Tradução da Forma SSA Gerada pela LLVM para Código Funcional

Igor Schiessl Froehner
Universidade do Estado de Santa Catarina
igor.sf14@edu.udesc.br

Orientador: Dr. Cristiano Damiani Vasconcellos

Coorientador: Dr. Paulo Henrique Torrens

28/06/2024

Sumário

Partindo dos fatos que:

- Compiladores usam de diversas representações intermediárias (IRs) durante o processo de compilação de código:
  - Árvore Sintática Abstrata;
  - Grafo de Fluxo de Controle;
  - SSA (Static Single-Assignment);
- LLVM (Low Level Virtual Machine) é um *framework* de compilação de código que é o estado da arte atualmente e fundamenta sua RI em SSA;
- Foi demonstrado que SSA é correspondente ao paradigma de programação funcional e ANF (APPEL, 1998) (CHAKRAVARTY; KELLER; ZADARNOWSKI, 2004);

O que seria esta tradução?

1. Começando de código arbitrário em uma linguagem fonte:

```c
int safe_div(int n, int d) {
    if (d == 0) return -1;
    else return n / d;
}
```

Figura: Exemplo de Código de Divisão Segura em C

Fonte: O autor

O que seria esta tradução?

2. Então será gerada a representação intermediária da LLVM, que é em forma SSA:

```
define i32 @safe_div(i32 %0, i32 %1) {
2:
  %3 = icmp eq i32 %1, 0
  br i1 %3, label %6, label %4
4:
  %5 = sdiv i32 %0, %1
  br label %6
6:
  %7 = phi i32 [ %5, %4 ], [ -1, %2 ]
  ret i32 %7
}
```

Figura: Divisão Segura em LLVM-IR na forma SSA

Fonte: O autor

O que seria esta tradução?

3. Este código é então traduzido para uma representação funcional em ANF.

```haskell
safe_div a0 a1 =
  let a2 () =
        let a3 = if a1 == 0 then 1 else 0
            a4 () =
              let a5 = a0 `div` a1
              in a6 a5
            a6 a7 =
              let
              in a7
        in if a3 /= 0
          then a6 (-1)
          else a4 ()
  in a2 ()
```

Figura: Divisão Segura em ANF em Haskell. Fonte: O autor

Por quê?

1. LLVM é utilizada por diversas ferramentas para linguagens imperativas:
   - *clang*: C e C++
   - Compilador do Rust
   - Swift, Júlia, etc.
2. Há diversos pontos positivos no paradigma de programação puramente funcional (HUGHES, 1989) (HU; HUGHES; WANG, 2015) (HAMMOND, 2011);
   - Modularização
   - Paralelização
   - Corretude

Objetivo Geral

O objetivo do presente trabalho é investigar a possibilidade de traduzir a representação intermediária SSA gerada pela LLVM para representação funcional em ANF, tendo como objetivo futuro o uso dessa representação em uma extensão do método proposto por Rigon, Torrens e Vasconcellos (2020) para inferir efeitos algébricos em código imperativo real.

Objetivos Específicos

- Estudar os conceitos de compiladores de tradução de código, representação intermediária, SSA e o paradigma de programação funcional;
- Estudar sobre a tradução de código intermediário (SSA) da LLVM para código puramente funcional;
- Implementar um tradutor de código SSA gerado pela LLVM para código puramente funcional;
- Elucidar a possibilidade de fazer tal tradução e quais são as ressalvas quanto a essa abordagem.

Revisão - Forma de Atribuição Única Estática (SSA)

Segundo Muchnick (1997), um procedimento está em Forma de Atribuição Única Estática (SSA - *Static Single-Assignment*), se cada variável que recebe um valor neste é alvo de atribuição somente uma vez.

$$\begin{array}{ccccc} p & \leftarrow & a & + & b \\ q & \leftarrow & p & - & c \\ p & \leftarrow & p & * & q \\ p & \leftarrow & d & - & p \\ q & \leftarrow & p & + & q \end{array}$$

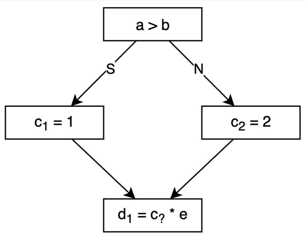
(a) Não SSA

$$\begin{array}{ccccc} p_1 & \leftarrow & a & + & b \\ q_1 & \leftarrow & p_1 & - & c \\ p_2 & \leftarrow & p_1 & * & q_1 \\ p_3 & \leftarrow & d & - & p_2 \\ q_2 & \leftarrow & p_3 & + & q_1 \end{array}$$

(b) Em forma SSA

Figura: Comparação entre atribuições em não SSA e em SSA

Fonte: O autor, adaptado de Lam et al. (2006)



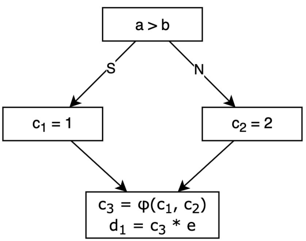
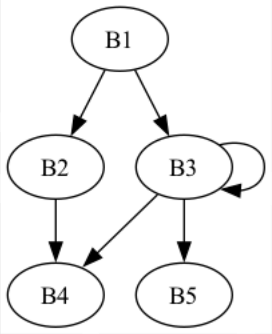


Figura: Exemplo de uso da função φ
Fonte: O Autor

SSA define a notação conceitual da função φ para combinar definições divergentes de uma mesma variável no fluxo de controle (LAM et al., 2006).

Em grafos de fluxo, um nó d é dito **dominar** um nó n se a partir do nó inicial todos os caminhos até n devem passar pelo nó d. E domina estritamente se $d \neq n$.

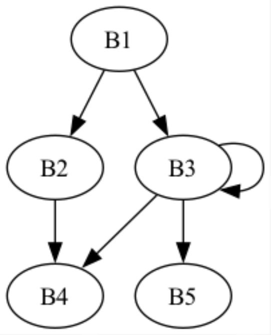


	Dominados				
B1	B1	B2	B3	B4	B5
B2	B2				
B3	B3	B5			
B4	B4				
B5	B5				

Figura: Exemplo de Dominância

Fonte: O autor

O **dominador imediato** de um nó *n* é o único nó que, domina estritamente *n*, mas não domina nenhum outro nó que domina estritamente *n*.

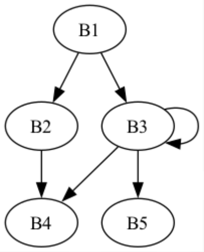


	Dominados Imediatos
B1	B2 B3 B4
B2	
B3	B5
B4	
B5	

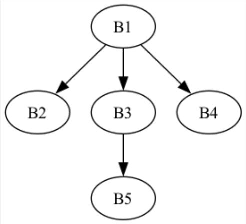
Figura: Exemplo de Dominância

Fonte: O autor

A **árvore de dominância** de um dado grafo de fluxo é a árvore em que os filhos de cada nó n_i são os nós dominados imediatamente por n_i. A raiz da árvore é o nó inicial.



(a) Grafo de fluxo de controle



(b) Árvore de dominância

Figura: Comparação entre CFG e Árvore de Dominância

Fonte: O autor

Revisão - ANF [1/2]

Em ANF (*Adminstrative Normal Form*) os argumentos em aplicações de expressões devem ser termos atômicos.

```
h (f (g x + 1)) (k (y * 2))
```

(a) Não ANF

```
let a = g x in
let b = a + 1 in
let c = y * 2 in
let d = k c in
let e = f b in
h e d
```

(b) Em ANF

Figura: Comparação de uma expressão em ANF

Fonte: O autor

$$
\begin{array}{rcl}
e & ::= & \textbf{let}\ \ x \ = \ v\ \overline{v}\ \ \textbf{in}\ \ e \\
 & | & v\ \overline{v} \\
v & ::= & x \ | \ c \ | \ \lambda x.e \\
x & \in & \textit{variáveis} \\
c & \in & \textit{constantes}
\end{array}
$$

Figura: Gramática Exemplo de ANF

Fonte: O autor

- Appel (1998) demonstrou a correspondência entre SSA e o paradigma funcional.
- Chakravarty, Keller e Zadarnowski (2004) demonstraram de maneira mais formal a correspondência entre SSA e ANF. E também demonstraram que uma otimização que é tradicionalmente feita utilizando SSA pode ser feita por meio de ANF.

"[...] um *framework* de compilação projetado para suportar análise e transformação de programas arbitrários de forma transparente e duradoura, provendo informação de alto nível para as transformações do compilador em *compile-time*, *link-time*, *run-time* e em *idle time* (entre execuções)." (LATTNER; ADVE, 2004)

Conseguindo isso através de 2 principais pontos:

- A LLVM-IR (fundamentada no modelo SSA);
- O *design* do compilador.

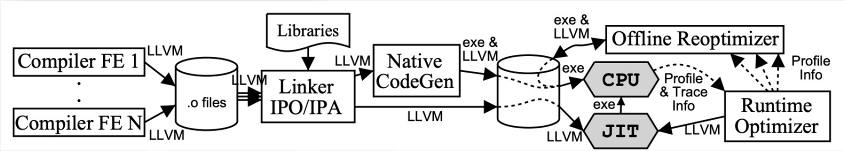


Figura: Projeto da LLVM
Fonte: Lattner e Adve (2004)

Revisão - LLVM-IR [1/2]

LLVM-IR é a linguagem que a LLVM utiliza como representação intermediária durante todo o processo de compilação.

Além de ser fundamentada na forma SSA, Lattner e Adve (2004) colocam três principais aspectos:

- Sistema de Tipos de Baixo Nível;
- Instruções de Baixo Nível para Conversão de Tipos e Aritmética de Endereços;
- Instruções de Baixo Nível para tratamento de Exceções;

Revisão - LLVM-IR [2/2]

```
; ModuleID = 'myfib.c'
source_filename = "myfib.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx14.0.0"

; Function Attrs: nofree norecurse nosync nounwind readnone ssp ↩
    uwtable(sync)
define i32 @fib(i32 noundef %0) local_unnamed_addr #0 {
  %2 = icmp sgt i32 %0, 99
  br i1 %2, label %12, label %3

3:                                                ; preds = %1
  %4 = icmp sgt i32 %0, 0
  br i1 %4, label %5, label %12

5:                                                ; preds = %3, %5
  %6 = phi i32 [ %8, %5 ], [ 0, %3 ]
  %7 = phi i32 [ %10, %5 ], [ 0, %3 ]
  %8 = phi i32 [ %9, %5 ], [ 1, %3 ]
  %9 = add nsw i32 %6, %8
  %10 = add nuw nsw i32 %7, 1
  %11 = icmp eq i32 %10, %0
  br i1 %11, label %12, label %5, !llvm.loop !6

12:                                               ; preds = %5, %3, %1
  %13 = phi i32 [ -1, %1 ], [ 0, %3 ], [ %8, %5 ]
  ret i32 %13
}

attributes #0 = { nofree norecurse nosync nounwind readnone } ; ...

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 2, !"SDK Version", [2 x i32] [i32 14, i32 4]}
!1 = !{i32 1, !"wchar_size", i32 4}
; ... continua com mais metadados
```

Figura: Exemplo de LLVM-IR. Fonte: O autor, gerada com *clang*.

- Uma adaptação do método de Chakravarty, Keller e Zadarnowski (2004) para traduzir código de LLVM-IR na forma SSA para programação funcional em ANF.

Etapas:

1. *Parser* de um subconjunto da LLVM-IR;
2. Definição do ANF de saída;
3. Implementação do método de tradução.

Criado um *parser* para um subconjunto da LLVM-IR em que:

- São compreendidos inteiros simples;
- Não são tratadas operações que geram efeitos colaterais;
- Todos os registradores e blocos devem ter nome;
- Durante a análise léxica são ignoradas palavras-chave e construções que fogem ao escopo desse trabalho: metadados, informação para debug, etc.

$$
\begin{array}{rcl}
p & ::= & \textbf{define}\ t\ x\ (\ \overline{t\ \ x}\ )\ \{\ \overline{b}\ \} \\
b & ::= & x\ \ \textbf{:}\ \ \overline{\varphi}\ \ \overline{s}\ \ f \\
\varphi & ::= & x\ \ =\ \ \textbf{phi}\ \ t\ \ \overline{a} \\
s & ::= & x\ \ \textbf{=}\ \ o\ \mid\ x\ \ \textbf{=}\ \ q \\
q & ::= & \textbf{call}\ \ t\ \ x\ \ (\ \overline{t\ v}\ ) \\
f & ::= & \textbf{br}\ \ t\ \ x\ \mid\ \textbf{br}\ \ t\ \ v\ \ \textbf{,}\ \ t\ \ x\ \ \textbf{,}\ \ t\ \ x \mid\ \textbf{ret}\ \ t\ \ v \\
a & ::= & \textbf{[}\ \ v\ \ \textbf{,}\ \ x\ \ \textbf{]} \\
o & ::= & \beta\ \ t\ \ v\ \ \textbf{,}\ \ v\ \mid \\
 & & \textbf{icmp}\ \ \tau\ \ t\ \ v\ \ \textbf{,}\ \ v\ \mid \\
 & & \textbf{select}\ \ t\ \ v\ \ \textbf{,}\ \ t\ \ v\ \ \textbf{,}\ \ t\ \ v\ \mid \\
 & & \mu\ \ t\ \ v\ \ \textbf{to}\ \ t \\
v & ::= & x \mid c \\
t & \in & \textit{tipos nativos da LLVM} \\
x & \in & \textit{variáveis locais ou globais} \\
c & \in & \textit{constantes} \\
\beta & \in & \textit{operações binárias da LLVM.} \\
\tau & \in & \textit{opções de comparação da LLVM.} \\
\mu & \in & \textit{operações de conversão da LLVM.}
\end{array}
$$

Figura: Gramática de interpretação da LLVM-IR. Fonte: O autor.

Define-se também a gramática de saída do método de tradução, que é uma extensão de uma gramática que garante a forma ANF.

$$
\begin{array}{rcl}
f & ::= & \mathbf{def}\ \ x\ \ \mathbf{=}\ \ \boldsymbol{\lambda}\,\overline{x}\ .\ \mathbf{let}\ \ l\ \ \mathbf{in}\ \ x\ \ \overline{v} \\
l & ::= & x\ \ \mathbf{=}\ \ \boldsymbol{\lambda}\,\overline{x}\ .\ \mathbf{let}\ \ \overline{d}\ \ \overline{l}\ \ \mathbf{in}\ \ j \\
d & ::= & x\ \ \mathbf{=}\ \ e \\
e & ::= & v\ \mid\ x\ \ \overline{v}\ \mid\ o\ \ \overline{v} \\
j & ::= & v\ \ \overline{v}\ \mid\ \mathbf{if}\ \ v\ \ \neq\ 0\ \ \mathrm{then}\ \ v\ \ \overline{v}\ \ \mathbf{else}\ \ v\ \ \overline{v} \\
v & ::= & x \mid c \\
o & \in & \textit{operações nativas da LLVM traduzidas} \\
x & \in & \textit{variáveis} \\
c & \in & \textit{constantes}
\end{array}
$$

Figura: Gramática do ANF Gerado como Saída

Fonte: O autor

Desenvolvimento - Tradução [1/6]

O método de tradução desenvolvido é uma adaptação do método apresentado por Chakravarty, Keller e Zadarnowski (2004).

Há uma diferença fundamental entre SSA e ANF quanto ao escopo das variáveis:

- Em SSA o escopo é implícito no código e ditado pelo CFG;
- Em ANF o escopo é explícito no código;

Para tratar isso é calculada a árvore de dominância sobre o grafo de fluxo de controle do código em SSA. A árvore serve como guia para a tradução.

Função de Tradução

$$\mathcal{F}(\textbf{define}\ \ t\ \ x\ \ \textbf{(}\ \ \overline{t\ x}\ \ \textbf{)}\ \ \textbf{\{}\ \overline{b}\ \textbf{\}})\ =$$

$$\textbf{def}\ \ x\ \ \boldsymbol{=}\ \ \boldsymbol{\lambda}\ \overline{x}\ \textbf{.let}\ \ \overline{\mathcal{F}_b(b)}\ \ \textbf{in}\ \ \mathcal{F}_i(b)$$

onde $b =$ nó inicial de $\overline{b}$

Retorna o *Label* do Bloco

$$\mathcal{F}_i(x\ \ \textbf{:}\ \ \overline{\varphi}\ \ \overline{s}\ \ f)\ =\ x$$

Tradução de um Bloco Básico

$\mathcal{F}_b(x\ :\ \overline{\varphi}\ \ \overline{s}\ \ f)\ =$

$$x\ \ \mathbf{=}\ \ \boldsymbol{\lambda}\ \overline{\mathcal{F}_\varphi(\varphi)}\ \textbf{.let}\ \ \overline{\mathcal{F}_s(s)}\ \ \overline{\mathcal{F}_b(b')}\ \ \textbf{in}\ \ \mathcal{F}_f(x, f)$$

onde $\overline{b'}$ = nós filhos de x na árvore de dominância

Retorna a Variável Correspondente ao φ

$\mathcal{F}_\varphi(x\ \ \mathbf{=}\ \ \textbf{phi}\ \ t\ \ \overline{a})\ =\ x$

Tradução das Declarações

$$\mathcal{F}_s(x\ =\ \beta\ \ t\ \ v_1\ ,\ v_2)\ =\ x\ =\ v_1\ \text{`}\beta\text{`}\ v_2$$

$$\mathcal{F}_s(x\ =\ \textbf{icmp}\ \ \tau\ \ t\ \ v_1\ ,\ v_2)\ =$$
$$x\ =\ \textbf{if}\ \ v_1\ \tau\ v_2\ \ \textbf{then}\ \ \textbf{1}\ \ \textbf{else}\ \ \textbf{0}$$

$$\mathcal{F}_s(x\ =\ \textbf{select}\ \ t_c\ \ v_c\ ,\ t_1\ \ v_1\ ,\ t_2\ \ v_2)\ =$$
$$x\ =\ \textbf{if}\ \ v_c\ /=\ \textbf{0}\ \ \textbf{then}\ \ v_1\ \ \textbf{else}\ \ v_2$$

$$\mathcal{F}_s(x\ =\ \mu\ \ t_1\ \ v\ \ \textbf{to}\ \ t_2)\ =\ x\ =\ v$$

$$\mathcal{F}_s(x_1\ =\ \textbf{call}\ \ t\ \ x_2\ \ (\ \overline{t\ v}\ ))\ =\ x_1\ =\ x_2\ \ \overline{v}$$

Tradução de um Jump

$\mathcal{F}_f(x_1, \textbf{br}\ \ t\ \ x_2)\ =$

$\quad x_2\ \ \overline{\mathcal{F}_p(x_1, \varphi)}$

$\quad$ onde $x_2\ \ :\ \ \overline{\varphi}\ \ \overline{s}\ \ f =$ bloco com label x_2

$\mathcal{F}_f(x_1, \textbf{br}\ \ t\ \ v\ \ ,\ \ t\ \ x_2\ \ ,\ \ t\ \ x_3)\ =$

$\quad \textbf{if}\ \ v\ \neq\ 0\ \ \textbf{then}\ \ x_2\ \ \overline{\mathcal{F}_p(x_1, \varphi_2)}\ \ \textbf{else}\ \ x_3\ \ \overline{\mathcal{F}_p(x_1, \varphi_3)}$

$\quad$ onde $x_2\ \ :\ \ \overline{\varphi_2}\ \ \overline{s_2}\ \ f_2 =$ bloco com label x_2

$\quad\quad$ e $x_3\ \ :\ \ \overline{\varphi_3}\ \ \overline{s_3}\ \ f_3 =$ bloco com label x_3

$\mathcal{F}_f(x_1, \textbf{ret}\ \ t\ \ v)\ =\ v$

Encontra os Argumentos no φ do Bloco Alvo

$\mathcal{F}_p(x_1, x_2$ **=** **phi** t $\bar{a}) =$

v

onde **[** v **,** x_1 **]** é um elemento de $\bar{a}$

Resultados - Algoritmo de Euclides [1/2]

O Algoritmo de Euclides calcula o máximo divisor comum entre dois inteiros (KNUTH, 2014).

```cpp
int euclides_gcd(int a, int b) {
    if (b == 0) return a;
    else return euclides_gcd(b, a % b);
}
```

Figura: Algoritmo de Euclides em C++

Fonte: O autor

Resultados - Algoritmo de Euclides [2/2]

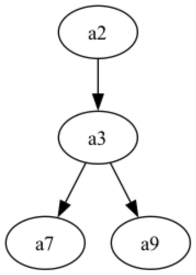
```
define i32 @euclides_gcd(i32 %0, i32 %1) {
2:
  br label %3

3:
  %4 = phi i32 [ %0, %2 ], [ %5, %7 ]
  %5 = phi i32 [ %1, %2 ], [ %8, %7 ]
  %6 = icmp eq i32 %5, 0
  br i1 %6, label %9, label %7

7:
  %8 = srem i32 %4, %5
  br label %3

9:
  ret i32 %4
}
```

(a) CFG Algoritmo de Euclides



(b) Árvore de dominância

Figura: CFG e Árvore de Dominância do Código do Algoritmo de Euclides

Fonte: O autor

Resultados - Exemplo de Tradução [1/9]

$$
\begin{aligned}
&\mathcal{F}(\texttt{define i32 @euclides\_gcd(i32 \%0, i32 \%1) } \{\overline{b}\}) = \\
&\quad \textbf{def}\ \texttt{euclides\_gcd} = \boldsymbol{\lambda}\texttt{a0 a1}.\,\textbf{let} \\
&\qquad\qquad \overline{\mathcal{F}_b(b)} \\
&\qquad\quad \textbf{in}\ \texttt{a2}
\end{aligned}
$$

Resultados - Exemplo de Tradução [2/9]

$$
\begin{array}{l}
\textbf{def}\ \texttt{euclides\_gcd} = \boldsymbol{\lambda}\texttt{a0 a1}.\ \textbf{let} \\
\quad \mathcal{F}_b(\texttt{2: [] [] j}) = \\
\qquad \texttt{a2} = \boldsymbol{\lambda}.\ \textbf{let} \\
\qquad\qquad \overline{\mathcal{F}_s([])} \\
\qquad\qquad \overline{\mathcal{F}_b(b')} \\
\qquad\quad \textbf{in}\ \mathcal{F}_j(\texttt{a2, j}) \\
\quad \textbf{in}\ \texttt{a2}
\end{array}
$$

Resultados - Exemplo de Tradução [3/9]

```
def euclides_gcd = λa0 a1.let
    a2 = λ.let
            𝓕_b(3: φ̄ s̄ j)
        in 𝓕_j(a2, j)
    in a2
```

```
def euclides_gcd = λa0 a1.let
    a2 = λ.let
            F_b(3: φ̄ s̄ j) =
                a3 = λ‾F_φ(φ)‾.let
                        ‾F_s(s)‾
                        ‾F_b(b′)‾
                in F_j(a3, j)
        in a3 a0 a1
    in a2
```

$$
\begin{array}{l}
\textbf{def}\ \texttt{euclides\_gcd} = \boldsymbol{\lambda}\texttt{a0 a1}.\ \textbf{let} \\
\quad \texttt{a2} = \boldsymbol{\lambda}.\ \textbf{let} \\
\qquad\qquad \mathcal{F}_b(3\!: \overline{\varphi}\ \overline{s}\ \texttt{j}) = \\
\qquad\qquad\qquad \texttt{a3} = \boldsymbol{\lambda}\overline{\mathcal{F}_\varphi(\varphi)}.\ \textbf{let} \\
\qquad\qquad\qquad\qquad \overline{\mathcal{F}_s(s)} \\
\qquad\qquad\qquad\qquad \overline{\mathcal{F}_b(b')} \\
\qquad\qquad\qquad \textbf{in}\ \mathcal{F}_j(\texttt{a3},\ \texttt{j}) \\
\qquad \textbf{in}\ \texttt{a3 a0 a1} \\
\quad \textbf{in}\ \texttt{a2}
\end{array}
$$

$$
\begin{array}{l}
\textbf{def}\ \text{euclides\_gcd} = \boldsymbol{\lambda}\text{a0 a1}.\,\textbf{let} \\
\quad \text{a2} = \boldsymbol{\lambda}.\,\textbf{let} \\
\qquad\qquad \mathcal{F}_b(3\colon\ \overline{\varphi}\ \overline{s}\ \ \text{j}) = \\
\qquad\qquad\quad \text{a3} = \boldsymbol{\lambda} \\
\qquad\qquad\quad \mathcal{F}_\varphi(\%4 = \text{phi}\ \ \ldots) \\
\qquad\qquad\quad \mathcal{F}_\varphi(\%5 = \text{phi}\ \ \ldots).\,\textbf{let} \\
\qquad\qquad\qquad \mathcal{F}_s(\%6 = \text{icmp eq i32 }\%5,\ 0) \\
\qquad\qquad\qquad \mathcal{F}_b(7\colon\ \varphi_{a7}\ \overline{s_{a7}}\ j_{a7}) \\
\qquad\qquad\qquad \mathcal{F}_b(9\colon\ \varphi_{a9}\ \overline{s_{a9}}\ j_{a9}) \\
\qquad\qquad\quad \textbf{in}\ \mathcal{F}_j(\text{a3},\ \text{j}) \\
\qquad\ \ \textbf{in}\ \text{a3 a0 a1} \\
\quad \textbf{in}\ \text{a2}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{def}\ \ \text{euclides\_gcd}\ =\ \boldsymbol{\lambda}\text{a0 a1}.\ \mathbf{let} \\
\quad \text{a2} = \boldsymbol{\lambda}.\ \mathbf{let} \\
\qquad\qquad \text{a3} = \boldsymbol{\lambda}\text{a4 a5}.\ \mathbf{let} \\
\qquad\qquad\qquad \text{a6} = \mathbf{if}\ \text{a5} == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\qquad\qquad\qquad \mathcal{F}_b(7\colon\ [\,]\ \ \overline{s_{a7}}\ \ j_{a7}) = \\
\qquad\qquad\qquad\qquad \text{a7} = \boldsymbol{\lambda}\overline{\mathcal{F}_\varphi([])}).\ \mathbf{let} \\
\qquad\qquad\qquad\qquad\qquad\quad \overline{\mathcal{F}_s(s_{a7})} \\
\qquad\qquad\qquad\qquad\qquad\quad \overline{\mathcal{F}_b([])} \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathcal{F}_j(a7, j_{a7}) \\
\qquad\qquad\qquad \mathcal{F}_b(9\colon\ [\,]\ \ [\,]\ \ j_{a9}) = \\
\qquad\qquad\qquad\qquad \text{a9} = \boldsymbol{\lambda}\overline{\mathcal{F}_\varphi([])}).\ \mathbf{let} \\
\qquad\qquad\qquad\qquad\qquad\quad \overline{\mathcal{F}_s([]])} \\
\qquad\qquad\qquad\qquad\qquad\quad \overline{\mathcal{F}_b([])} \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathcal{F}_j(a9, j_{a9}) \\
\qquad\qquad \mathbf{in}\ \ \mathbf{if}\ \text{a6} =/\ 0\ \mathbf{then}\ \text{a9}\ \mathbf{else}\ \text{a7} \\
\qquad\quad \mathbf{in}\ \text{a3 a0 a1} \\
\quad \mathbf{in}\ \text{a2}
\end{array}
$$

```
def euclides_gcd = λa0 a1. let
    a2 = λ. let
            a3 = λa4 a5. let
                  a6 = if a5 == 0 then 1 else 0
                  a7 = λ. let
                             Fs(%8 = srem i32 %4, %5)
                        in Fj(a7, ja7)
                  a9 = λ. let
                        in Fj(a9, ja9)
            in if a6 =/ 0 then a9 else a7
        in a3 a0 a1
    in a2
```

```
def euclides_gcd = λa0 a1. let
    a2 = λ. let
            a3 = λa4 a5. let
                a6 = if a5 == 0 then 1 else 0
                a7 = λ. let
                        a8 = a5 `mod` a5
                    in a3 a5 a8
                a9 = λ. let
                    in a4
            in if a6 =/ 0 then a9 else a7
        in a3 a0 a1
    in a2
```

```haskell
import Data.Bits

euclides_gcd a0 a1 =
  let
    a2 () =
      let
        a3 a4 a5 =
          let
            a6 = if a5 == 0 then 1 else 0
            a7 () =
              let
                a8 = a4 `mod` a5
              in a3 a5 a8
            a9 () =
              let
              in a4
          in if a6 /= 0
            then a9 ()
            else a7 ()
      in a3 a0 a1
  in a2 ()
```

Figura: Saída do Tradutor para o Algoritmo de Euclides. Fonte: O autor.

Resultados - Discussão [1/2]

Em termos: **o método implementado nesse trabalho é capaz de traduzir funções puras que usam somente tipos inteiros simples de LLVM-IR para código Haskell em ANF**.

Resultados - Discussão [2/2]

- A extensão para o uso de mais tipos como os de ponto flutuante pode ser trivial:
  - Envolve o *parsing* os novos tipos e constantes;
  - Adicionar possíveis instruções específicas para estes;

- Ao não permitir efeitos colaterais a tradução desenvolvida não compreende variáveis globais, aritmética de ponteiros, entrada e saída, ou as demais chamadas de sistema. Esse tratamento poderia ser feito por meio de duas maneiras:
  - O uso de efeitos algébricos como o trabalho de Rigon, Torrens e Vasconcellos (2020);
  - Implementar uma tradução sintática para a metalinguagem monádica de Moggi (1988);

- Foi possível criar um tradutor de funções puras da LLVM-IR para o paradigma funcional em ANF;
- O tradutor desenvolvido gera código executável em Haskell;
- Há ressalvas quanto a tradução proposta relativos ao tratamento de efeitos colaterais;
- Trabalhos futuros podem explorar a extensão do método proposto.

APPEL, A. W. Ssa is functional programming. *Acm Sigplan Notices*, ACM New York, NY, USA, v. 33, n. 4, p. 17–20, 1998.

CHAKRAVARTY, M. M.; KELLER, G.; ZADARNOWSKI, P. A functional perspective on ssa optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 82, n. 2, p. 347–361, 2004.

HAMMOND, K. Why parallel functional programming matters: Panel statement. In: SPRINGER. *International Conference on Reliable Software Technologies*. [S.l.], 2011. p. 201–205.

HU, Z.; HUGHES, J.; WANG, M. How functional programming mattered. *National Science Review*, Oxford University Press, v. 2, n. 3, p. 349–370, 2015.

HUGHES, J. Why functional programming matters. *The computer journal*, Oxford University Press, v. 32, n. 2, p. 98–107, 1989.

KNUTH, D. E. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. [S.l.]: Addison-Wesley Professional, 2014.

LAM, M. et al. Compilers: principles, techniques, and tools. *Pearson Education*, 2006.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: IEEE. *International symposium on code generation and optimization, 2004. CGO 2004.* [S.l.], 2004. p. 75–86.

MOGGI, E. *Computational lambda-calculus and monads*. [S.l.]: University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1988.

Referências

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

RIGON, L. F.; TORRENS, P.; VASCONCELLOS, C. Inferring types and effects via static single assignment. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2020. p. 1314–1321.