [10]. O código a seguir apresenta a implementação, em *Haskell*, da mônada livre *Free ST*, para representar computações com estados.

```
data ST a = Get (Int -> a) | Put Int a
          deriving Functor

type FreeST = Free ST

get = Free (Get (\s -> Pure s))
put s = Free (Put s (Pure s))
```

Assim, *FreeST* representa a mônada livre sobre o funtor *ST*, e as funções de *get* e *put* elevam os construtores de *ST* à mônada *FreeST*. A seguir, um exemplo de computação monádica com *get* e *put*, além de um interpretador que executa a computação com estado é apresentado.

```
testST :: FreeST String
testST = do put 2
            x <- get
            put (x + 2)
            return "oi"

runST :: FreeST b -> Int -> (Int, b)
runST p s =
  case p of
    Pure x -> (s, x)
    Free (Get f) -> runST (f s) s
    Free (Put s' x) -> runST x s'
```

Dentro dessa abordagem, a combinação de efeitos é direta utilizando coprodutos. Por meio do tipo *Either*, os funtores *[]* (que modela não determinismo, representando uma situação de ambiguidade) e *ST* (que representa estados) são combinados e elevados à condição de mônada livre por meio da estrutura *Free*. O tipo *FlipST* representa o coproduto entre estado e ambiguidade. O código a seguir demonstra essa construção, onde a função *flip* gera todas as possibilidades de valores booleanos.

```
type FlipST = Free (Either [] ST)

flip = Free (Left [Pure True, Pure False])
get = Free (Right (Get (\s -> Pure s)))
put s = Free (Right (Put s (Pure s)))
```

A seguir, é apresentada uma computação monádica simples que combina os dois efeitos:

```
testSTAmb :: FlipST Int
testSTAmb = do p <- flip
               if p then put 1 else put 2
               y <- get
               return y
```

[3] Nas categorias dos conjuntos e dos tipos, o coproduto corresponde à união disjunta

Em mônadas livres a propagação implícita da computação é executada na função que interpreta a composição das mônadas, nesse exemplo a função *runSTAmb*, portanto é nessa função que uma semântica é atribuída ao código, no exemplo a mônada de estado é resolvida antes da que representa ambiguidade.

```
runSTAmb (Pure x) s = [(s, x)]
runSTAmb (Free xs) s = runEither xs s

runEither (Left xs) s = foldl fun [] xs
  where
    fun = \ys y -> ys ++ runSTAmb y s
runEither (Right (Get f)) s =
  runSTAmb (f s) s
runEither (Right (Put s x)) _ =
  runSTAmb x s
```

A escolha de uma ordem diferente para resolução das mônadas definiria uma semântica distinta para a expressão. O seguinte código apresenta a versão alternativa para interpretar a função de teste, onde a mônada de ambiguidade é resolvida primeiro.

```
runAmbST (Pure x) s = (s, [x])
runAmbST (Free xs) s = runEither xs s

runEither (Right (Get f)) s = runAmbST (f s) s
runEither (Right (Put s x)) _ = runAmbST x s
runEither (Left xs) s =
    foldl (\(a, b) y -> let (w, t) = runAmbST y a
                        in (w, b++t)) (s,[]) xs
```

Ao executar ambos os interpretadores, os resultados abaixo podem ser observados, onde se escolhe a ordem em que os efeitos são tratados:

```
> runSTAmb testSTAmb 0
[(1,1) , (2,2) ]
> runAmbST testSTAmb 0
(2, [1,2])
```

3 Desenvolvimento

Com o objetivo de apresentar a implementação da tradução proposta, esta seção descreve a biblioteca utilizada, bem como a formalização da tradução desenvolvida e a função de tradução criada.

3.1 Biblioteca Extensible Effects

Proposta como uma alternativa à biblioteca MTL de Haskell, a biblioteca *extensible effects* oferece uma abordagem para o tratamento de efeitos baseada nos conceitos de mônadas livres e *open union* [8]. Diferentemente das mônadas livres tradicionais, essa biblioteca permite que qualquer dado de *kind* $* \longrightarrow *$ seja elevado à condição de mônada por meio do construtor *Eff* [7].

Seguindo essa perspectiva, os autores destacam que a implementação do operador *bind* para uma mônada livre utiliza a função *fmap* para estender a continuação, como pode ser observado no código a seguir:

```
instance (Functor f) => Monad (Free f) where
    Free g >>= f = Free (fmap (>>= f) g)
```

Como essa extensão ocorre de maneira uniforme, é possível incorporar a continuação diretamente no construtor da mônada. Dessa forma, o operador *bind* atua como um acumulador de continuações, utilizando a composição de Kleisli (⋙).

```
data Freer f a where
    Pure :: a -> Freer f a
    Free :: f x -> (x -> Freer f a)
            -> Freer f a

instance Monad (Freer f) where
    Free fx k >>= k' = Freer fx (k >>> k')
```

Para adicionar efeitos extensíveis à *Freer*, utiliza-se a estrutura de dados abstrata *Union (r :: [* → *]) v*, que representa a união de requisições (efeitos) *r* sobre um tipo de dado *v*.

```
data EFreer r a where
    Pure :: a -> EFreer r a
    Free :: Union r x -> (x -> EFreer r a) -> EFreer r a
```

De modo a resolver questões de desempenho, a biblioteca define o tipo *Arr*, que representa uma função com efeito, e o tipo *FTCQueue*, que modela o acúmulo de continuações.

```
type Arr r a b = a -> Eff r b

type FTCQueue (m :: * -> *) a b
tsingleton :: (a -> m b) -> FTCQueue m a b
(⊳) :: FTCQueue m a x -> (x -> m b) -> FTCQueue m a b
(⋈) :: FTCQueue m a x -> FTCQueue m x b -> FTCQueue m
    a b
```

Contudo, o tipo *FTCQueue* representa sequências genéricas, sendo posteriormente especializado por meio do tipo *Arrs*, conforme apresentado abaixo:

```
type Arrs r a b = FTCQueue (Eff r) a b
```

Por fim, com base nas definições anteriores, é possível declarar a estrutura de dados *Eff*, juntamente com sua instância para a classe *Monad*.

```
data Eff r a where
  Pure :: a -> Eff r a
  Impure :: Union r x -> Arrs r x a -> Eff r a

instance Monad (Eff r) where
  return = Pure
  Pure x >>= k = k x
  Impure u q >>= k = Impure u (q >>> k)
```

Outra função importante disponibilizada pela biblioteca, e utilizada no desenvolvimento do presente trabalho, é a função *send*, responsável por converter um valor em uma computação do tipo *Eff*, atuando de forma análoga ao *lift*.

3.2 Formalização da Tradução

O primeiro passo da tradução consiste na definição de uma gramática de origem e outra para representar a linguagem de destino. Como a gramática de origem é fornecida pelo trabalho de Rigon et al. [17], é necessário apenas especificar a sintaxe de saída da função de tradução. Conforme mencionado na Seção 2.1, a linguagem

ANF é equivalente ao cálculo lambda computacional e, portanto, a sintaxe da linguagem-alvo modifica apenas alguns dos elementos da gramática de origem com base em aspectos da metalinguagem para o cálculo lambda computacional proposta por Moggi [15].

A Figura 2 apresenta a sintaxe da linguagem-alvo proposta neste trabalho. Essa linguagem separa instruções puras de instruções com efeitos, incorpora as regras de *bind* monádico e injeção monádica por meio do operador η, e representa a mônada sobre o tipo τ_2 por meio da notação $T(\tau_2)$.

$e ::=$	termos
$\mid e_1\ e_2$	aplicação
$\mid$ if v then e_2 else e_3	condicional
$\mid$ let $x = e_1$ in e_2	let bind
$\mid x \leftarrow e$	bind monádico
$\mid \eta(v)$	return
$v ::=$	valores
$\mid x$	variáveis
$\mid n$	números
$\mid b$	booleanos
$\mid s$	strings
$\mid \lambda x.e$	abstração
$\tau ::=$	tipos
$\mid \alpha$	variável
$\mid$ Int	inteiro
$\mid$ Bool	booleano
$\mid$ String	string
$\mid \tau_1 \rightarrow T(\tau_2)$	função

Figura 2: Sintaxe da linguagem alvo

3.3 Função de Tradução

Para efetuar uma tradução sintática, a função $\mathcal{F}$ recebe como entrada a gramática de origem e deve produzir código na sintaxe da linguagem alvo proposta. Essa tradução é inspirada pela tradução de Moggi [14].

$\mathcal{F}_{st}[[\lambda x.e]] = \eta(\lambda x.\mathcal{F}_{st}[[e]])$
$\mathcal{F}_{st}[[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]] =$
  $x \leftarrow \mathcal{F}_{st}[[e_1]]$
  $\texttt{if } x \texttt{ then } \mathcal{F}_{st}[[e_2]] \texttt{ else } \mathcal{F}_{st}[[e_3]]$
$\mathcal{F}_{st}[[\texttt{let } x = e_1 \texttt{ in } e_2]] =$
  $x \leftarrow \mathcal{F}_{st}[[e_1]]$
  $\mathcal{F}_{st}[[e_2]]$
$\mathcal{F}_{st}[[e_1\ e_2]] = \texttt{join } (\mathcal{F}_{st}[[e_1]] \texttt{ <*> } \mathcal{F}_{st}[[e2]])$
$\mathcal{F}_{st}[[v]] = \eta(v)$, onde v é um valor
$\mathcal{F}_{st}[[e \texttt{ where } \{bindings\}]] =$
  $\mathcal{B}[[bind]]\ \mathcal{F}_{st}[[e]]$, onde $bind \in bindings$
$\mathcal{B}[[x\ id1\ ids = e]] = \texttt{let } x = \lambda id1.\mathcal{L}[[ids, \mathcal{F}_{st}[[e]]]]$
  , onde $\mathcal{L}$ irá encadear as entradas em abstrações
$\mathcal{F}_t[[t]] = t$, para t sendo uma variável ou tipo básico
$\mathcal{F}_t[[t_1 \rightarrow \epsilon\ t_2]] = t_1 \rightarrow T(t_2)$

Dessa forma, a função $\mathcal{F}$, com auxílio das funções secundárias $\mathcal{B}$ e $\mathcal{L}$, traduz instruções e tipos da linguagem de entrada para a linguagem de saída.

A primeira parte da função de tradução, $\mathcal{F}_{st}$, é encarregada das instruções. Ao receber valores como entrada, $\mathcal{F}_{st}$ realiza apenas a injeção monádica sobre esses valores. Para abstrações lambda, a função é chamada recursivamente sobre o corpo da abstração. No caso de instruções condicionais, um *bind* monádico é criado para desencapsular a condição, que também deve ser traduzida por $\mathcal{F}_{st}$, para, em seguida, gerar a estrutura condicional equivalente na linguagem-alvo.

Quanto à aplicação de funções, tanto a função quanto o argumento são traduzidos. Como a saída da função $\mathcal{F}_{st}$ é sempre um valor encapsulado pela mônada, o tipo da função torna-se $m(\tau_1 \rightarrow m\ \tau_2)$ e o argumento terá tipo $m\ \tau_1$. Para garantir a aplicação correta, os operadores `<*>` e `join` são utilizados. Portanto, o operador `<*>` aplica a função ao argumento monádico, enquanto `join` elimina a mônada duplicada.

Para instruções do tipo `where`, cada cláusula *bind* é traduzida como uma instrução `let` na linguagem alvo. Na linguagem de origem, as cláusulas `where` estão associadas ao conceito de dominância entre blocos, noção que é mantida na tradução por meio do encadeamento de instruções `let`. A função $\mathcal{B}$ é responsável por traduzir essas cláusulas, enquanto $\mathcal{L}$ transforma cada argumento em abstrações encapsuladas pela mônada, com exceção do primeiro argumento, garantindo o tipo da instrução como $\tau_1 \rightarrow m\ \tau_2$.

A segunda parte da função de tradução, $\mathcal{F}_t$, é responsável por traduzir os tipos e efeitos das funções. Como a tradução ocorre a partir da sintaxe, os tipos são convertidos diretamente, sem inferência adicional. Tipos básicos e polimórficos não sofrem alterações, apenas os tipos funções são modificados. Na linguagem de origem, as funções possuem tipo $\tau_1 \rightarrow \epsilon\ \tau_2$, onde ϵ representa zero ou mais efeitos.

Para transformar os efeitos em ϵ em mônadas, cada efeito não polimórfico é representado por um GADT (*generalized algebraic data type*). Por exemplo, um efeito para lançar uma exceção (*exception*), juntamente com sua operação (*throw*), será traduzido para a seguinte estrutura:

```
data Exception t where
    Throw :: String -> Exception ()
```

Com a estrutura de dados definida, *Exception* pode ser elevada a condição de mônada utilizando o construtor *Eff*, fornecido pela biblioteca *extensible-effects*. Para isso, a função *send*, cujo tipo inferido é $Member\ t\ r \Rightarrow t\ v \rightarrow Eff\ r\ v$, é usada para elevar o construtor *Throw*.

```
throw :: String -> Eff r ()
throw s = send (Throw s)
```

A função *throw*, gerada com *send*, pode então ser utilizada durante a computação. Qualquer outro efeito passa pelo mesmo procedimento.

Por fim, a declaração de uma função *foo* cujo tipo inferido na linguagem de origem é:

$$int \rightarrow \langle Amb, State, \mu \rangle\ bool$$

é traduzida para:

$$(Member\ Amb\ r,\ Member\ State\ r) \Rightarrow Int \rightarrow Eff\ r\ Bool$$

Neste caso, é a classe *Member* a responsável por modelar a união dos efeitos *Amb* e *State* em *r*.

Funções apenas com efeitos polimórficos são traduzidas de forma equivalente, onde uma função de tipo $a \rightarrow \mu\ b$ torna-se $a \rightarrow Eff\ r\ b$. Como *r* representa a lista de efeitos, executar essa função com uma lista vazia de efeitos vai resultar em um valor puro. Para efeitos presentes em diferentes partes do código, diferentes variáveis são geradas para cada segmento. Por exemplo, uma função com tipo $int \rightarrow \langle State, \mu \rangle\ int \rightarrow \langle Amb, \mu \rangle\ int$ será traduzida como $(Member\ State\ r,\ Member\ Amb\ v) \Rightarrow Int \rightarrow Eff\ r\ Int \rightarrow Eff\ v\ Int$.

Com a função de tradução devidamente estabelecida, foi possível gerar testes utilizando as saídas apresentadas no trabalho de Rigon et al. [17].

4 Resultados

Por questão de espaço, apresentamos alguns testes simples, focando em algumas funcionalidades específicas. Outros exemplos e a implementação estão disponíveis para acesso online pela plataforma GitHub[4]. Para verificar a propriedade de currificação, o seguinte código foi proposto como entrada para o algoritmo de tradução.

```
testCurry1 : ∀a b c d.int → d int → c int → b int → a
      int
testCurry1 a0 b0 c0 d0 = bl2 () where {
   bl2 _ =
    if (a0 < c0)
       then (bl3 ())
            else (bl4 ()));
   where {
    bl3 _ =
     (a0 + b0)
    bl4 _ =
     (c0 + d0)
   }
}
```

[4]Em https://github.com/KarlaJoriatti/ANF-to-MonadLC.

```
testCurry2 : ∀a b c d.int → d int → c int → b int → a
      int
testCurry2 a0 b0 = bl2 () where {
   bl2 _ =
    ((testCurry1 a0) b0)
  }
```

Neste primeiro teste, `testCurry1` é uma função pura que recebe quatro valores inteiros, realiza algumas operações booleanas e aritméticas e retorna um valor inteiro. A função `testCurry2`, por outro lado, foi implementada com dois parâmetros explícitos que são utilizados na aplicação parcial de `testCurry1` que é retornada. Ambas as funções foram traduzidas para o trecho de código abaixo. Com o intuito de simplificar o código, os operadores `<*>` e `join` foram embutidos na função `eapp`.

```
testCurry1 :: Int -> Eff t0 (Int -> Eff t1 (Int -> Eff t2
     (Int -> Eff t3 Int )))
testCurry1 a0 =
 return $ \b0 ->
 return $ \c0 ->
 return $ \d0 -> do
    let bl2 _ = do
        let bl3 _ = do
                return (a0 + b0)
            bl4 _ = do
                return (c0 + d0)
        t4 <- return (a0 < c0)
        if t4
        then ( return bl3 `eapp` return ())
        else ( return bl4 `eapp` return ())
    ( return bl2 `eapp` return ())

testCurry2 :: Int -> Eff t0 (Int -> Eff t1 (Int -> Eff t2
     (Int -> Eff t3 Int )))
testCurry2 a0 =
 return $ \b0 -> do
    let bl2 _ = do
        (( return testCurry1 `eapp` return a0) `eapp`
            return b0)
    ( return bl2 `eapp` return ())
```

Assim como nas funções de entrada, que apresentam variáveis de efeito (*a, b, c* e *d*), a tradução da declaração de tipo de ambas as funções inclui *sequence points* (termo vindo da literatura de C). Dessa forma, é possível manter as funções consistentemente com tipo $\tau_1 \rightarrow \tau_2$, permitindo currificação. Para realizar a execução, como ambas as funções são puras, a função *run* da biblioteca é utilizada. Este interpretador resolve a mônada *Eff* para uma lista de efeitos vazia.

```
> run $ (run $ (run $ (run $ testCurry2 10) 6) 3) 8
11
```

A fim de testar a tradução de funções recursivas, a função que calcula o fatorial de um número foi dada como entrada para o algoritmo de tradução. O código na linguagem de origem está presente abaixo.

```
fatorial : ∀a.int → a int
fatorial n0 = bl2 () where {
```

```
    bl2 _ =
     (n0 * ( fatorial (n0 - 1)))
  }
```

Para tratar a utilização de um operador juntamente com valores puros e com efeito, os operadores <$> e <*> foram utilizados, tendo como saída o seguinte código.

```
fatorial :: Int -> Eff t0 Int
fatorial n0 = do
 let bl2 _ = do
     ( ( * ) <$> return n0 <*> ( return fatorial `eapp`
           return (n0 - 1)))
 ( return bl2 `eapp` return ())
```

Por fim, o último teste inclui a tradução de uma função com múltiplos efeitos. A função `foo` apresenta dois efeitos, ambiguidade e estado, modelados na linguagem de origem como se segue:

```
effect Amb {
 flip : ∀a.unit → <Amb, a> bool
}
effect State {
 get : ∀a.unit → <State, a> int
 set : ∀a.int → <State, a> unit
}
```

Assim como mencionado na Seção 3.3, a tradução dos efeitos ocorre por meio de GADTs e da função *send* da biblioteca que vai elevar os efeitos para a mônada *Eff*. Portanto, a estrutura de dados equivalente aos efeitos *Amb* e *State* na linguagem de destino é dada a seguir:

```
data Amb x where
   Flip :: Amb Bool

flip () = send Flip

data State x where
   Get :: State Int
   Set :: Int -> State ()

get () = send Get
set x0 = send (Set x0)
```

A seguir, é possível realizar normalmente a tradução da função `foo`.

```
foo : ∀a b.a → <Amb, State, b> int
foo _ = bl2 () where {
   bl2 _ =
    let p1 = (flip ()) in bl3 ()
      where {
        bl3 _ =
            (if p1
                then (bl4 ())
                    else (bl5 ()))
            where {
              bl4 _ =
               let _ = (set 1) in (bl6 ())
              bl5 _ =
               let _ = (set 2) in (bl6 ())
              bl6 _ =
               let y1 = (get ()) in (bl7 ())
                 where {
                    bl7 _ = y1
                 } } } }
```

Neste caso, para inserir os efeitos *Amb* e *State*, na lista de efeitos correspondente, aqui indicada pela variável $t0$, a restrição para a classe *Member* é adicionada.

```
foo :: (Member Amb t0,Member State t0) => a -> Eff t0 Int
foo _ = do
 let bl2 _ = do
     p1 <- ( return flip `eapp` return ())
     let bl3 _ = do
         let bl4 _ = do
                _ <- ( return set `eapp` return 1)
                ( return bl6 `eapp` return ())
             bl5 _ = do
                _ <- ( return set `eapp` return 2)
                ( return bl6 `eapp` return ())
             bl6 _ = do
                y1 <- ( return get `eapp` return ())
                let bl7 _ = do
                       return y1
                ( return bl7 `eapp` return ())
         t17 <- return p1
         if t17
         then ( return bl4 `eapp` return ())
         else ( return bl5 `eapp` return ())
     ( return bl3 `eapp` return ())
 ( return bl2 `eapp` return ())
```

Além das traduções, todas as funções puras traduzidas (*fatorial* e *testCurry2*) podem ser executadas utilizando a função *run* da biblioteca, que resolve a mônada *Eff* para uma lista vazia de efeitos. Já para as funções com efeitos, criando manualmente *handlers* para cada efeito, é possível chegar a diferentes resultados dependendo da ordem em que os efeitos são resolvidos, assim como mencionado na Seção 2.3.2.

```
> run $ runAmb$ runState 0 $ foo 1
[(1, 1), (2, 2)]
> run $ runState 0 $ runAmb $ foo 1
(2, [1, 2])
```

Em síntese, os testes propostos foram capazes de validar certas funcionalidades essenciais, além de apresentar traduções satisfatórias capazes de serem compiladas e executadas através do compilador de *Haskell* sem gerar erros de tipo ou semânticos.

5 Conclusão

O presente trabalho demonstra com êxito a tradução dirigida pela sintaxe de uma linguagem intermediária em ANF, utilizando os conceitos de mônadas livres para lidar com a combinação de efeitos. Essa proposta representa uma solução mais geral do que a abordada por Vazou and Leijen [20], na qual são geradas novas mônadas para combinar efeitos.

Além disso, a partir dos testes, foi possível observar que a tradução final condiz com a semântica esperada para o código original,

validando que a representação intermediária em ANF com efeitos é executável no GHC versão 9.2.6, onde os tipos declarados condizem com o código traduzido. Considerando que o código em ANF é gerado a partir de código SSA, o trabalho também fornece uma forma indireta de especificar uma semântica para SSA com efeitos algébricos.

Apesar dos resultados, o trabalho limita-se à tradução das funções e efeitos, não podendo gerar os códigos equivalentes aos *handlers* responsáveis por tratar os efeitos, o que seria correspondente à geração da função *run*. Portanto, um importante trabalho futuro é investigar a possibilidade da geração da função *run*, ou, ao menos, dadas funções *run* que correspondam ao tratamento de efeitos individuais, a geração automática da combinação desses efeitos.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, bem como da Universidade Federal de Ouro Preto. Além disso, o grupo de Fundamentos da Computação (FUNÇÃO) da Universidade do Estado de Santa Catarina é parcialmente financiado pela FAPESC.

REFERÊNCIAS

[1] Andrew W. Appel. 1998. SSA is functional programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20. doi:10.1145/278283.278285

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming* 84, 1 (2015), 108–123.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320

[4] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. *ACM Sigplan Notices* 28, 6 (1993), 237–247.

[5] David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. doi:10.1145/319838.319848

[6] Richard A. Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95)*. Association for Computing Machinery, New York, NY, USA, 13–22. doi:10.1145/202529.202532

[7] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.

[8] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices* 48, 12 (2013), 59–70.

[9] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. doi:10.4204/eptcs.153.8

[10] Daan Leijen. 2016. *Algebraic effects for functional programming*. Technical Report. Technical Report. MSR-TR-2016-29. Microsoft Research technical report.

[11] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. doi:10.1145/199448.199528

[12] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. doi:10.1145/73560.73564

[13] Christoph Lüth and Neil Ghani. 2002. Composing monads using coproducts. *ACM SIGPLAN Notices* 37, 9 (2002), 133–144.

[14] E. Moggi. 1989. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, Pacific Grove, California, USA, 14–23.

[15] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[16] Gordon D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science* 1, 2 (1975), 125–159.

[17] Leonardo Filipe Rigon, Paulo Torrens, and Cristiano Vasconcellos. 2020. Inferring types and effects via static single assignment. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1314–1321. doi:10.1145/3341105.3373888

[18] Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style.. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. Association for Computing Machinery, New York, NY, USA, 288–298. doi:10.1145/141471.141563

[19] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

[20] Niki Vazou and Daan Leijen. 2016. From Monads to Effects and Back. In *Practical Aspects of Declarative Languages*, Marco Gavanelli and John Reppy (Eds.). Springer International Publishing, Cham, 169–186.

[21] Philip Wadler. 1995. Monads for functional programming. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–52.