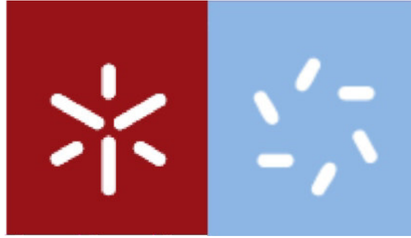

Universidade do Minho
Dep. de Matemática e Aplicações

Tiago Alexandre da Costa Ferreira

UM ESTUDO SOBRE A CORRESPONDÊNCIA ENTRE PROGRAMAÇÃO FUNCIONAL COM CONTINUAÇÕES E PROGRAMAÇÃO IMPERATIVA SINGLE ASSIGNMENT

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho realizado sob a orientação do
Professor Doutor Luís Filipe Ribeiro Pinto
Professora Doutora Maria João Gomes Frade

dezembro 2014

Nome
Tiago Alexandre da Costa Ferreira

Endereço eletrónico dev.tiago@gmail.com

Número do Cartão de Cidadão 12490198

Orientadores
Professor Doutor Luís Filipe Ribeiro Pinto
Professora Doutora Maria João Gomes Frade

Ano de Conclusão 2014

Designação do Mestrado Mestrado em Matemática e Computação

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE

Universidade do Minho, dezembro 2014

Assinatura:

RESUMO

No universo das linguagens de programação existe uma diversidade grande de paradigmas, frequentemente motivados pela necessidade da resolução computacional de novos problemas, em novos contextos. Dois destes paradigmas, que atraem a atenção de programadores e cientistas da computação desde os primórdios das linguagens de programação, são o paradigma imperativo e o paradigma funcional.

Estes dois paradigmas assentam em ideias bastante diferentes. O paradigma imperativo baseia-se numa noção de estado e em programas que são sequências de comandos cuja execução vai provocando alterações ao estado. Por sua vez, o paradigma funcional baseia-se na noção matemática de função e a execução de programas (uma coleção de funções) corresponde ao cálculo do valor de uma expressão, envolvendo as funções definidas no programa.

No entanto, existem fragmentos do paradigma imperativo e do paradigma funcional que podem ser relacionados. De facto, o formato *Single-Assignment* (*SA*), do lado imperativo, e o formato *Continuation-Passing Style* (*CPS*), do lado funcional, que são usados como linguagens "intermédias" no processo de compilação de linguagens de programação mais abstratas, por facilitarem e agilizarem processos de optimização na geração de código máquina eficiente correspondem, na essência, a uma mesma linguagem.

Neste trabalho estuda-se de uma forma detalhada a relação entre programação imperativa no formato *SA* e programação funcional no formato *CPS*.

Recorrendo a uma linguagem imperativa simples para representar o formato *SA* e a um subconjunto do λ-calculus para o formato *CPS*, construímos, entre eles, uma bijeção sintática (ao nível dos programas) e semântica (ao nível da execução de programas). Um resultado destas bijeções é que o formato *SA* pode ser pensado como uma certa escolha de representantes de classe para a noção de α-equivalência para programas imperativos correspondente a essa mesma noção para λ-termos

Durante o trabalho, foi ainda desenvolvida uma pequena ferramenta computacional, implementada na linguagem Haskell, que permitiu testar e animar os diversos conceitos envolvidos no estudo.

ABSTRACT

In the universe of programming languages there is a wide range of paradigms that are often motivated by the need to solve new computational problems, in new contexts. Two of these paradigms, which have drawn programmers and computer scientists' attention since the beginnig of programming languages, are the imperative programming paradigm and the functional programming paradigm.

Both paradigms are built upon different ideas. The imperative programming paradigm is based on the notion of state, its programs are built by sequencing multiple commands, and execution generates state changes. The functional programming paradigm is based on the notion of mathematical function (a program is a collection of functions) and program execution corresponds to the evaluation of expressions (involving the functions defined in the program).

Nevertheless, there are fragments within the imperative programming paradigm and the functional programming paradigm that can be related. The *Single Assignment* (*SA*) form, on the imperative programming paradigm, and the *Continuation-Passing Style* (*CPS*) form in the functional programming paradigm, are used as "intermediate" languages in the compilation process of more abstract languages, due to the fact that they simplify and make more agile the processes of optimization when it comes to the genaration of efficient machine code, they are in its essence the same language.

In this work we study in detail the relationship between imperative programming in the *SA* form and functional programming in the *CPS* form.

Using a simple imperative language to represent the *SA* form and a subset of λ-calculus in *CPS* form, we built between them a syntatic bijection (at code level) and a semantic bijection (at program execution level). We concluded that the *SA* form can be tought as a certain choice of class representatives in the notion of α-equivalence in imperative programs that matches that same notion in λ-terms.

During this work, we developed a small computational tool using the Haskell language which allowed us to simulate multiple concepts that are involved in the study.

CONTEÚDO

1 INTRODUÇÃO 1
- 1.1 Linguagens de programação . . . 1
- 1.2 Single Assignment . . . 3
- 1.3 Continuation-Passing Style . . . 5
- 1.4 Relação entre *SA* e *CPS* . . . 6
- 1.5 Contribuições . . . 7
- 1.6 Estrutura da dissertação . . . 8

2 SINGLE ASSIGNMENT VS CONTINUATION-PASSING STYLE 9
- 2.1 Single Assignment . . . 9
  - 2.1.1 Linguagens $\mathcal{LA}$ e $\mathcal{SA}$ . . . 9
  - 2.1.2 Semântica de Transições/Estados . . . 11
  - 2.1.3 Conversão para o formato $\mathcal{SA}$ . . . 14
- 2.2 Continuation-Passing Style . . . 15
  - 2.2.1 λ-Calculus . . . 15
  - 2.2.2 λ-Calculus e o formato *CPS* . . . 17
  - 2.2.3 Linguagem $\mathcal{CPS}$ . . . 19
  - 2.2.4 Redução de programas $\mathcal{CPS}$ . . . 21
- 2.3 Relação entre $\mathcal{LA}$ e $\mathcal{CPS}$ . . . 24
  - 2.3.1 Funções de conversão . . . 24
  - 2.3.2 Bijecção ao nível sintático . . . 26
  - 2.3.3 Preservação da semântica pelas traduções . . . 29

3 EXTENSÕES 37
- 3.1 Condicionais . . . 37
  - 3.1.1 Construção *If* . . . 37
  - 3.1.2 Semântica . . . 40
- 3.2 Procedimentos . . . 41
  - 3.2.1 Procedimentos em $\mathcal{LA}$ . . . 41
  - 3.2.2 Procedimentos em $\mathcal{CPS}$ . . . 58
- 3.3 Estendendo a relação entre $\mathcal{LA}$ e $\mathcal{CPS}$ . . . 63
  - 3.3.1 Relação sintática entre programas . . . 63
  - 3.3.2 Peservação da semântica para a extensão da linguagem . . . 65

4 IMPLEMENTAÇÃO 67
- 4.1 Módulos . . . 67
- 4.2 Exemplos . . . 70

5 CONCLUSÃO 75

BIBLIOGRAFIA 79

Anexos 81

A CÓDIGO DA IMPLEMENTAÇÃO 83
- A.1 Módulo de Linguagens . . . 83

A.2 Módulo de Ferramentas 85
A.3 Módulo de Conversões 88
A.4 Módulo de Compilação 89
A.5 Módulo de Execução 91

LISTA DE FIGURAS

Figura 1	Ex. Programa DSA com "if"	4
Figura 2	Ex. Programa SSA como apresentado por Vanbroekhoven	4
Figura 3	Ex. Programa DSA como apresentado por Vanbroekhoven	5
Figura 4	Ex. Programa funcional	6
Figura 5	Ex. Programa funcional no formato CPS	6
Figura 6	Ex. Programa na Linguagem de Atribuições	10
Figura 7	Ex. Programa na Linguagem Single Assignment	11
Figura 8	Diagrama comutativo da relação sintática	26
Figura 9	Preservação da relação sintática c/ α-equivalência	28
Figura 10	Diagrama comutativo da relação semântica $\mathcal{LA} \rightarrow \mathcal{CPS}$	30
Figura 11	Diagrama comutativo da relação semântica $\mathcal{CPS} \rightarrow \mathcal{LA}$	32
Figura 12	Diagrama comutativo da relação semântica $\mathcal{LA} \Leftrightarrow \mathcal{CPS}$	33
Figura 13	Ex. comando *If* usual	38
Figura 14	Ex. programa com composição de comandos *If*	38
Figura 15	Ex. programa sem composição de comandos *If*	39
Figura 16	Árvore de comandos do exemplo da Figura 13	39
Figura 17	Ex. procedimento na linguagem $\mathcal{LA}$	42
Figura 18	Expansão do procedimento da Figura 17 dados parâmetros de invocação w e z	42
Figura 19	Ex. de procedimento com invocação de outro	43
Figura 20	Ex. Procedimento da Figura 19 com invocação de função pré-processada	44
Figura 21	Ex. Programa com procedimentos no formato *SA*	56
Figura 22	Ex. Programa da Figura 21 após compilação	57
Figura 23	Ex. Procedimento na Linguagem $\mathcal{CPS}$	59
Figura 24	Diagrama da comutação da Compilação c/ as traduções **F** e **I**	65
Figura 25	Ficheiro "examples/program1.la"	70
Figura 26	Conversão de "examples/program1.la" para $\mathcal{CPS}$	71
Figura 27	Ficheiro "examples/program2.la"	72
Figura 28	Ficheiro "examples/program3.la"	73

ACRÓNIMOS

CPS Continuation Passing Style

SA Static Assignment

SSA Static Single Assignment

DSA Dynamic Single Assignment

1 INTRODUÇÃO

1.1 LINGUAGENS DE PROGRAMAÇÃO

No universo das linguagens de programação existe uma grande variedade de paradigmas, nomeadamente *Imperativo*, *Funcional*, *Lógico*, *Orientado a Objetos* e *Orientado a Eventos*. Esta variedade tem vindo constantemente a aumentar, principalmente devido à crescente necessidade de resolução de novos problemas em novos contextos. De todos estes paradigmas há dois que destacamos - o imperativo e o funcional - tanto pela sua grande utilização como pela sua longa história e pelo grande número de estudos sobre eles realizados.

O paradigma imperativo baseia-se na noção de *estado*, entendido como um conjunto de variáveis (que correspondem a células de memória da máquina), e de comandos que alteram o estado. Um programa é uma sequência de comandos para o computador executar. Linguagens como *FORTRAN*, *ALGOL*, *Pascal* ou *C*, remontam aos primórdios do paradigma imperativo. A maioria das linguagens actuais permitem a utilização de diversos paradigmas (sendo o imperativo um deles), como *C++* e *Java*. Aliás, é cada vez mais comum novas linguagens surgirem como sendo multi-paradigma.

Até certo ponto, podemos afirmar que linguagens que acomodam o paradigma imperativo estão mais próximas das *Máquinas de Turing* (Turing [18]) e da sua forma de execução.

No paradigma funcional, um programa consiste num conjunto de funções e a sua execução está intimamente relacionada com a noção matemática de aplicação de funções. As variáveis de um programa funcional são as variáveis usadas nas declarações das funções como seus parâmetros, que são substituídos por valores concretos quando a função é invocada, mas que não sofrem alterações ao longo da execução da função.

Pelas suas características, é mais fácil estabelecer uma relação precisa entre o "input" e o "output" ("entrada" e "saída" de dados) de um programa funcional do que de um programa imperativo e, dado isto, um programador tende a cometer menos erros. Existem diversas linguagens que implementam o paradigma funcional, sendo a linguagem *Lisp* apontada como uma das primeiras implementações do paradigma. Outras linguagens funcionais usadas atualmente são *ML*, *Scheme* e *Haskell*. Para além destas, as linguagens *Python* e *Erlang*, apresentam também características funcionais, mas são linguagens multi-paradigma.

O modelo abstrato do paradigma funcional, por excelência, é o λ-calculus (Church [7]). Esta teoria matemática, contemporânea das *Máquinas de Turing*, é uma teoria de funções onde o conceito de aplicação de funções a argumentos assume um papel central.

Linguagens no paradigma imperativo são, usualmente, a escolha preferencial para a construção de programas, devido à maior proximidade ao funcionamento das máquinas físicas. As linguagens funcionais estão muitas vezes associadas à construção de programas e modelos no âmbito matemático, dada a sua relação próxima com estes conceitos. No entanto, como já referimos, com a evolução das linguagens ao longo da história a existência de linguagens multi-paradigmas tem vindo a aumentar, o que permite formas de construção de programas mais abrangentes e a partilha de conceitos entre paradigmas.

Atualmente, o modelo de computação apresenta-nos máquinas físicas que, por construção, respondem a um paradigma imperativo (sequências de comandos/bits). Estas máquinas têm as suas próprias linguagens, de baixo nível, desenvolvidas para trabalhar diretamente com as suas especificidades. Apesar de fornecerem acesso direto às capacidades da máquina, tornou-se impraticável, devido à crescente complexidade dos programas, utilizar estas linguagens de baixo nível diretamente. Ainda assim, mesmo linguagens imperativas, como as que mencionamos anteriormente, revelam-se bastante mais abstratas que as linguagens das máquinas físicas.

Por sua vez, linguagens de alto nível tornam mais simples o processo de criação de programas comparativamente com linguagens de baixo nível, pois permitem programar a um nível mais abstrato, sem nos preocuparmos com as especificidades da máquina, passando o trabalho de converter os comandos de alto nível para comandos de baixo nível a um outro "programa" designado de *compilador*. Ao compilador é, assim, incumbida a tarefa de transformar corretamente os programas em código máquina, de forma a aproveitar as particularidades da mesma, preferencialmente de forma eficiente. Parte deste processo passa por transformar os programas naquilo a que chamamos de *linguagens intermédias*. Estas linguagens têm características próprias que permitem uma análise e consequente optimização do programa original.

Em relação a linguagens intermédias, e em particular àquelas inerentes aos paradigmas imperativo e funcional, temos, respectivamente, o *Single Assignment* e o *Continuations-Passing Style*. Serão estes dois formatos que iremos utilizar como objetos para estudar a relação entre o paradigma imperativo e o funcional.

1.2 SINGLE ASSIGNMENT

Single Assignment é um formato de linguagens intermédias, imperativas, utilizadas como suporte na compilação de programas, geralmente também eles de natureza imperativa. Deste modo, ao longo do presente trabalho estas linguagens serão referenciadas como linguagens *SA*. Sendo *SA* um formato para linguagens imperativas, temos então o conceito usual de variável associado a esse paradigma. É aqui que reside a característica particular deste formato. A peculiaridade de uma linguagem *SA* consiste na limitação da atribuição de variáveis, ou seja, qualquer construção da forma "o variável x toma o valor v" para um x específico está limitada no número de ocorrências. Esta característica permite desenvolver optimizadores de código imperativo, rápidos e simples, como apontado por exemplo em [4].

Usualmente consideram-se dois tipos possíveis de linguagens *SA*, o estático e o dinâmico. O estático (*Static Single Assignment*, ou *SSA*) deve-se ao facto de a limitação ser definida a nível do código fonte, sendo que qualquer atribuição apenas pode ocorrer uma vez no código. Note-se que, por vezes, podemos considerar que uma variável após lida sem nunca ter sido inicializada (atribuída), se encontra, por defeito, escrita a partir da primeira ocorrência de leitura da mesma. Consideração esta que depende das especificidades da linguagem que quisermos considerar. O outro tipo de formato *SA* é o dinâmico (*Dynamic Single Assignment, ou* DSA), onde a restrição de uma única ocorrência de atribuição se impõe, não no código fonte, mas para cada traço de execução, ou seja em cada caminho possível no código. As diferenças entre os tipos de formato *SA* serão ilustradas, posteriormente, recorrendo a exemplos concretos.

A ideia do formato *SA* é remover informação sobre a reutilização de variáveis antes de realizar processos de optimização como, por exemplo, referido em [19]:

> "...idea of SSA is to discard information about the reuse of variables before doing optimizing transformations by allowing only one definition of each variable to appear in the program text."

A restrição imposta por este formato foi desenvolvida a partir da necessidade de simplificar a "def-use chain", que informalmente é nada mais que uma estrutura onde estão armazenados os locais de "definição-uso" de cada variável (Cytron et al. [9]). Muitas técnicas e tipos de optimizações, nomeadamente as designadas por "técnicas de corte agressivas", responsáveis pela redução significativa do tamanho do código fonte, contam com a eficiência da estrutura de "definição-uso" de variáveis. A característica inerente ao formato *SA* reduz significativamente a complexidade da estrutura em causa aumentando com isto, a eficiência de acesso à mesma. Como cada variável é apenas atribuída uma vez, temos apenas um conjunto de variáveis com

a localização onde são definidas, sendo que a cada uma dessas variáveis está associada um outro conjunto de locais onde a mesma é utilizada.

O formato *DSA* é especialmente relevante no caso da existência de ciclos e de construtores condicionais, dado que esta informação adicional sobre a reutilização de variáveis, aquando de alterações no controlo de fluxo, é útil em particular na área da paralelização e outras análises que envolvem observar o fluxo dos dados [10].

Assumindo que estamos confortáveis com algum tipo de linguagem imperativa e, em particular, com a noção de ciclos e condicionais, vejamos dois exemplos (em *C*) sobre a diferença entre os dois tipos do formato *SA*.

Esta diferença entre *SSA* e *DSA* é facilmente perceptível na presença do comando *condicional*, normalmente conhecido como "if-else". Conforme suprarreferido, a diferença no *DSA* está no facto em que num traço específico não podem existir atribuições da mesma variável.

```
if(x<z){
    x = z;
    w = 1 + x;
    return(w);
} else {
    x = 5;
    h = 2 + x;
    return(h);
}
```

Figura 1: Ex. Programa DSA com "if"

No comando condicional, dado que existem dois caminhos possíveis, temos à partida dois traços distintos, e assim sendo, como cada um destes "ramos" da condicional pertence a traços distintos, podemos eventualmente ter, para um variável específica, duas atribuições, uma em cada "ramo". O exemplo da Figura 1 ilustra um programa *DSA* (mas não *SSA*) com o construtor `if`.

Um outro exemplo, onde a diferença entre *SSA* e *DSA* é visível, surge com a utilização de ciclos. Aqui remetemos para um exemplo em [19],

```
for(1 = 0; i < 1000; i++) {
        s1: t = f(a[i]);
        s2: b[i] = g(t);
}
```

Figura 2: Ex. Programa SSA como apresentado por Vanbroekhoven

Como podemos observar na Figura 2 a variável `t` é reutilizada a cada execução do ciclo. Uma vez que hoje em dia existem aplicações

que lidam com grandes quantidades de dados, pretende-se evitar este tipo de comportamento quando queremos usar uma linguagem intermédia para compilar esses programas. Surge então, a questão da diferença entre o tipo estático e o dinâmico. No caso das linguagens dinâmicas esta situação indesejável seria evitada substituindo a variável `t` por um "array" de variáveis. Utilizando o exemplo acima vejamos como seria o código no formato dinâmico:

```
for (i = 0; i < 1000; i++) {
    s1: t[i] = f(a[i]);
    s2: b[i] = g(t[i]);
}
```

Figura 3: Ex. Programa DSA como apresentado por Vanbroekhoven

É ainda, de realçar que existem diversos métodos para transformar programas imperativos em programas no formato *SA*. Alguns exemplos dessas transformações podem ser observados com detalhe em [6, 8, 20].

1.3 CONTINUATION-PASSING STYLE

Continuation-Passing style (*CPS*) é um formato de linguagens intermédias de cariz funcional, com o principal intuito de ajudar na compilação de programas desse paradigma, em particular para facilitar a implementação de optimizações como as referidas por Sabry and Felleisen [16]:

> "Typical examples of such optimizations are loop unrolling, procedure inlining, and partial evaluation."

Tal como o próprio nome indica, o formato *CPS* ("estilo de passagem por continuações") remete-nos para uma "continuação", i. e., para um argumento extra passado a uma função cujo papel é controlar a evolução da computação, após a execução do corpo da função. Olhando em particular para o λ-calculus com continuações o que temos são abstrações com uma forma muito particular que realizam uma operação de transporte dos valores até uma outra função que os irá tratar. Por abuso de linguagem, podemos dizer que um valor é calculado e continuado para a função seguinte, que por sua vez, irá trabalhar esse valor e transportar à próxima função.

Observaremos de seguida um exemplo da utilização usual de continuações recorrendo à sintaxe do *Scheme* (Sussman and Jr. [17]). Falamos em utilização usual de continuações, referindo-nos à introdução de um parâmetro extra que representa a *continuação*, uma vez que a linguagem *Scheme* contém um comando próprio para este efeito.

```
(define (sum a b)
        (+ a b)
                )
```

Figura 4: Ex. Programa funcional

A Figura 4 representa uma função que soma dois valores, escrito numa linguagem como *Scheme*. Para a converter para o formato *CPS* é necessário adicionar um parâmetro extra (a continuação), conforme apresentado na Figura 5.

```
(define (sum a b k)
        k (+ a b)
                )
```

Figura 5: Ex. Programa funcional no formato CPS

Na Figura 5 observamos então que existe uma parâmetro `k` que representa a *continuação* e que, mais uma vez, é a função que "continuará" a execução do programa com o resultado da função anterior.

A tradução de programas funcionais para formato *CPS* é um assunto bastante estudado, em particular apontamos para [15, 17]. Um outro exemplo destas traduções é apresentado por Kelsey [13], onde o autor dá a conhecer uma tradução para o formato *CPS* do fragmento *Scheme* que considera no seu estudo. No contexto do λ-calculus existem também traduções em diversos estilos para o formato *CPS*. Plotkin [14], por exemplo, apresenta formas de simular "call-by-name" em "call-by-value" e vice-versa, recorrendo ao estilo de continuações. Surpreendentemente, como observado por Griffin [12], traduções *CPS* estão relacionadas com traduções de *dupla-negação* introduzidas por Gödel para mapear provas da lógica clássica em provas da lógica intuicionista.

1.4 RELAÇÃO ENTRE *SA* E *CPS*

A relação entre os formatos intermédios *SA* e *CPS* foi abordada por alguns autores [13, 2], principalmente ao nível do mapeamento de programas de um formato no outro.

Kelsey [13] define uma correspondência entre programas *SA* e programas *CPS* utilizando uma linguagem imperativa base, que inclui condicionais, saltos e procedimentos, e uma linguagem funcional base correspondente a um fragmento do *Scheme*. Kelsey observa uma relação bijectiva a nível de programas *SA* e *CPS*, sendo que, o único obstáculo que o impede de essa relação de forma completa, é o comando `letrec`. No entanto, ele indica algumas alternativas para a resolução desse problema como, por exemplo, o algoritmo por ele desenvolvido, que, apesar de não definir uma bijecção sintática, define

uma relação suficientemente forte que pode ser utilizada, em alguns casos, quase como se de uma bijecção se tratasse.

Appel [2] apresenta-nos uma outra abordagem para relacionar os dois paradigmas, associando as características do formato *SSA* com a utilização de funções, defendendo que *SSA* é programação funcional. O trabalho de Appel mostra que a utilização de funções permite obter os mesmos benefícios que o formato *SSA* no que toca à estrutura de "definição-uso" de variáveis. Isto deve-se ao facto de que, na utilização de funções, os nomes de variáveis estão inseridos no contexto de cada função, sendo esse contexto independente do contexto das restantes funções. Ou seja, as optimizações podem ser maioritariamente efetuadas recorrendo à estrutura "definição-uso" para o formato *SA*, podem também ser realizadas tendo em também conta os contextos e a estrutura de fluxo associada às funções.

1.5 CONTRIBUIÇÕES

Neste trabalho estuda-se de uma forma detalhada a relação entre programação imperativa no formato *SA* e programação funcional no formato *CPS*, tomando como ponto de partida o trabalho de Kelsey [13].

Para a linguagem imperativa iremos usar inicialmente uma linguagem que contém apenas atribuições e sequenciação de comandos, e posteriormente, acrescentaremos condicionais e procedimentos. Esta linguagem corresponde a uma restrição da linguagem imperativa considerada em [13], e será a base para o formato *SA*. No caso do formato *CPS*, a linguagem que iremos usar como base será o λ-calculus (estendido com condicionais e procedimentos numa fase posterior), uma linguagem mais pura que o *Scheme*, que é a linguagem considerada em Kelsey [13],

Relativamente às linguagens consideradas (incluindo as extensões com condicionais e procedimentos), estabelecemos a bijecção ao nível sintático dos programas *SA* e *CPS*, um resultado mencionado em [13] mas cujos detalhes e provas não são dados a conhecer. Neste trabalho estudamos ainda a relação ao nível semântico. Designadamente estabelecemos que a execução de programas *SA* e a redução de programas *CPS* também obedece a uma relação 1-1, para escolhas apropriadas destas duas noções. O estudo desta relação constitui uma importante lacuna na literatura que tivemos oportunidade de analisar - incluindo em [13], artigo no qual a relação ao nível semântico é esperada mas não enunciada.

Uma observação interessante, que resultou do estudo detalhado da bijecção entre programas *SA* e programas *CPS*, prende-se com o facto de o formato *SA* poder ser pensado como uma certa escolha de *representantes de classe* para uma noção de α-equivalência para programas imperativos correspondente à noção de α-equivalência para λ-termos. De facto, se transformarmos um programa imperativo num

programa *CPS* e, neste último, escolhermos nomes para as abstrações λ que não colidam nem entre si, nem com os nomes das variáveis livres, o que obtemos quando mapeamos este programa de volta ao mundo imperativo é um programa no formato *SA*.

É ainda de realçar que, durante o trabalho, foi sendo desenvolvida uma ferramenta computacional para animar os diversos conceitos envolvidos nesta dissertação, que se revelou crucial para testar eficazmente as nossas ideias.

1.6 ESTRUTURA DA DISSERTAÇÃO

No Capítulo 2 iremos construir e detalhar duas linguagens, uma para o formato *SA* e outra para o formato o *CPS*, apresentando nas secções finais a relação entre ambas, tanto a nível sintático (tradução de programas) como semântico (preservação da semântica pelas traduções de programas). No Capítulo 3, iremos apresentar primeiramente extensões das linguagens construídas no Capítulo 2 e, da mesma forma, proceder à análise ao nível sintático e semântico. O Capítulo 4 apresenta a ferramenta computacional construída para animar e testar os diversos conceitos presentes neste documento, relativos às linguagens base e às respetivas linguagens estendidas, assim como alguns exemplos ilustrativos da sua utilização. No Capítulo 5 expomos as principais conclusões do trabalho e, simultaneamente, as dificuldades mais prementes no desenvolvimento deste estudo. Este último capítulo, integra ainda, algumas sugestões quanto a investigação futura no sentido de aprofundar e complementar os resultados por nós verificados. O código relativo à ferramenta computacional, segue, no final da dissertação, em anexo.

2 SINGLE ASSIGNMENT VS CONTINUATION-PASSING STYLE

2.1 SINGLE ASSIGNMENT

Comecemos este capítulo por introduzir uma linguagem imperativa, a qual designaremos de $\mathcal{LA}$, e algumas construções que nos permitirão restringi-la de modo a obter uma linguagem no formato *SA* estático. Posteriormente iremos referir-nos a essa linguagem apenas como $\mathcal{SA}$ pois, apesar de existirem dois tipos de linguagens *Single Assignment*, a análise que iremos efetuar incide sobre o tipo estático.

2.1.1 *Linguagens $\mathcal{LA}$ e $\mathcal{SA}$*

Denotaremos por $Vars$ o conjunto de variáveis do programa e usaremos como notação para variáveis $x, y, x_1, a, b, \ldots$. O conjunto das constantes será denotado por $Const$ e representaremos as constantes por $c, c_1, \ldots$.

Definição 1. *O conjunto das expressões $\mathcal{LA}$ ($\mathcal{LA}$ abreviatura de Linguagem de Atribuições) é definido pela seguinte gramática,*

$$
\begin{aligned}
S &::= x := E; S \\
&\quad\mid \mathtt{ret}(E) \\
E &::= x \\
&\quad\mid c
\end{aligned}
$$

As letras utilizadas nas produções da gramática servirão também como meta-variáveis para representar elementos das mesmas, recorrendo-se à respetiva letra caligráfica para representar o conjunto de todas as expressões desse tipo. Por exemplo, utilizaremos S como meta-variável para uma expressão $\mathtt{ret}(E)$ (onde E é também ela uma meta-variável) com $\mathcal{S}$ e $\mathcal{E}$, denotando os conjuntos associados a essas classes sintáticas, S e E, respetivamente. Designaremos os elementos de $\mathcal{S}$ como *programas* e $\mathcal{E}$ como *expressões aritméticas*, e no caso de não existir ambiguidade iremos designa-las apenas de “expressões”.

As mesmas convenções (relativas às meta-variáveis) irão igualmente aplicar-se à linguagem $\mathcal{CPS}$, posteriormente apresentada no capítulo.

Com esta gramática podemos apenas construir programas que são sequências de atribuições terminadas por uma instrução $\mathtt{ret}(E)$, conforme é possível verificar no exemplo da Figura 6.

```
x:= a;
x:= b;
ret(x);
```

Figura 6: Ex. Programa na Linguagem de Atribuições

Conforme se apresenta na Figura 6, a variável x possui duas atribuições e, por isso, não se encontra no formato *SA*. Para que isso aconteça, teremos de definir a linguagem $\mathcal{SA}$ como uma restrição da linguagem $\mathcal{LA}$. Neste sentido, é necessário, primeiramente, definir alguns conceitos sobre a estrutura sintática que possuímos. Mais concretamente, precisamos de uma função que coleciona todas as variáveis às quais são efetuadas atribuições de valores num programa (Def. 2), bem como de uma função que selecione todas as variáveis que ocorrem num programa (Def. 3), quer estas ocorram do lado esquerdo ou do lado direito (em expressões aritméticas) de uma atribuição

Definição 2. *Seja* $assign$*:* $\mathcal{LA} \to \wp(Vars)$ *função definida da seguinte modo:*

$$\begin{aligned} assign(x := E; S) &= \{x\} \cup assign(S) \\ assign(ret(E)) &= \{\} \end{aligned}$$

Definição 3. *Seja* $var_{\mathcal{LA}}$*:* $L_A \to \wp(Vars)$ *função definida do seguinte modo:*

$$\begin{aligned} var(x := E; S) &= \{x\} \cup var(E) \cup var(S) \\ var(ret(E)) &= var(E) \\ var(x) &= \{x\} \\ var(c) &= \{\} \end{aligned}$$

Uma vez reunidas as condições para definir a linguagem $\mathcal{SA}$, colocaremos as restrições nas produções, que permitem um comportamento mais liberal. De seguida apresentaremos a definição do predicado que nos permite aferir se um programa está nesse formato:

Definição 4. *O predicado* $\mathsf{sa}(S)$*, a ler-se "o programa* S *está no formato single assignment estático", é dado indutivamente pelas seguintes regras:*

$$rtn\ \frac{}{\mathsf{sa}(ret(E))}$$

$$atr\ \frac{\mathsf{sa}(S) \quad x \notin var(E) \quad assign(S) \cap (\{x\} \cup var(E)) = \emptyset}{\mathsf{sa}(x := E; S)}$$

Note-se que para um programa $x := E; S$ estar no formato *SA*:

i x não pode ocorrer na expressão E que lhe está a ser atribuída; e

ii as variáveis que ocorrem em E não podem ser atribuídas adiante, em S.

iii S tem que estar no formato *SA*.

Dado que sabemos agora determinar quais os elementos de $\mathcal{LA}$ que se encontram no formato *SA*, apresentamos a definição da nossa linguagem $\mathcal{SA}$:

Definição 5. *Dizemos que* $S \in \mathcal{S}$ *pertence à linguagem* $\mathcal{SA}$ *se* $\mathsf{sa}(S)$. *Designaremos esta classe como programas* $\mathcal{SA}$.

Retomando o exemplo da Figura 6, podemos adaptá-lo de forma a obter um programa que respeite o predicado da Def. 4, obtendo assim um programa $\mathcal{SA}$, apresentado na Figura 7:

```
x:= a;
y:= b;
ret(y);
```

Figura 7: Ex. Programa na Linguagem Single Assignment

De referir que este tipo de tradução preserva o significado do programa- isto é, para os mesmos "valores" obtemos os mesmos resultados. Na Subseção 2.1.3, é apresentado um método de conversão da linguagem $\mathcal{LA}$ para o formato *SA*.

Concluímos esta secção com o princípio de indução sobre os programas $\mathcal{LA}$ pois iremos recorrer a ele múltiplas vezes nas próximas secções.

Proposição 1 (Princípio de indução sobre programas $\mathcal{LA}$)**.** *Seja* $P(S)$ *uma propriedade sobre o conjunto de sequências* $\mathcal{S}$. *Se,*

1. $P(\mathsf{ret}(E)),\ \forall E$
2. $P(S) \implies P(x := E; S),\ \forall S, E, x$

Então, $P(S),\ \forall S \in \mathcal{S}$

2.1.2 *Semântica de Transições/Estados*

Passemos agora a detalhar a semântica de programas em $\mathcal{LA}$, que, à semelhança de outras linguagens imperativa se baseia na noção de estado e de transições de estado. O conceito de estado não será mais do que a associação de valores ao conjunto de variáveis.

Definição 6. *Um estado é uma função total* $\sigma : \mathsf{Vars} \to \mathsf{Const}$.

O conjunto de estados será denotado por Σ *e* $\sigma, \sigma', \sigma_0, \ldots$ *serão usadas como meta-variáveis para representar estados.*

Definição 7. *Sejam* $f : A \to B$ *uma função* , $y \in A$ *e* $v \in B$. $f[y \mapsto v]$ *é a função de* A *em* B *definida do seguinte modo:*

$$f[y \mapsto v] = \begin{cases} f(x), & \text{se } x \neq y \\ v, & \text{se } x = y \end{cases}$$

Esta definição permite-nos alterar a imagem de uma função para um dado objecto de domínio, quer para funções totais quer par parciais. Esta notação para funções será conveniente tanto para representar atualizações de estado como para atualizações de *fechos*, uma noção semelhante para expressões $\mathcal{CPS}$, conforme teremos oportunidade de abordar posteriormente.

Definição 8. *Sejam* $E \in \mathcal{E}$ *e* $\sigma \in \Sigma$ *um estado. A operação de* avaliação *de* E *no estado* σ *é uma função,* $[\![.]\!]_\sigma : \mathcal{E} \to \mathrm{Const}$, *definida do seguinte modo:*

$$[\![x]\!]_\sigma = \sigma(x)$$
$$[\![c]\!]_\sigma = c$$

Note-se que caso seja necessário estender a linguagem $\mathcal{LA}$ com operações sobre constantes/variáveis, basta estender esta função de avaliação.

Uma outra noção que iremos necessitar mais à frente neste estudo e que está relacionado com o conceito de estado é, a noção de estados equivalentes, que se define da seguinte forma:

Definição 9. *Sejam* $\sigma, \sigma' \in \Sigma$. σ *e* σ' *dizem-se* equivalentes *para um programa* S *se* $\forall x \in var(S),\ \sigma(x) = \sigma'(x)$.

Uma vez definido o conceito de estado, prosseguimos com a especificação da semântica para linguagem $\mathcal{LA}$, que é dada por um sistema de transições. Este sistema corresponde a uma relação binária de transição entre *configurações*, dadas por pares (programa,estado). A "execução" de um programa corresponde, então, a uma sequência de configurações (Def. 11), os ditos pares (programa, estado) (Def. 10), que eventualmente atingem uma *configuração final* (a partir dela não existem mais transições possíveis) da qual é extraído o resultado da execução (Def. 12).

Definição 10 (Configurações em $\mathcal{LA}$)**.** *Uma* configuração *de* $\mathcal{LA}$ *é um par* (S, σ) *com* $S \in \mathcal{S}$ *e* $\sigma \in \Sigma$. *Por consequência* $\mathcal{S} \times \Sigma$ *é o conjunto de todas as configurações. Uma configuração final de* $\mathcal{LA}$ *é uma configuração da forma* $(\mathrm{ret}(E), \sigma)$, *para todo* $E \in \mathcal{E}$ *e todo* $\sigma \in \Sigma$.

Definição 11 (Relação de Transição)**.** *A* relação de transição *para* $\mathcal{LA}$ *é notada por* $\rightsquigarrow$ *e é a relação binária no conjunto de configurações, definida pela seguinte regra:*

$$assig\ \frac{}{(x := E; S, \sigma) \rightsquigarrow (S, \sigma[x \mapsto [\![E]\!]_\sigma])}$$

Representaremos por $\rightsquigarrow^n$ *a relação de transição em* n *passos definida, indutivamente da seguinte forma:*

i $\forall S, \sigma : (S, \sigma) \rightsquigarrow^0 (S, \sigma)$;

ii $\forall S, S', S'', \sigma, \sigma', \sigma'' \;:\; (S, \sigma) \rightsquigarrow^k (S'', \sigma'') \wedge (S'', \sigma'') \rightsquigarrow (S', \sigma') \implies (S, \sigma) \rightsquigarrow^{k+1} (S', \sigma')$.

Representamos por $\rightsquigarrow^*$ a relação de transição em zero ou mais passos, i.e., o fecho reflexivo e transitivo de $\rightsquigarrow$, ou seja,

$$(S, \sigma) \rightsquigarrow^* (S', \sigma') \implies (S, \sigma) \rightsquigarrow^n (S', \sigma') \text{ para algum } n \in \mathcal{N}$$

Definição 12 (Avaliação em $\mathcal{LA}$)**.** *Sejam,* $S \in \mathcal{S}$*,* $E \in \mathcal{E}$*,* $\sigma, \sigma' \in \Sigma$ *e* $v \in \mathsf{Const}$*. Se,*

$$(S, \sigma) \rightsquigarrow^* (\mathsf{ret}(E), \sigma') \wedge [\![E]\!]_{\sigma'} = v$$

então, diz se que (S, σ) avalia *em* v*.*

A proposição seguinte propõe que a execução na linguagem $\mathcal{LA}$ é determinista e termina sempre, facto que fica por demonstrar, pelo que a avaliação de um par (S, σ) produz exatamente um valor.

Proposição 2 (Determinismo e Terminação da execução em $\mathcal{LA}$)**.** *Para toda a configuração* (S, σ) *existe e é única configuração final* (S', σ')*, tal que* $(S, \sigma) \rightsquigarrow^* (S', \sigma')$

Demonstração. Por indução na estrutura de S. □

Lema 1 (Equivalência de estados)**.** *Se σ e σ′ são estados equivalentes para S, então*

$$\begin{array}{ll} (S, \sigma) & \textit{avalia em } v \\ \iff & \\ (S, \sigma') & \textit{avalia em } v \end{array}$$

Demonstração. Por indução na estrutura de S. **Caso:** S da forma $\mathsf{ret}(E)$. Sejam σ e σ′ equivalentes para $\mathsf{ret}(E)$ e $[\![E]\!]_\sigma = v$.

$$\forall x \in var(\mathsf{ret}(E)) = var(E),\ \sigma(x) = \sigma'(x)$$

Logo,

$$[\![E]\!]_{\sigma'} = [\![E]\!]_\sigma = v$$

A implicação contrária, "$(S, \sigma') \implies \ldots$" avalia em v, é análoga.

Caso: S da forma $x := E; S'$. Assumimos por hipótese que

$$\begin{array}{l} \forall \sigma, \sigma', \text{estados equivalentes} \;: \\ \quad (S', \sigma) \text{ avalia em } v \\ \quad \iff \\ \quad (S', \sigma') \text{ avalia em } v \end{array}$$

Sejam então σ e σ' estados equivalentes para S. Basta mostrar que

$$(S, \sigma) \rightsquigarrow (S', \sigma_0)$$
e
$$(S, \sigma') \rightsquigarrow (S', \sigma'_0)$$

com σ_0 e σ'_0 estados equivalentes em S'.

$$(x := E; S', \sigma) \rightsquigarrow (S', \sigma[x \mapsto [\![E]\!]_\sigma])$$
e
$$(x := E; S', \sigma') \rightsquigarrow (S', \sigma'[x \mapsto [\![E]\!]_{\sigma'}])$$

Como σ e σ' são equivalentes em S, são também equivalentes S', então $[\![E]\!]_\sigma = [\![E]\!]_{\sigma'}$ e consequentemente $\sigma[x \mapsto [\![E]\!]_\sigma]$ e $\sigma'[x \mapsto [\![E]\!]_{\sigma'}]$ são equivalentes em S'. □

Reforçando a ideia anterior, as definições 10, 11 e 12 definem, a semântica operacional da linguagem $\mathcal{LA}$.

2.1.3 *Conversão para o formato $\mathcal{SA}$*

Nesta secção vamos explicitar uma forma simples de converter um programa $\mathcal{LA}$ para o formato *SA*. Para isso utilizaremos uma função finita $Vars \rightharpoonup \mathbb{N}$ que nos indica o último índice usado na renomeação de uma dada variável, i.e., a última versão da variável.

Seja S o programa $x := y; S'$, e f uma função finita tal que, $f(x) = 3$ e $f(y) = 7$. Como x e y já se encontram definidos pela função f, significa que, a dada altura, já foram atribuídos. Então, de forma a que o comando reflita corretamente o valor atual das variáveis x e y, teríamos de gerar o código $x_4 := y_7; S'$ e a atualizamos a função f(utilizando a mesma notação apresentada na atualização de estados), $f[x \mapsto f(x)+1][y \mapsto f(y)+1]$, pois verifica-se a existência de uma nova atribuição a x mas y apenas está a ser utilizado e, portanto, mantemos na nossa estrutura o valor de y e aumentamos o de x.

Iremos usar $\neg f(x)$ para indicar que $f(x)$ não está definida.

Passemos então à definição formal desta conversão.

Definição 13. *Seja* $\boldsymbol{sa} : \mathcal{S} \times \mathcal{F} \to \mathcal{S}$ *uma função, onde* $\mathcal{F}$ *representa o universo das funções finitas.* ***sa*** *fica definida da seguinte forma,*

$$\boldsymbol{sa}(x := c; S, f) = x_{f(x)+1} := c; \boldsymbol{sa}(S, f[x \mapsto f(x)+1])$$
$$\boldsymbol{sa}(x := c; S, f) = x_1 := c; \boldsymbol{sa}(S, f[x \mapsto 1]), \neg f(x)$$

$$\boldsymbol{sa}(x := y; S, f) = \begin{cases} x_{f(x)+1} = y_{n'}; \boldsymbol{sa}(S, f[x \mapsto f(x)+1]) \\ \quad \textit{se } f(x) \wedge f(y) = n' \\ x_{f(x)+1} = y_1; \boldsymbol{sa}(S, f[x \mapsto f(x)+1][y \mapsto 1]) \\ \quad \textit{se } f(x) \wedge \neg f(y) \\ x_1 := y_{n'}; \boldsymbol{sa}(S, f[x \mapsto 1]) \\ \quad \textit{se } \neg f(x) \wedge f(y) = n' \\ x_1 := y_1; \boldsymbol{sa}(S, f[x \mapsto 1][y \mapsto 1) \\ \quad \textit{se } \neg f(x) \wedge \neg f(y) \end{cases}$$

$$\boldsymbol{sa}(ret(c), f) = ret(c)$$

$$\boldsymbol{sa}(ret(y), f) = \begin{cases} ret(y_1) & , \neg f(y) \\ ret(y_n) & , f(y) = n \end{cases}$$

2.2 CONTINUATION-PASSING STYLE

2.2.1 λ-*Calculus*

Segundo Barendregt [3],

> "The lambda calculus is a type free theory about functions as rules..."

, o λ-calculus é uma teoria de funções desprovida de tipos e que vê as funções como regras. Regras essas que nos permitem ir de um conjunto de parâmetros para um resultado. O conjunto de expressões do λ-calculus é dado pela seguinte gramática,

$$\begin{aligned} M, N ::= & \; x \\ & \mid (\lambda x.M) \\ & \mid (MN) \end{aligned}$$

onde, conforme explicitado para a gramática de $\mathcal{LA}$, x é uma variável. As expressões de λ-calculus são denominadas de λ-*termos*. Os λ-termos da forma $(\lambda x.M)$ são chamados de *abstrações* e os λ-termos da forma (MN) são chamados de aplicações.

Podemos, eventualmente, omitir os parêntesis nas expressões, por exemplo $(\lambda x.M)$ ser representado por $\lambda x.M$. Nestes casos assume-se sempre que a associação é efetuada à esquerda, ou seja, MNM' é o mesmo que $(MN)M'$.

2.2.1.1 β-*Redução*

O conceito de β-*redução* é fundamental na definição da semântica usual do λ-calculus. A β-redução assenta na ideia de substituição e traduz o "resultado" de aplicar uma função (uma abstração lambda) a um argumento.

Não obstante, para uma melhor compreensão do conceito, é necessário conhecer previamente os conceitos de variáveis ligadas e livre Uma *variável ligada* é uma variável que ocorre numa abstração, por exemplo, em $(\lambda x.M)$, x, enquanto que uma *variável livre* é aquela à qual não está associada nenhuma abstração, por exemplo, $(\lambda x.xz)$ tem z como única variável livre.

A notação que usaremos para a substituição é $M[N/y]$, que significa, "substituir a variável y pelo termo N no termo M". A definição recursiva da operação de susbtituição é a seguinte:

$$(\lambda x.M)[N/z] = \begin{cases} (\lambda x.M[N/z]), & \text{se } x \neq z \\ (\lambda x.M), & \text{se } x = z \end{cases}$$

$$x[N/z] = \begin{cases} x, & \text{se } x \neq z \\ N, & \text{se } x = z \end{cases}$$

$$(M_1 M_2)[N/z] = (M_1[N/z]M_2[N/z])$$

Esta definição por si só, pode levar à captura de variáveis livres como veremos adiante.

Por sua vez, a β-redução (representada por $\rightarrow_\beta$) dá-nos uma regra de cálculo para o valor de uma função aplicada a um argumento (Def. 14)

Definição 14. *Seja* $\rightarrow_\beta$ *a relação binária em* λ*-termos dada pela seguinte regra*

$$(\lambda x.M)N \rightarrow_\beta M[N/x]$$

Note-se que $\rightarrow_\beta$ representa como é usual o fecho de compatibilidade da relação β, ou seja, num termo da forma $(\lambda x.M)N$ podemos efetuar a redução tal como indica a regra ou procurar outras possíveis reduções (ou como são designadas usualmente, β-redexes) nos sub-termos M e N.

Para ilustrar a captura de variáveis livres, tomemos por exemplo o termo:

$$(\lambda x.(\lambda z.M))z$$

, note-se o facto de z aparecer tanto livre como ligada, ao efetuar a redução temos

$$(\lambda z.M)[z/x]$$

Ou seja, se x ocorrer em M então iremos estar a "ligar" as ocorrências de x à abstração ($\lambda z. \ldots$) o que não é o comportamento desejado.

Este problema é resolvido com a simples renomeação da variável ligada de forma a que a variável livre não seja então capturada por uma abstração.

2.2.1.2 α *Equivalência*

O conceito de α-equivalência, tal como a β-redução, é uma relação binária em λ-termos e permite estabelecer que dois λ-termos são iguais, se apenas diferem nos "nomes" das variáveis ligadas. Por exemplo:

$$(\lambda x.x)$$
$$(\lambda y.y)$$

são λ-termos α-equivalentes. Usamos a notação $M =_\alpha N$ quando M é α-equivalente a N.

Definição 15. *Seja $=_\alpha$ uma relação binária definida do seguinte modo,*

$$\begin{aligned} \lambda x.M =_\alpha \lambda y.M', &\quad \textit{se } M =_\alpha M' \\ (\lambda x.M) =_\alpha (\lambda y.M'), &\quad \textit{se } M =_\alpha M'[x/y] \\ MN =_\alpha M'N', &\quad \textit{se } M =_\alpha M' \wedge N =_\alpha N' \end{aligned}$$

2.2.2 λ-*Calculus e o formato* CPS

Nesta secção apresentamos o formato *CPS* para λ-calculus, bem como uma forma de converter λ-termos para esse formato.

Observaremos também a conversão de um λ-termo em particular, dado que será a base para a linguagem, no formato *CPS*, que iremos construir e utilizar para estudar a relação entre os dois formatos intermédios.

Conforme referido anteriormente, o formato *CPS* consiste, informalmente, numa forma de escrever programas funcionais em que cada função recebe um argumento extra chamado de *continuação*. Argumento este que representa uma outra função responsável por tratar o valor de retorno da primeira.

A linguagem que iremos apresentar vai difere em larga medida da apresentada por Kelsey em [13]. O autor apresenta uma linguagem que consiste num subconjunto de outra: o *Scheme*. Apresenta-nos ainda uma linguagem na qual as continuações apenas se revelam associadas aos valores de retorno. Tal como no λ-calculus puro, cada λ-termo é considerado ele próprio uma função única, e então, temos que no formato *CPS* todo o λ-termo tem uma continuação associada. Na linguagem *CPS* que iremos construir nesta dissertação, serão apresentados termos menos "restritos" na medida em que apenas λ-termos que representam expressões aritméticas terão uma continuação associada.

Prosseguimos então com a apresentação da conversão de λ-termos para o formato *CPS*.

Definição 16. *Para todo o λ-termo* N *seja* $\overline{N}$ *função sobre λ-termos definida da seguinte forma:*

$$\overline{x} = x$$
$$\overline{\lambda x.M} = \lambda k.k(\lambda x.\overline{M})$$
$$\overline{M\ N} = \lambda k.\overline{M}(\lambda f.\overline{N}(\lambda v.fvk))$$

com M, N *λ-termos e* x *uma variável.*

A função permite converter λ-termos para um formato de *continuações* com método de avaliação "call-by-value". Esta tradução CPS refere-se a uma versão simplificada da conversão apresentada por Fischer [11]. Por sua vez, Sabry and Felleisen oferecem uma versão mais otimizada desta conversão em [16].

A construção que iremos usar como base para a nossa linguagem no formato *CPS* é o "binder". Esta construção permite associar um valor/termo a uma variável. Dada a sua importância para a nossa linguagem, iremos explicitar a sua dedução e simplificação utilizando a tradução para o formato *CPS* que acabamos de apresentar. Usualmente representamos este construtor em *Scheme* e noutras linguagens idênticas, da seguinte forma

$$\text{let } x = t \text{ in } u$$

onde x é a variável à qual queremos associar um valor, neste caso representado por t, dentro do contexto u. Num formato de λ-calculus mais puro, teríamos o seguinte termo:

$$(\lambda x.u)t$$

Dada esta explicação informal, podemos agora apresentar a dedução da nova regra.

$$\begin{aligned}
\overline{\text{let } x = t \text{ in } u} &= \overline{(\lambda x.u)t} \\
&= \lambda k.\overline{(\lambda x.u)}(\lambda f.\overline{t}(\lambda v.fvk)) \\
&= \lambda k.(\lambda k.k(\lambda x.\overline{u}))(\lambda f.\overline{t}(\lambda v.fvk)) \\
&\to_\beta \lambda k.(\lambda f.\overline{t}(\lambda v.fvk))(\lambda x.\overline{u}) \\
&\to_\beta \lambda k.\overline{t}(\lambda v.(\lambda x.\overline{u})vk) \\
&\to_\beta \lambda k.\overline{t}(\lambda v.\overline{u}[v/x]k) \\
&=_\alpha \lambda k.\overline{t}(\lambda x.\overline{u}k)
\end{aligned}$$

Vejamos a título de exemplo as seguintes contruções e respectivas traduções para CPS.

- $\overline{(\lambda x.(\lambda y.(\lambda z.\dots)))} = (\lambda k.k(\lambda x.(\lambda k.k(\lambda y.(\lambda k.k(\lambda z.\dots))))))$
- $\overline{(\lambda x_1.(\lambda x_2.\dots)t_2)t_1} \equiv \lambda k.t_1(\lambda x_1.(\lambda k t_2(\lambda x_2.\ \dots k)k))$
- $\overline{t} \equiv \lambda k.kt$, sendo t uma variável ou uma constante.

2.2.3 *Linguagem $\mathcal{CPS}$*

Na Subseção 2.2.2 apresentamos alguns exemplos da conversão de λ-termos para o formato de continuações. Recorremos de seguida a esses mesmos exemplos para construir uma linguagem funcional capaz de oferecer alguns dos mecanismos usuais de uma qualquer linguagem de programação, nomeadamente os mecanismos usualmente referidos como "expressões" e "retorno". As nossas expressões serão apenas constantes ou variáveis e então, no formato de continuações, serão da forma:

- $\lambda k.kx$ ou
- $\lambda k.kc$

sendo x uma variável e c uma contante. No formato de continuações o "retorno" prende-se com a passagem de uma "expressão" a um outro λ-termo pelo que os "retornos" assumirão a seguinte forma:

$$\lambda k.kB$$

onde B é uma "expressão". Utilizamos B pois denominamos as "expressões" no contexto funcional de termos base. Regressemos então

ao exemplo anterior. Habitualmente, o mecanismo que se segue, é denominado de "binder", consistindo na associação de uma variável a um λ-termo:

$$\overline{(\lambda x_1.(\lambda x_2.\dots)N_2)N_1} \equiv \lambda k.N_1(\lambda x_1.(\lambda k N_2(\lambda x_2.\ \dots k)k))$$

O parâmetro k é o parâmetro que irá ser passado consecutivamente até que seja necessário passar/continuar algum valor para outro termo. Ora, dado o facto de termos construído o nosso elemento de "retorno" de forma a receber o termo/continuação, não existe a necessidade de efetuar a passagem do elemento de continuação em todos os termos até se atingir o momento da sua utilização. Então, de modo a reduzir a verbosidade dos nossos programas, vamos simplificar os os termos a que designamos de "binders". Assim, apenas as "expressões" $\mathcal{CPS}$ terão uma continuação associada.

$$\overline{(\lambda x_1.(\lambda x_2.\dots)N_2)N_1} \equiv N_1(\lambda x_1.N_2(\lambda x_2.\ \dots))$$

Note-se que, no que concerne à nossa linguagem, os λ-termos N_1 e N_2 seriam "expressões". Neste sentido, a Def. 17 apresenta a gramática da nossa linguagem funcional.

Definição 17. *Seja $\mathcal{CPS}$ a linguagem de λ-termos definida pela seguinte gramática.*

$$
\begin{aligned}
M &::= B(\lambda x.M) \\
&\quad |\ (\lambda k.kB) \\
B &::= \lambda k.kx \\
&\quad |\ \lambda k.kc
\end{aligned}
$$

onde x, k são variáveis (assumindo k carácter especial, pois representa uma continuação), e c uma constante. Note-se desde já a introdução de um novo elemento: as constantes. Estes elementos representam valores "imutáveis". Iremos tomar $\mathcal{CPS} = \mathcal{M} \cup \mathcal{B}$. Designaremos os elementos $\mathcal{M}$ por *expressões* e os $\mathcal{B}$ por *expressões base*.

De salientar que, tal como em $\mathcal{LA}$, existe um principio de recursão associado à gramática, princípio este que iremos utilizar na construção de funções sobre a mesma.

Segue-se a adaptação de algumas funções básicas, geralmente definidas para λ-calculus, para o nosso fragmento de λ-termos. Iremos fazer o sobrecarregamento do nome das funções de forma a abranger as diferentes classes da gramática (conforme nas 18 e 19).

Definição 18. *Seja* BV*:* $\mathcal{CPS} \rightarrow \wp(Vars)$ *função definida do seguinte modo:*

$$
\begin{aligned}
BV(B(\lambda x.M)) &= \{x\} \cup BV(B) \cup BV(M) \\
BV(\lambda k.kB) &= \{k\} \cup BV(B) \\
BV(\lambda k.kx) &= \{k\} \\
BV(\lambda k.kc) &= \{k\}
\end{aligned}
$$

Esta função coleciona todas a *variáveis ligadas* de um λ-termo.

Definição 19. *Seja* $FV : \mathcal{CPS} \rightarrow \wp(Vars)$ *a função definida do seguinte modo:*

$$
\begin{aligned}
FV(B(\lambda x.M)) &= FV(B) \cup (FV(M) \setminus \{x\}) \\
FV(\lambda k.kB) &= FV(B) \setminus \{k\} \\
FV(\lambda k.kx) &= \{x\} \\
FV(\lambda k.kc) &= \{\}
\end{aligned}
$$

Esta função coleciona todas a *variáveis livres* de uma expressão em $\mathcal{CPS}$.

Passamos agora a definir, tal como fizemos anteriormente para a linguagem $\mathcal{LA}$, duas funções que colecionam detalhes relevantes sobre programas $\mathcal{CPS}$, fazendo-o à custa das funções que acabamos de enunciar. Por abuso de notação iremos sobrecarregar alguns nomes das funções definidos anteriormente para a linguagem $\mathcal{LA}$ (assign e var) e, caso necessário, usar em índices (CPS e LA) para distinguir entre as funções para expressões $\mathcal{CPS}$ e para expressões $\mathcal{LA}$.

Definição 20. *A função* var*:* $\mathcal{CPS} \to \wp(Vars)$ *é função definida do seguinte modo:*

$$var(T) = (BV(T) \cup FV(T)) \setminus \{k\}$$

Esta função coleciona todas as variáveis que ocorrem (de modo ligado ou livre) numa expressão $\mathcal{CPS}$, exceto a variável de carácter especial k.

Definição 21. *A função* $assign$ *:* $\mathcal{CPS} \to \wp(Vars)$ *é definida do seguinte modo:*

$$assign(T) = BV(T) \setminus \{k\}$$

$assign$ coleciona todas as variáveis que ocorrem em abstrações lambda de expressões $\mathcal{CPS}$, exceto a variável de carácter especial k.

Note-se que a remoção da variável de continuação nestas funções deve-se ao facto de esta ter um carácter de certa forma "imutável". Pretendemos com estas funções selecionar variáveis relevantes ao significado do programa e não à sua avaliação.

Mais uma vez concluímos a secção com um princípio de indução (Prop. 3), necessário para a análise efetuada nas secções seguintes, sobre elementos M de $\mathcal{CPS}$.

Proposição 3 (Princípio de indução sobre programas $\mathcal{CPS}$)**.** *Seja* $P(M)$ *uma propriedade sobre o conjunto de termos* $\mathcal{M}$*.*

Se,

1. $P(\lambda k.kB),\ \forall B$
2. $P(B) \implies P(B(\lambda x.M)),\ \forall M, B$

Então, $P(M),\ \forall M \in \mathcal{M}$

2.2.4 *Redução de programas* $\mathcal{CPS}$

Atendendo à gramática de programas $\mathcal{CPS}$ e à noção de β-redex no λ-calculus em $\mathcal{CPS}$, temos apenas β-redex da forma $B(\lambda x.M)$. De facto, conforme B representa uma variável ou consoante, temos dois casos possíveis:

1. $(\lambda k.kx)(\lambda x.M)$, com $B = (\lambda k.kx)$
2. $(\lambda k.kc)(\lambda x.M)$, com $B = (\lambda k.kc)$

Assim, em expressões $\mathcal{CPS}$, a relação de β-redução será definida pelas duas seguintes regras:

1. $(\lambda k.ky)(\lambda x.M) \to_\beta M[y/x] \qquad (\beta_{var})$

2. $(\lambda k.kc)(\lambda x.M) \rightarrow_\beta M[c/x]$ (β_c)

Note-se que esta relação em λ-calculus corresponde a 2 passos β. Por exemplo, em $\mathcal{CPS}$, não podemos escrever o λ-termo $(\lambda x.M)y$ que resulta de fazer apenas uma redução β a partir do λ-termo $(\lambda k.ky)(\lambda x.M)$.

Assim a nossa relação de redução $(\rightarrow_\beta)$ fica definida do seguinte modo:

Definição 22. *Seja $\rightarrow_\beta$ a relação binária definida pelas seguintes regras.*

$$\beta_{var}\frac{}{(\lambda k.ky)(\lambda x.M) \rightarrow^2_\beta M[y/x]}$$

$$\beta_c\frac{}{(\lambda k.kc)(\lambda x.M) \rightarrow^2_\beta M[c/x]}$$

Note-se que esta relação é muito mais restrita que a usal para λ-calculus, o que se deve principalmente ao facto de termos tido como objectivo aproximar o significado de ambas as linguagens ($\mathcal{LA}$ e $\mathcal{CPS}$). Daqui sai então que não iremos considerar o fecho de compatibilidade da β-redução mas apenas o fecho reflexivo e transitivo. Como consequência, para cada expressão $\mathcal{CPS}$ existe no máximo um β-redex.

2.2.4.1 *Semântica de redução com fechos*

Antes de relacionarmos as linguagens $\mathcal{CPS}$ e $\mathcal{LA}$ vamos ainda apróximar as suas formas de execução introduzindo na linguagem $\mathcal{CPS}$ o conceito de *fecho* e da relação de programas no contexto de um fecho. Um fecho, como o nome sugere, servirá para completar o "significado de programa" na expressão $\mathcal{CPS}$, associando valores às variáveis livres do programa.

A utilização de fechos é bastante útil particularmente quando queremos dar significado a linguagens funcionais num contexto computacional, dado que a noção de função nessas linguagens é diferente da definição oferecida quando trabalhamos com máquinas de von Neumann (Appel [1]).

> "...the notion of function definition is more primitive on a von Neumann machine than in the CPS. ... have "free variables": expressions can refer to variables defined outside de innermost-enclosing function."

Tomemos por exemplo o λ-termo $(\lambda x.(\lambda y.xy))5$. Podemos observar este termo tanto pelo conceito de redução, onde teríamos uma redução para $(\lambda y.xy)[5/x]$, como pelo conceito de fecho, onde teríamos o termo $(\lambda y.xy)$ juntamente com um fecho que associa o valor 5 à variável x.

Neste sentido, apresentamos as definições 23 e 24, bem como o lema 2, relativos à nossa noção de fecho:

Definição 23. *Dada uma expressão* $M \in \mathcal{CPS}$, *um* fecho *para* M *é uma função parcial* $\delta : D \rightharpoonup \mathsf{Const}$ *com* $FV(M) \subseteq D \in \wp(\mathsf{Vars})$. *Denotaremos o conjunto de "todos os fechos por"* Δ, *ou seja,* Δ *é o conjunto de todas as funções de* $D \in \wp(\mathsf{Vars})$ *em* Const.

Definição 24. *Seja* δ *um fecho para* B. *A operação de avaliação,* $\langle\!\langle B \rangle\!\rangle_\delta$, *no fecho* δ *está definida do seguinte modo:*

$$\langle\!\langle \lambda k.kx \rangle\!\rangle_\delta = \delta(x),$$
$$\langle\!\langle \lambda k.kc \rangle\!\rangle_\delta = c$$

Lema 2. *Se* δ *é fecho para* $B(\lambda x.M_0)$ *então* $\delta[x \mapsto \langle\!\langle B \rangle\!\rangle_\delta]$ *é fecho para* M_0

Demonstração. Ora $FV(M_0) = FV(M) \setminus \{x\}$, como δ é fecho para M,
$\forall y \in FV(M)$, $\delta(y)$ está definido, e então
$\forall y \in FV(M_0)$, $\delta[x \mapsto \langle\!\langle B \rangle\!\rangle_\delta](y)$ está definido.
Logo $\delta[x \mapsto \langle\!\langle B \rangle\!\rangle_\delta]$ é fecho para M_0. □

Com isto mostramos que a relação de β-redução que definimos para $\mathcal{CPS}$ preserva a noção de fecho, pelo que vamos estender a noção de β-redução para $\mathcal{CPS}$ no contexto de fechos, conforme mostra a definição que se segue.

Definição 25. *A relação binária* $\rightarrow_\beta$ *sobre pares* (M, δ) *é definida pela seguinte regra:*

$$\beta_{base}\frac{}{(B(\lambda x.M), \delta) \rightarrow_\beta (M, \delta[x \mapsto \langle\!\langle B \rangle\!\rangle_\delta])}$$

Representamos por $\rightarrow^*_\beta$ *a relação em zero ou mais passos, i.e., o fecho reflexivo e transitivo de* $\rightarrow_\beta$ *e por* $\rightarrow^n_\beta$ *a relação em* n*-passos, definida indutivamente da seguinte forma;*

i $\forall M, \delta$:

$$(M, \delta) \rightarrow^0_\beta (M, \delta)$$

ii $\forall M, M', M'', \delta, \delta', \delta''$:

$$(M, \delta) \rightarrow^k_\beta (M'', \delta'') \wedge (M'', \delta'') \rightarrow_\beta (M', \delta') \implies (M, \delta) \rightarrow^{k+1}_\beta (M', \delta')$$

Para esta relação $\rightarrow_\beta$ temos também uma noção de *configuração final*, sendo esta um par da forma $((\lambda k.kB), \delta)$ para algum B e δ fecho de B.

Esta noção de $\rightarrow_\beta$ com fecho é semelhante à usual para λ-calculus, pois uma substituição é representada por uma associação num fecho e múltiplas substituições de uma mesma variável representadas por uma "actualização" do valor associado a uma variável no fecho.

Definição 26 (Avaliação em $\mathcal{CPS}$)**.** *Sejam* $M \in \mathcal{M}$, δ *fecho para* M *e* $v \in \mathsf{Const}$. *Se*

$$(M, \delta) \rightarrow^*_\beta (\lambda k.kB, \delta') \wedge \langle\!\langle B \rangle\!\rangle_{\delta'} = v$$

então, diz se que (M, δ) *avalia em* v.

2.3 RELAÇÃO ENTRE $\mathcal{LA}$ E $\mathcal{CPS}$

Comecemos por estabelecer as funções que nos permitem converter entre as linguagens $\mathcal{LA}$ e $\mathcal{CPS}$. Posteriormente, prova-se algumas propriedades que nos vão ser úteis para demonstrar que existe uma relação bijectiva entre estas duas linguagens.

2.3.1 *Funções de conversão*

Conforme se apresenta na Def. 27, a função **F** que converte programas na linguagem $\mathcal{CPS}$ para a linguagem $\mathcal{LA}$.

Definição 27. *A função **F**: $\mathcal{CPS} \to \mathcal{LA}$ está definida do seguinte modo:*

$$\begin{aligned} \boldsymbol{F}(B(\lambda x.M)) &= x := \boldsymbol{F}(B);\ \boldsymbol{F}(M) \\ \boldsymbol{F}(\lambda k.kB) &= \mathrm{ret}(\boldsymbol{F}(B)) \\ \boldsymbol{F}(\lambda k.kx) &= x \\ \boldsymbol{F}(\lambda k.kc) &= c \end{aligned}$$

De igual forma a função, **I** converte programas na linguagem $\mathcal{LA}$ para a linguagem $\mathcal{CPS}$(Def. 28).

Definição 28. *Seja $P \in \mathcal{LA}$. A função **I**: $\mathcal{LA} \to \mathcal{CPS}$ está definida do seguinte modo:*

$$\begin{aligned} \boldsymbol{I}(x := E;\ S) &= \boldsymbol{I}(E)(\lambda x.\boldsymbol{I}(S)) \\ \boldsymbol{I}(\mathrm{ret}(E)) &= (\lambda k.k\ \boldsymbol{I}(E)) \\ \boldsymbol{I}(x) &= \lambda k.kx \\ \boldsymbol{I}(c) &= \lambda k.kc \end{aligned}$$

Também a capacidade de converter fechos em estados é necessária. Para isso, e uma vez que um estado é uma função total em $Vars$ e um fecho é uma função parcial, necessitamos de totalizar a função fecho utilizando uma contante quando a queremos converter em estado.

A função que converte fechos em estados é designada cs.

Definição 29.

$$\begin{aligned} \mathrm{cs} : \Delta &\to \Sigma \\ \delta &\mapsto \delta_c \end{aligned}$$

$$\begin{aligned} \delta_c : Vars &\to \mathrm{Const} \\ x &\mapsto \begin{cases} \delta(x) & , x \in \mathrm{dom}(\delta) \\ c & , x \notin \mathrm{dom}(\delta) \end{cases} \end{aligned}$$

A função que converte estados em fechos é designada de sc:

Definição 30.

$$\begin{aligned} sc : \Sigma \times \wp(Vars) &\to \Delta \\ (\sigma, X) &\mapsto \sigma|_X \end{aligned}$$

onde a notação $f|_X$ *representa a restrição da função* f *aos elementos do conjunto* X.

Nos lemas 3 e 4 apresentamos dois resultados que relacionam fechos, estados e as traduções entre $\mathcal{LA}$ e $\mathcal{CPS}$.

Lema 3. *Seja* δ *um fecho para* B. $\langle\!\langle B \rangle\!\rangle_\delta = [\![\boldsymbol{F}(B)]\!]_{cs(\delta)}$

Demonstração. Seja δ um fecho para B.
Caso $B = \lambda k.kc$.

$$\mathbf{F}(\langle\!\langle \lambda k.kc \rangle\!\rangle_\delta) = [\![\mathbf{F}(\lambda k.kc)]\!]_{cs(\delta)}$$
$$\Longleftrightarrow [\text{Def. } \langle\!\langle . \rangle\!\rangle \text{ e } \mathbf{F}]$$
$$c = [\![c]\!]_{cs(\delta)}$$
$$\Longleftrightarrow [\text{Def. } [\![.]\!]_{cs(\delta)} \text{ e } \mathbf{F}]$$
$$c = c$$

Caso $B = \lambda k.kx$.

$$\mathbf{F}(\langle\!\langle \lambda k.kx \rangle\!\rangle_\delta) = [\![\mathbf{F}(\lambda k.kx)]\!]_{cs(\delta)}$$
$$\Longleftrightarrow [\text{Def. } \langle\!\langle . \rangle\!\rangle \text{ e } \mathbf{F}]$$
$$\delta(x) = [\![x]\!]_{cs(\delta)}$$
$$\Longleftrightarrow [\text{Def. } [\![.]\!]_{cs(\delta)}]$$
$$\delta(x) = cs(\delta)(x)$$
$$\Longleftrightarrow [\text{Def. } cs]$$
$$\delta(x) = \delta_c(x)$$
$$\Longleftrightarrow [\text{Def. } \delta_c \text{ e } x \in dom(\delta), \text{ pois } \delta \text{ é fecho de } B]$$
$$\delta(x) = \delta(x)$$

□

Lema 4. *Sejam* σ *um estado,* $E \in \mathcal{E}$, $X \subseteq Vars$. *Então* $FV(\boldsymbol{I}(E)) \subseteq X \implies [\![E]\!]_\sigma = \langle\!\langle \boldsymbol{I}(E) \rangle\!\rangle_{sc(\sigma, X)}$

Demonstração. Seja σ um estado arbitrário.
Caso $E = c$. Trivial dado que c é uma constante (valor).

Caso $E = x$.

$$\llbracket x \rrbracket_\sigma = \langle\!\langle \mathbf{I}(x) \rangle\!\rangle_{sc(\sigma, X)}$$
$$\iff [\text{Def. } \mathbf{I}]$$
$$\llbracket x \rrbracket_\sigma = \langle\!\langle \lambda k.kx \rangle\!\rangle_{sc(\sigma, X)}$$
$$\iff [\text{Def. } \llbracket . \rrbracket \text{ e } \langle\!\langle . \rangle\!\rangle]$$
$$\sigma(x) = sc(\sigma, X)(x)$$
$$\iff [\text{Def. } sc]$$
$$\sigma(x) = \sigma|_X(x)$$
$$\iff [x \in FV(\mathbf{I}(x)) \subseteq X]$$
$$\sigma(x) = \sigma(x)$$

□

Tendo em consideração os dois resultados anteriores, podemos afirmar que fechos e estados são estruturas "iguais", ou seja, as funções sc e cs produzem fechos e estados equivalentes aos estados e fechos, respectivamente, que recebem como argumentos.

2.3.2 *Bijecção ao nível sintático*

As transformações **F** e **I** são inversas e que, portanto, existe uma relação bijectiva entre os programas das linguagens $\mathcal{LA}$ e $\mathcal{CPS}$.

O diagrama da Figura 8 explicita aquilo que pretendemos mostrar.

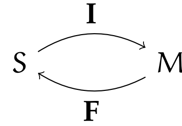

Figura 8: Diagrama comutativo da relação sintática

Proposição 4.

1. $\forall B \in \mathcal{B}, \boldsymbol{I} \circ \boldsymbol{F}(B) = B$
2. $\forall M \in \mathcal{M}, \boldsymbol{I} \circ \boldsymbol{F}(M) = M$

Demonstração.

1. **Caso** $B = \lambda k.kx$.

$$\mathbf{I}(\mathbf{F}(\lambda k.kx)) = \mathbf{I}(x) = \lambda k.kx$$

Caso $B = \lambda k.kc$.

$$\mathbf{I}(\mathbf{F}(\lambda k.kc)) = \mathbf{I}(c) = \lambda k.kc$$

2. Por indução em M.

   Caso $M = \lambda k.kB$.

   $$\mathbf{I}(\mathbf{F}(\lambda k.kB)) = \mathbf{I}(ret(\mathbf{F}(B))) = \lambda k.k\mathbf{I}(\mathbf{F}(B))$$
   $\mathbf{I}(\mathbf{F}(B)) = B$, pelo ponto 1.

   Logo, $\mathbf{I}(\mathbf{F}(\lambda k.kB)) = (\lambda k.kB)$.

   Caso $M = B(\lambda x.M')$.

   $$\begin{aligned}\mathbf{I}(\mathbf{F}(B(\lambda x.M'))) &= \mathbf{I}(x := \mathbf{F}(B); \mathbf{F}(M')) \\ &= \mathbf{I}(\mathbf{F}(B))(\lambda x.\mathbf{I}(\mathbf{F}(M'))).\end{aligned}$$

   $\mathbf{I}(\mathbf{F}(B)) = B$, pelo ponto 1.
   $\mathbf{I}(\mathbf{F}(M')) = M'$, por hipótese de indução.
   Logo, $\mathbf{I}(\mathbf{F}(B(\lambda x.M'))) = B(\lambda x.M')$.

□

Proposição 5.

1. $\forall E \in \mathcal{E}, \boldsymbol{F} \circ \boldsymbol{I}(E) = E$

2. $\forall S \in \mathcal{S}, \boldsymbol{F} \circ \boldsymbol{I}(S) = S$

Demonstração. Análoga à anterior. □

Para observarmos o resultado, sobre a α-equivalência, que referimos no capítulo introdutório, iremos primeiramente introduzir os lemas 5 e 6 que auxiliam a construção deste resultado.

Lema 5.

1. $\forall B \in \mathcal{B}, var(B) = var(\boldsymbol{F}(B)) \wedge assign(B) = assign(\boldsymbol{F}(B))$

2. $\forall M \in \mathcal{M}, var(M) = var(\boldsymbol{F}(M)) \wedge assign(M) = assign(\boldsymbol{F}(M))$

Demonstração. 1. **Caso:** B da forma $\lambda k.kx$. $\mathbf{F}(\lambda k.kx) = x$. $var(\lambda k.kx) = x = var(x)$ e $assign(\lambda k.kx) = \{\} = assign(x)$.

   Caso: B da forma $\lambda k.kc$. $\mathbf{F}(\lambda k.kc) = c$. $var(\lambda k.kc) = \{\} = var(c)$ e $assign(\lambda k.kc) = \{\} = assign(c)$.

2. Por indução na classe sintática de programas $\mathcal{M}$.

   Caso: M da forma $\lambda k.kB$. $\mathbf{F}(\lambda k.kB) = ret(\mathbf{F}(B))$. No caso anterior já verificamos o resultado para $B \in \mathcal{B}$. Assim temos que:

   $var(\lambda k.kB) = var(B) = var(\mathbf{F}(B)) = var(ret(\mathbf{F}(B)),$

   e que

   $assign(\lambda k.kB) = \{\} = assign(ret(\mathbf{F}(B))).$

Caso: M da forma $B(\lambda x.M)$. Por hipótese de indução, podemos assumir que a propriedade é válida para M. $B \in \mathcal{B}$ e pelo caso anterior sabemos que B verifica a propriedade. Por hipótese M também verifica, logo,

$$\begin{aligned} var(B(\lambda x.M)) &= \{x\} \cup var(B) \cup var(M) \\ &= \{x\} \cup var(\mathbf{F}(B)) \cup var(\mathbf{F}(M)) \\ &= var(x := \mathbf{F}(B); \mathbf{F}(M)) \end{aligned}$$

$$\begin{aligned} assign(B(\lambda x.M)) &= \{x\} \cup assign(M) \cup assign(B) \\ &= \{x\} \cup assign(\mathbf{F}(M)) \cup assign(\mathbf{F}(B)) \\ &= assign(\mathbf{F}(B(\lambda x.M))) \end{aligned}$$

□

Lema 6.

1. $\forall E \in \mathcal{E}, var(E) = var(\boldsymbol{I}(E)) \wedge assign(E) = assign(\boldsymbol{I}(E))$
2. $\forall S \in \mathcal{S}, var(S) = var(\boldsymbol{I}(S)) \wedge assign(S) = assign(\boldsymbol{I}(S))$

Demonstração. Análoga à anterior. □

Posto isto, apresentaremos o passo seguinte na relação entre entre programas $\mathcal{LA}$ e $\mathcal{CPS}$, que consiste em demonstrar que para cada programa $\mathcal{CPS}$ é possível encontrar um representante da sua classe para a α-equivalência cuja conversão produz um programa na linguagem $\mathcal{SA}$.

O diagrama da Figura 9 sintetiza o nosso objectivo a demonstrar.

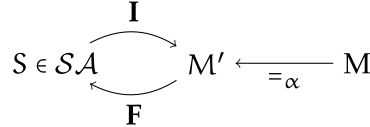

Figura 9: Preservação da relação sintática c/ α-equivalência

Passemos agora a enunciar e demonstrar o resultado sobre a α-equivalência.

Proposição 6.

$$\forall M \in \mathcal{M}\ \forall \vec{y}\ \exists M' \in \mathcal{M} : M' =_{\alpha} M \wedge\ (FV(M) \cup \vec{y}) \cap BV(M') = \emptyset \\ \Longrightarrow \boldsymbol{F}(M') \in \mathcal{SA}$$

Demonstração. Indução em $M \in \mathcal{M} \subseteq \mathcal{LA}$

Caso $M = (\lambda k.kB)$**.** Imediato. Para todo B, $\mathbf{F}((\lambda k.kB)) = ret(\mathbf{F}(B))$ que é um elemento de $\mathcal{SA}$.

Caso $M = B(\lambda x.M_0)$**.**

Sejam $\vec{z} = \vec{y} \cup FV(B) \cup \{x'\}$, com x' uma variável fresca, ou seja, $x' \notin FV(M) \cup \vec{y} \cup BV(M')$ e $M' = B'(\lambda x'.M'_0)$, tal que,

- $M =_\alpha M'$; e
- $(FV(M) \cup \vec{y}) \cap BV(M') = \emptyset$

Por hipótese de indução sabemos que

$$\exists M'_0 : M'_0 =_\alpha M_0 \wedge (FV(M_0) \cup \vec{z}) \cap BV(M'_0) = \emptyset \implies \mathbf{F}(M'_0) \in \mathcal{SA}$$

Por definição de $\mathbf{F}$ temos,
$\mathbf{F}(B(\lambda x'.M'_0)) = x' := \mathbf{F}(B); \mathbf{F}(M'_0)$.
Pela regra atr basta mostrar que,
$SA(M'_0) \wedge x' \notin var(\mathbf{F}(B)) \wedge assign(\mathbf{F}(M'_0)) \cap (\{x'\} \cup var(\mathbf{F}(B))) = \emptyset$.

- $\mathbf{F}(M'_0) \in \mathcal{SA}$, verdade pela hipótese de indução.
- $x' \notin var(\mathbf{F}(B'))$, pois x' é uma variável fresca.
-

$$\begin{aligned}
& assign(\mathbf{F}(M'_0)) \cap (\{x'\} \cup var(\mathbf{F}(B'))) \\
=& [\text{Lema 5}] \\
& assign(M'_0) \cap (\{x'\} \cup var(B')) \\
=& [\text{Def. } assign \text{ e } var] \\
& BV(M'_0)\{k\} \cap (\{x'\} \cup (BV(B') \cup FV(B')) \setminus \{k\}) \\
=& [\text{Por hipótese}] \\
& \emptyset
\end{aligned}$$

Logo, $\mathbf{F}(B'(\lambda x'.M'_0)) \in \mathcal{SA}$. □

Esta proposição permite-nos então selecionar para cada programa $\mathcal{CPS}$ um outro, o qual designamos de representante de classe (existindo uma infinidade de escolhas para o representante de cada classe), que tem como propriedade o facto de, ao ser convertido para $\mathcal{LA}$, produz um programa $\mathcal{SA}$. Abusando da linguagem, então representantes de classe são nada mais que programas $\mathcal{CPS}$ que estão no formato SA.

Fica por demonstrar, pois pode ser obtido fácilmente da bijecção nas traduções, que conversão de qualquer elemento em $\mathcal{SA}$, produz um programa $\mathcal{CPS}$, e que consiste num representante de classe (relação está patente na Figura 9).

2.3.3 *Preservação da semântica pelas traduções*

Nesta secção iremos mostrar de que forma as traduções preservam as semânticas de ambas as linguagens.

Tendo por base a noção de equivalência de estados, é possível alternar entre linguagens conforme as nossas necessidades e, simultaneamente, realizar transições/reduções sendo que, no final da execução,

o valor do programa será o mesmo. Vamos começar por provar a preservação da semântica em cada um dos sentidos (de $\mathcal{LA}$ para $\mathcal{CPS}$ e vice-versa) que iremos começar por ilustrar recorrendo novamente à utilização de diagramas. Comecemos então pela preservação da relação de transição.

As figuras 10 e 11 apresentam os diagramas que ilustram o resultado de que a semântica em cada um dos sentidos (de *LA* para *CPS* e vice-versa) é preservada. Prova essa que surge com os teoremas 1 e 2, e com os corolários 1 e 2.

$$\begin{array}{ccc} (S,\sigma) & \xrightarrow{\rightsquigarrow^n} & (S',\sigma') \\ \Big\downarrow (\mathbf{I}, \mathsf{sc}) & & \Big\downarrow (\mathbf{I}, \mathsf{sc}) \\ (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) & \xrightarrow{\rightarrow^n_\beta} & (\mathbf{I}(S'), \mathsf{sc}(\sigma', X)) \end{array}$$

Figura 10: Diagrama comutativo da relação semântica $\mathcal{LA} \rightarrow \mathcal{CPS}$

onde $X = var(S)$.

Teorema 1 (Preservação das transições)**.** *Sejam* $S, S' \in \mathcal{S}$, $\sigma, \sigma' \in \Sigma$ *e* $X = var(S)$

$$(S,\sigma) \rightsquigarrow (S',\sigma') \implies (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) \rightarrow_\beta (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$$

Demonstração. **Caso:** S da forma $x := E; S_0$. Temos que $X = \{x\} \cup var(E) \cup var(S_0)$.

Assumimos que $(S,\sigma) \rightsquigarrow (S',\sigma')$.

Ora $(x := E; S_0, \sigma) \rightsquigarrow (S_0, \sigma[x \mapsto [\![E]\!]_\sigma])$.

Temos então que,

- $S' = S_0$;
- $\sigma' = \sigma[x \mapsto [\![E]\!]_\sigma]$;

Sabemos que $\mathbf{I}(S) = \mathbf{I}(E)(\lambda x.\mathbf{I}(S_0))$.

Logo,

$$(\mathbf{I}(E)(\lambda x.\mathbf{I}(S_0)), \mathsf{sc}(\sigma, X))$$
$$\to_\beta \text{ [Por definição]}$$
$$(\mathbf{I}(S_0), \mathsf{sc}(\sigma, X)[x \mapsto \langle\!\langle \mathbf{I}(E) \rangle\!\rangle_{\mathsf{sc}(\sigma,X)}])$$
$$= \text{[Lema 4]}$$
$$(\mathbf{I}(S_0), \mathsf{sc}(\sigma, X)[x \mapsto [\![E]\!]_\sigma])$$
$$= \text{[Lema 7]}$$
$$(\mathbf{I}(S_0), \mathsf{sc}(\sigma[x \mapsto [\![E]\!]_\sigma], X))$$
$$=$$
$$(\mathbf{I}(S'), \mathsf{sc}(\sigma', X)),$$

como queriamos demonstrar.

Caso: $S = \mathtt{ret}(E)$. Trivial pois é configuração final, ou seja, apenas reduz para ele próprio em zero passos (reflexividade). □

Corolário 1. *Sejam,* $S, S' \in \mathcal{S}, \sigma, \sigma' \in \Sigma$ *e* $X = \mathtt{var}(S)$

$$(S, \sigma) \leadsto^n (S', \sigma') \implies (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) \to_\beta^n (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$$

Demonstração. Por indução no comprimento n da sequência de reduções.

1. **n = 0**. Temos então que, $(S, \sigma) \leadsto^0 (S', \sigma')$, ou seja, $S = S'$ e $\sigma = \sigma'$. Assim, $\mathbf{I}(S) = \mathbf{I}(S')$ e consequentemente

$$(\mathbf{I}(S), \mathsf{sc}(\sigma, \mathbf{I}(S))) \to_\beta^0 (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$$

2. **n = k + 1**. Temos então que mostrar que,

$$(S, \sigma) \leadsto^{k+1} (S', \sigma') \implies (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) \to_\beta^{k+1} (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$$

Da hipótese segue que existem S'' e σ'', tal que,

i $(S, \sigma) \leadsto^k (S'', \sigma'')$; e

ii $(S'', \sigma'') \leadsto (S', \sigma')$

De i. e pela hipótese de indução segue que,

$$(\mathbf{I}(S), \mathsf{sc}(\sigma, X)) \to_\beta^k (\mathbf{I}(S''), \mathsf{sc}(\sigma''))$$

De ii. e do Teor. 1 segue que

$(S, \sigma) \leadsto^{k+1} (S', \sigma') \implies (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) \to_\beta^{k+1} (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$, como pretendido.

$$(\mathbf{I}(S''), \mathsf{sc}(\sigma'')) \to_\beta (\mathbf{I}(S'), \mathsf{sc}(\sigma', X))$$

□

Terminada a demonstração da preservação da relação de transição passamos à demonstração preservação da relação de redução. Começando então pela observação do diagrama ilustrativo.

$$\begin{array}{ccc} (M,\delta) & \xrightarrow{\rightarrow^{n}_{\beta}} & (M',\delta') \\ \Big\downarrow (\mathbf{F},\mathsf{cs}) & & \Big\downarrow (\mathbf{F},\mathsf{cs}) \\ (\mathbf{F}(M),\mathsf{cs}(\delta)) & \xrightarrow{\rightsquigarrow^{n}} & (\mathbf{F}(M'),\mathsf{cs}(\delta')) \end{array}$$

Figura 11: Diagrama comutativo da relação semântica $\mathcal{CPS} \to \mathcal{LA}$

Teorema 2 (Preservação da redução)**.** *Sejam* $M, M' \in \mathcal{M}$, $\delta, \delta' \in \Delta$, *fechos para* M *e* M'*, respectivamente.*

$$(M,\delta) \to_{\beta} (M',\delta') \implies (\boldsymbol{F}(M),\mathsf{cs}(\delta)) \rightsquigarrow (\boldsymbol{F}(M'),\mathsf{cs}(\delta'))$$

Demonstração. Vamos assumir que $(M,\delta) \to_{\beta} (M',\delta')$. Consideremos que $M = B(\lambda x.M_0)$.

Ora $(B(\lambda x.M_0),\delta) \to_{\beta} (M_0,\delta[x \mapsto \langle\!\langle B\rangle\!\rangle_{\delta}])$. Temos então que,

- $M' = M_0$;
- $\delta' = \delta[x \mapsto \langle\!\langle B\rangle\!\rangle_{\delta}]$;

Sabemos também que $\mathbf{F}(B(\lambda x.M_0)) = x := \mathbf{F}(B);\mathbf{F}(M_0)$. Logo,

$$\begin{aligned} (x := \mathbf{F}(B);\mathbf{F}(M_0),\mathsf{cs}(\delta)) &\rightsquigarrow (\mathbf{F}(M_0),\mathsf{cs}(\delta)[x \mapsto [\![\mathbf{F}(B)]\!]_{\mathsf{cs}(\delta)}]) \\ &= [\text{Lema 3}] \\ &(\mathbf{F}(M_0),\mathsf{cs}(\delta)[x \mapsto \langle\!\langle B\rangle\!\rangle_{\delta}]) \\ &= \\ &(\mathbf{F}(M_0),\mathsf{cs}(\delta[x \mapsto \langle\!\langle B\rangle\!\rangle_{\delta}])) \\ &= \\ &(\mathbf{F}(M'),\mathsf{cs}(\delta')) \end{aligned}$$

Como queríamos demonstrar. □

No Lema 7 encontramos um resultado necessário para a obtenção da preservação semântica para a tradução **I**.

Lema 7. *Para todo* $v \in \mathrm{Const}$, $X \in \wp(\mathrm{Vars})$, $x \in X$, $\sigma \in \Sigma$, $\mathsf{sc}(\sigma,X)[x \mapsto v] = \mathsf{sc}(\sigma[x \mapsto v],X)$.

Demonstração. Tomemos

- $\delta = \mathsf{sc}(\sigma, X)[x \mapsto v]$;
- $\delta' = \mathsf{sc}(\sigma[x \mapsto v], X)$.

Queremos mostrar que $\delta = \delta'$. Temos que, para $y \in X$, δ é tal que

$$\delta(y) = \begin{cases} v & \text{se } y = x \\ \sigma(y) & \text{se } y \neq x \end{cases}$$

e δ' é tal que

$$\delta'(y) = \delta[x \mapsto v] = \begin{cases} v & \text{se } y = x \\ \sigma(y) & \text{se } y \neq x \end{cases}$$

Como $\mathrm{dom}(\delta) = \mathrm{dom}(\delta') = X$, então $\delta = \delta'$. □

Corolário 2. *Sejam* M, M' *e* $\delta, \delta' \in \Delta$ *fechos para* M *e* M'*, respectivamente.*

$$(M, \delta) \to_{\beta}^{n} (M', \delta') \implies (\boldsymbol{F}(M), \mathsf{cs}(\delta)) \rightsquigarrow^{n} (\boldsymbol{F}(M'), \mathsf{cs}(\delta'))$$

Demonstração. Análoga à demontração do Teor. 1. □

Em suma, e conforme suprarreferido, é possível, então, alternar entre linguagens preservando o significado do programa. Na Figura 12, onde são compostos os diagramas das figuras 10 e 11, observamos o fenómeno de alternância entre linguagens preservando o "significado" do programa.

$$\begin{array}{ccc} (S, \sigma) & \xrightarrow{\rightsquigarrow^{n}} & (S', \sigma') \\ \downarrow{\scriptstyle (\mathbf{I}, \mathsf{sc})} & & \downarrow{\scriptstyle (\mathbf{I}, \mathsf{sc})} \\ (\mathbf{I}(S), \mathsf{sc}(\sigma, X)) & \xrightarrow{\to_{\beta}^{n}} & (\mathbf{I}(S'), \mathsf{sc}(\sigma', X)) \\ \downarrow{\scriptstyle (\mathbf{F}, \mathsf{cs})} & & \downarrow{\scriptstyle (\mathbf{F}, \mathsf{cs})} \\ (S, \sigma'') & \xrightarrow{\rightsquigarrow^{n}} & (S', \sigma''') \end{array}$$

Figura 12: Diagrama comutativo da relação semântica $\mathcal{LA} \Leftrightarrow \mathcal{CPS}$

Na Figura 12, σ é equivalente a σ'', e σ' a σ'''. Este diagrama é também ilustrativo do resultado final (a bijeção semântica entre os formatos), com o qual iremos concluir este capítulo, sobre a a preservação da semântica nas traduções (Teor. 3).

Os lemas 8, 9 e 10, apresentam resultados auxiliares ao resultado final.

No Lema 8, em particular, é possível observar que a conversão de uma configuração final em $\mathcal{LA}$ origina também uma configuração final em $\mathcal{CPS}$ e vice-versa.

Lema 8.

1. *Seja* (S, σ) *uma configuração final em* $\mathcal{LA}$ *e* X *tal que* $var(S) \subseteq X \subseteq Vars$, $(\boldsymbol{I}(S), \mathsf{sc}(\sigma, X))$ *também é configuração final em* $\mathcal{CPS}$.

2. *Seja* (M, δ) *uma configuração final em* $\mathcal{CPS}$, *então* $(\boldsymbol{F}(M), \mathsf{cs}(\delta))$ *é configuração final em* $\mathcal{LA}$.

Demonstração. Imediata das definições de **F** e **I**. □

O Lema 9 mostra-nos a preservação de fechos pelas traduções.

Lema 9. *Seja* $M \in \mathcal{M}$ *e* $\delta \in \Delta$ *fecho para* M.

$$\mathsf{sc}(\mathsf{cs}(\delta), var(M)) = \delta$$

Demonstração.

$$\begin{aligned} & \mathsf{sc}(\mathsf{cs}(\delta), var(M)) = \delta \\ \Longleftrightarrow & [\text{Def. } 29] \\ & \mathsf{sc}(\delta_c, var(M)) = \delta \\ \Longleftrightarrow & [\text{Def. } 30] \\ & \delta_c|_{var(M)} = \delta \end{aligned}$$

□

O Lema 10 é o lema análogo ao Lema 9, no entanto neste caso não podemos afirmar que preserva o estado, apesar disto, o estado que é obtido é equivalente ao inicial. Este enfraquecimento do resultado, em relação ao anterior, deve-se ao facto de que o contexto de um fecho é menor do que o respectivo estado (pela tradução) e então, existe informação que è descartada, contudo aquilo que mostramos (com a equivalência de estados) é que essa mesma informação é descartável.

Lema 10. *Seja* S *um programa* $\mathcal{LA}$ *e* σ *um estado.* σ *e* $\mathsf{cs}(\mathsf{sc}(\sigma, var(S)))$ *são equivalentes relativamente a* S.

Demonstração. $\sigma' = \mathsf{cs}(\mathsf{sc}(\sigma, var(S)))$. σ e σ' são equivalentes em S se $\forall x \in var(S),\ \sigma(x) = \sigma'(x)$.

Ora,

$$
\begin{aligned}
& \mathsf{cs}(\mathsf{sc}(\sigma, var(S))) \\
=& [\text{Def. 30}] \\
& \mathsf{cs}(\sigma|_{var(S)}) \\
=& [\text{Def. 29}] \\
& \sigma|_{var(S)_\mathbf{c}} \\
=& [\text{Por definição}] \\
& \sigma|_{var(S)_\mathbf{c}} \begin{cases} \sigma|_{var(S)_\mathbf{c}}(x) = \sigma(x) & \text{, se } x \in var(S) \\ \sigma|_{var(S)_\mathbf{c}}(x) = \mathbf{c} & \text{, se } x \notin var(S) \end{cases}
\end{aligned}
$$

Então, σ e σ' são equivalentes relativamente a S. □

Teorema 3 (Preservação da semântica pelas traduções)**.** *Sejam* $S \in \mathcal{S}$, $M \in \mathcal{M}$, $\delta \in \Delta$, $\sigma \in \Sigma$ *e* $v \in \mathsf{Const}$.

1. (S, σ) *avalia em* $v \iff (\boldsymbol{I}(S), \mathsf{sc}(\sigma, var(S)))$ *avalia em* v.
2. (M, δ) *avalia em* $v \iff (\boldsymbol{F}(M), \mathsf{cs}(\delta))$ *avalia em* v.

Demonstração. Iremos demonstrar cada uma das implicações envolvidas no teorema. Os pontos a) e c) dizem respeito ao ponto 1 do teorema, enquanto que os pontos b) e d) dizem respeito ao ponto 2 do teorema.

(a) Suponhamos que (S, σ) avalia em v, ou seja,

$(S, \sigma) \rightsquigarrow^* (ret(E), \sigma') \wedge [\![E]\!]_{\sigma'} = v$.

Queremos mostrar que $(\mathbf{I}(S), \mathsf{sc}(\sigma, var(S)))$ avalia em v.

Pelo Corol. 1 $(\mathbf{I}(S), \mathsf{sc}(\sigma, var(S))) \to^*_\beta (\mathbf{I}(ret(E)), \mathsf{sc}(\sigma', var(S)))$.

Como $(ret(E), \sigma')$ é configuração final, pelo Lema 8,

$(\mathbf{I}(S'), \mathsf{sc}(\sigma', var(S)))$ também é. Portanto, $(\mathbf{I}(S), \mathsf{sc}(\sigma', var(S)))$ avalia para $\langle\!\langle \mathbf{I}(E) \rangle\!\rangle_{\mathsf{sc}(\sigma', var(S))}$ que pelo Lema 4 é igual a $[\![E]\!]_{\sigma'} = v$.

(b) Suponhamos que (M, δ) avalia em v, ou seja,

$(M, \delta) \to^*_\beta (\lambda k.kB, \delta') \wedge \langle\!\langle B \rangle\!\rangle_{\delta'} = v$.

Queremos mostrar que $(\mathbf{F}(M), \mathsf{cs}(\delta))$ avalia em v.

Pela Corol. 2 $(\mathbf{F}(M), \mathsf{cs}(\delta)) \rightsquigarrow^* (\mathbf{F}(\lambda k.kB), \mathsf{cs}(\delta'))$.

Como $(\lambda k.kB, \delta')$ é configuração final, pelo Lema 8,

$(\mathbf{F}(\lambda k.kb), \mathsf{cs}(\delta'))$ também é.

Portanto, $(\mathbf{F}(M), \mathsf{cs}(\delta'))$ avalia para

$[\![\mathbf{F}(B)]\!]_{\mathsf{cs}(\delta')}$ que pelo Lema 3 é igual a $\langle\!\langle B \rangle\!\rangle_{\delta'} = v$.

(c) Provemos que se $(\mathbf{I}(S), \mathsf{sc}(\sigma, var(S)))$ avalia em v então (S, σ) avalia em v

$$
\begin{aligned}
&(\mathbf{I}(S), \mathsf{sc}(\sigma, var(S))) \text{ avalia em } v\\
&\implies [\text{Ponto b)}]\\
&(\mathbf{F}(\mathbf{I}(S)), \mathsf{cs}(\mathsf{sc}(\sigma, var(S)))) \text{ avalia em } v\\
&\implies [\text{Lema 10}, \text{Def. 12 e Prop. 4}]\\
&(S, \sigma') \text{ avalia em } v \text{ , com } \sigma' = \mathsf{cs}(\mathsf{sc}(\sigma, var(S)))\\
&\iff [\sigma \text{ e } \sigma' \text{ equivalentes em } S]\\
&(S, \sigma) \text{ avalia em } v
\end{aligned}
$$

(d) Provemos que se $(\mathbf{F}(M), \mathsf{cs}(\delta))$ avalia em v então (M, δ) avalia em v

$$
\begin{aligned}
&(\mathbf{F}(M), \mathsf{cs}(\delta)) \text{ avalia em } v\\
&\implies [\text{Pelo ponto a)}]\\
&(\mathbf{I}(\mathbf{F}(M)), \mathsf{sc}(\mathsf{cs}(\delta), var(\mathbf{F}(M)))) \text{ avalia em } v\\
&\iff [\text{Lema 9}, \text{Prop. 5 e Lema 5}]\\
&(M, \delta) \text{ avalia em } v
\end{aligned}
$$

$\square$

3 EXTENSÕES

Este capítulo será dedica-se à realização de expansões para nossas linguagens intermédias. Até agora temos apenas a capacidade de atribuir valores a variáveis e retornar valores desses procedimentos. Neste capítulo iremos começar por introduzir comandos para realizar operações condicionais, usualmente conhecidos como "if-else". Posteriormente, acrescentaremos procedimentos às nossas linguagens intermédias.

Note-se que iremos reutilizar letras para meta-variáveis tal como nos capítulos anteriores, assim como nomes de funções. Assim, alguns nomes de funções serão idênticos para ambas as linguagens intermédias, dado que conceptualmente se tratam da mesma função (mas aplicada a variantes sintáticas). Em regra, será possível inferir os domínios e contradomínios dessas funções a partir do contexto.

3.1 CONDICIONAIS

3.1.1 *Construção* If

O comando composto "if-else" faz depender o fluxo de execução de um programa da avaliação de uma expressão lógica (geralmente designada de "guarda". Se o resultado da avaliação da guarda for "verdadeiro", então a execução do comando corresponde à execução do sub-comando que está no ramo "if"; se o resultado da avaliação da guarda for "falso", então a execução do comando corresponde à execução do sub-comando que está no ramo "else".

Nas nossas linguagens, apresentadas no capítulo anterior, começamos por definir expressões triviais, bem como um conjunto abstrato que contém os valores que as nossas expressões podem representar. Portanto, tal como fizemos ao definir a constante especial **c**, urge definir duas novas constantes especiais para representar o resultado "verdadeiro" e o "falso". Iremos então usar

- **v** para verdadeiro; e
- **f** para falso.

Seguidamente iremos apresentar a sintaxe para os construtores "if-else", para cada uma das linguagens.

3.1.1.1 $\mathcal{LA}$

Olhemos novamente para a gramática da nossa linguagem $\mathcal{LA}$.

$$
\begin{aligned}
S &::= x := E; S \\
&\quad \mid \mathtt{ret}(E) \\
E &::= x \\
&\quad \mid c
\end{aligned}
$$

Para a introdução de um novo comando é necessário expandir esta gramática. Geralmente, numa linguagem imperativa temos construções *If* como na Figura 13,

```
if (<Expressão Booleana>) {
        ....
} else {
        ....
}
```

Figura 13: Ex. comando *If* usual

onde o primeiro ramo representa os comandos que são executados caso a expressão avalie em "verdadeiro" e o segundo ramo (*else*) os comandos executados caso avalie em "falso".

Outro ponto que merece referência, anida antes da introdução da novo gramática, prende-se com o facto de que consideramos (assim como em [13]) que não é possível compor, tal como nos comandos de atribuição usando o conector ";", um comando *If* com outros comandos. Este é um formato para o qual qualquer programa que permita sição com *If* pode ser convertido. Uma das formas mais simples passa por transpor o comando a seguinte ao conector ";" para cada um dos ramos "if". Por exemplo, o programa da Figura 14 é convertido para um formato sem composição de comandos *If* originando o programa da Figura 15. De salientar que, neste caso, como o ramo "else" do primeiro "if" termina com um return o segundo "if" já não é transposto para esse ramo. Este método aplicado evita duplicação de código desnecessária.

```
 if (<Expressão Booleana>) {
        x: = b;
        y: = d;
 } else {
        ret(a)
 };
 if (<Expressão Booleana>) {
        ret(x)
} else {
        ret(y)
}
```

Figura 14: Ex. programa com composição de comandos *If*

```
if (<Expressão Booleana>) {
        x: = b;
        y: = d;
        if (<Expressão Booleana>) {
                ret(x)
        } else {
                ret(y)
        }
} else {
        ret(a)
}
```

Figura 15: Ex. programa sem composição de comandos *If*

Podemos também observar a particularidade deste formato olhando para a árvore de comandos/sintaxe na Figura 16.

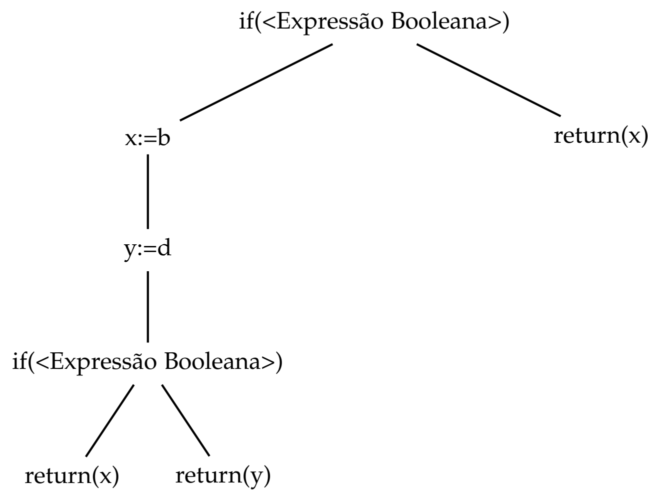

Figura 16: Árvore de comandos do exemplo da Figura 13

Deste modo, a nossa nova gramática para $\mathcal{LA}$ será a seguinte:

$$
\begin{aligned}
S &::= x := E; S \\
&\quad \mid \mathtt{if}(E)\{\ S\ \}\ else\ \{\ S\ \} \\
&\quad \mid \mathtt{ret}(E) \\
E &::= x \\
&\quad \mid c
\end{aligned}
$$

3.1.1.2 *CPS*

Na forma mais pura de λ-calculus podemos usar a codificação de Church para criar "if statements". A adoção deste tipo de codificação traria um nível de complexidade mais elevado ao nosso trabalho sem acrescentar proveitos significativos face aos nossos objetivos. Neste

sentido, iremos adicionar à nossa linguagem $\mathcal{CPS}$ uma construção "if-else" com a finalidade de, tal como apresentamos para a linguagem $\mathcal{LA}$, simular o comportamento de um *If*.

Quando falamos de linguagens funcionais, e, em particular, naquelas mais próximas do λ-calculus, podemos encarar um "if-else" como uma função que recebe 3 argumentos - avalia o primeiro e devolve o segundo ou terceiro consoante o resultado dessa avaliação. Note-se que esta observação serve também como prelúdio para as ideias subjacentes À semântica que que apresentaremos na próxima secção.

A nossa sintaxe para o "if-else" será a seguinte:

```
(if B M1 M2)
```

Com `B` uma expressão base e `M1` e `M2` expressões $\mathcal{CPS}$. Assim, a gramática $\mathcal{CPS}$ é redefinida para incluir esta nova construção, da seguinte forma:

$$
\begin{aligned}
M &::= B(\lambda x.M) \\
&\mid (\mathrm{if}\ B\ M\ M) \\
&\mid (\lambda k.kB) \\
B &::= \lambda k.kx \\
&\mid \lambda k.kc
\end{aligned}
$$

3.1.2 *Semântica*

Nesta breve secção seguem-se as novas regras para as relações $\rightsquigarrow$ e $\rightarrow_\beta$ de forma a dar semântica aos construtores *if* introduzidos anteriormente.

Não obstante, de salientar a existência de duas formas de definir a avaliação da expressão "booleana": (1) podemos definir duas constantes para representar o valor de "verdade" e "falsidade" ou (2) podemos optar por definir o valor para "falsidade" e todos os restantes são aceites como valores de "verdade".

No nosso estudo, iremos assumir a existência duas constantes distintas, uma para cada caso, nomeadamente **v** e **f**.

3.1.2.1 $\mathcal{LA}$

$$
\mathrm{if}_{\mathbf{v}}\ \frac{[\![E]\!]_\sigma = \mathbf{v}}{(\mathrm{if}(E)\{S_0\}\mathrm{else}\{S_1\}, \sigma) \rightsquigarrow (S_0, \sigma)}
$$

$$
\mathrm{if}_{\mathbf{f}}\ \frac{[\![E]\!]_\sigma = \mathbf{f}}{(\mathrm{if}(E)\{S_0\}\mathrm{else}\{S_1\}, \sigma) \rightsquigarrow (S_1, \sigma)}
$$

3.1.2.2 $\mathcal{CPS}$

$$\text{if}_{\mathbf{v}}\frac{\langle\!\langle B \rangle\!\rangle_{\delta} = \mathbf{v}}{((\text{if } B\ M_0\ M_1), \delta) \to_{\beta} (M_0, \delta)}$$

$$\text{if}_{\mathbf{f}}\frac{\langle\!\langle B \rangle\!\rangle_{\delta} = \mathbf{f}}{((\text{if } B\ M_0\ M_1), \delta) \to_{\beta} (M_1, \delta)}$$

3.2 PROCEDIMENTOS

Nesta secção iremos apresentar mecanismos para adicionar procedimentos às nossas linguagens. De modo a usufruir de todo o estudo apresentado no Capítulo 2, iremos tratar procedimentos como “macros”, que são alvo de pré-processamento. Assim, sempre que um procedimento for invocado, iremos expandir o código da função que o invoca com o código do procedimento que está a ser invocado.

A “remoção” de procedimentos ocorre com a expansão e com a selecção de um procedimento em específico, habitualmente designado de “main”, que consistira no ponto de entrada da avaliação do programa.

Então, um programa será um conjunto de procedimnetos.

3.2.1 *Procedimentos em $\mathcal{LA}$*

Um procedimento é um nome ao qual está associado um vetor de parâmetros formais e um comando, chamado o corpo do procedimento. No que respeita ao vetor de parâmetros associados a um procedimento, iremos tomar as seguintes considerações:

Sejam $x_1, x_2, \ldots, x_n$ os parâmetros de entrada de um procedimento. Iremos abreviar esta sequência de parâmetros usando a notação $\vec{x}$ (vetor x).

Iremos também, por abuso de notação, aceitar que a notação $\vec{x}$ possa ser vista como um conjunto. Assumimos ainda que não existe repetição de nomes nos parâmetros formais de um procedimento.

Neste sentido, iremos recorrer à seguinte sintaxe para procedimentos:

$$l := \text{proc}(\vec{x})\{S\}$$

onde l é o *nome* do procedimento (a que também chamamos “label”), $\vec{x}$ é o *vetor de parâmetros* do procedimento e S o *corpo* do procedimento, assumindo que dentro de um programa não existem procedimentos com nomes iguais.

A Figura 6 exemplifica um programa ao qual atribuimos a “label” `f`.

```
f := proc(a,b){
        if(E){
                x:= a;
                ret(x)
        } else {
                x:=b;
                ret(x)
        }
        }
```

Figura 17: Ex. procedimento na linguagem $\mathcal{LA}$

A invocação do procedimento `f` pode então ser feita da seguinte forma:

```
f(w,z)
```

onde w, z são os *parâmetros de invocação* a utilizar. Esta invocação de `f` dará origem à expansão do corpo do procedimento da Figura 17 com duas atribuições adicionais (às variáveis a e b), como ilustrado na Figura 18.

```
a:= w;
b:= z;

if(E){
          x:= a;
          ret(x)
        } else {
          x:=b;
          ret(x)
        }
```

Figura 18: Expansão do procedimento da Figura 17 dados parâmetros de invocação w e z

Tendo como objetivo sermos capazes de, em qualquer ponto do programa, e em particualar dentro dos procedimentos, invocar um procedimento com parâmetros concretos e associar o resultado dessa invocação a uma variável.

Para este efeito, recorremos à seguinte sintaxe: `x:=l(a1,a2,....,an)` onde `l` é o nome do procedimento, `a1,a2,...,an` os parâmetros concretos da invocação e `x` a variável à qual é atribuído o resultado da invocação. A Figura 19 apresenta um exemplo de um programa que invoca um procedimento auxiliar (onde `f` é o procedimento da Figura 17).

```
g := proc(c,d){
      y := f(c,d);
      ret(y)
}
```

Figura 19: Ex. de procedimento com invocação de outro

De modo a realizar a expansão expansão explicitada anteriormente (Figura 18), é necessário ter em conta que y terá de reter o valor retornado pelo comando `ret(x)` da Figura 18.

Para uma melhor compreensão do processo apresentamos resumidamente a ideia subjacente ao mecanismo de expansão a utilizar sempre que um procedimento é invocado. Consideremos, para tal, o seguinte código de programa:

```
f := proc(x1,....,xN){
 ...
 if(E) {
        ....
        ret(E2);
 } else {
         ...
         ret(E3);
 }
}
```

Então, atribuir o resultado de uma invoção de `f`, digamos `y:=f(a,...,b)` num contexto da forma

`...;S1;y:=f(a1,...,aN);S2;...`

produz o seguinte código:

```
...;
S1;
x1 := a1;
...
xN := aN;
 if(E) {
        ....
        y:=E2;
        S2;
        ...
 } else {
         ...
         y:=E3;
         S2;
         ...
 }
```

Retomemos agora o exemplo da Figura 19. A expansão do procedimento `g`, usando para `f` a definição da Figura 17, produz o seguinte procedimento:

```
g := proc(c,d){
a:= c;
b:= d;
if(E) {
        x:=a;
        y:=x;
        ret(y);
    } else {
        x:=b;
        y:=x;
        ret(y);
    }
}
```

Figura 20: Ex. Procedimento da Figura 19 com invocação de função pré-processada

Antecedendo a nova gramática é ainda necessário introduzir algumas notações relativamente ao conceito de lista. Dado um conjunto A, representaremos por $[A]$ o conjunto das listas de elementos de A. $[]$ denota a lista vazia e dado um elemento $e \in A$ e uma lista $L \in [A]$, $e : L$ representa a lista de cabeça e e cauda L. Por conveniência de notação, por vezes usamos a notação $[a_1, \ldots, a_n]$ em vez de $a_1 : \ldots : a_n : []$.

Uma vez assimilada a notação, apresentamos de seguida a nova gramática, que estende a apresentada nas secções anteriores:

$$
\begin{aligned}
P &::= [] \mid (l := proc(\vec{x})\{S\}) : P \\
S &::= x := E; S \\
&\quad \mid \ x := l(\vec{E}); S \\
&\quad \mid \ if(E)\{\ S\ \}\ else\ \{\ S\ \} \\
&\quad \mid \ ret(E) \\
E &::= x \\
&\quad \mid \ c
\end{aligned}
$$

onde l representa um nome de procedimento, e $\vec{x}$ e $\vec{E}$ vetores de variáveis e de expressões, respetivamente. Os elementos da classe P serão então os nossos programas (listas de procedimentos) e denotaremos por $\mathcal{P}$ o conjunto de todos os programas. Quanto às restantes classes, usaremos as mesmas convenções de nomes enunciadas no capítulo anterior.

Uma representação alternativa seria considerar um programa como um conjunto de procedimentos e fixar um procedimento como ponto de entrada (“main”) do programa.

Não obstante, este processo contém alguns fenómenos limitantes/-relevantes, nomeadamene:

- Colisão de nomes de variáveis, isto é, podemos ter parâmetros formais de diferentes procedimentos que têm o mesmo nome.
- Impossibilidade de recursão.

Relativamente segundo ponto, a impossibilidade de recursão, deve-se ao facto de querermos evitar ciclos ao nível da substituição de invocação de procedimentos por parte do preprocessador. No que concerne primeiro vamos assumir que é possível renomear os parâmetros formais do procedimento a ser invocado, de forma a evitar colisões de nomes.

Uma vez que, com a alteração feita, os programas $\mathcal{LA}$ permitem agora procedimentos, é necessário redefinir o conceito de programa no formato $\mathcal{SA}$ para esta nova gramática, o que pode acontecer de uma das seguintes formas,

- Impor que num programa cada variável seja atribuída apenas uma vez e apenas num único procedimento.
- Impor apenas a restrição de que, após a atribuição de uma variável, esta não seja atribuída novamente na seu grafo de invocações.

Com a nova gramática definida, estamos aptos a formalizar o conceito de expansão / pré-processamento, o qual também designaremos de *compilação* daqui em diante.

Existe, no entanto, antes de formalizar a noção de compilação, a necessidade de apresentar alguns conceitos auxiliares necessários à definição de *boa formação* de um programa. Iremos, informalmente, apresentar um método que recorre à utilização de grafo de invocações de um programa P ($DI(P)$) e, posteriormente, apresentar uma definição alternativa, formal, que será utilizada para a análise.

Comecemos por definir a função $LB(P)$, que representa a coleção de "labels" ou "nomes" de procedimentos de um programa P que, ser-nos-á útil tanto na definição de grafo de invocações como na noção alternativa.

Seja $\mathcal{R}$ o conjunto de "nomes" de procedimentos.

Definição 31. *Seja* $LB : \mathcal{P} \to [\mathcal{R}]$ *a função definida da seguinte forma*

$$\begin{aligned} LB([]) &= [] \\ LB((l := proc(\vec{x})\{S\}) : P') &= l : LB(P') \end{aligned}$$

Um grafo direcionado é composto por um conjunto de vértices e por um conjunto de arestas. A notação (x, y) usar-se-á para indicar que existe uma aresta do vértice x para o vértice y. Diz-se um grafo

direcionado pois uma aresta (x, y) significa que "podemos ir" de x para y mas não o inverso.

Seja P um programa. $DI(P)$ é o grafo direcionado de invocações de P, ou seja,

$$(g, f) \in DI(P) \text{ se } g \text{ contém alguma invocação de } f,$$

onde os vértices g e f são elementos de $LB(P)$.

Dizemos que P é *bem formado* se o grafo $DI(P)$ não contém ciclos.

As duas definições anteriores permitem-nos recorrer a alguns conceitos da teoria de grafos para raciocinar acerca de programas. Apesar disso, e de forma a facilitar a demonstração dos resultados que se seguem, optámos por recorrer a uma definição alternativa do conceito de programa bem formado. Na definição Def. 33 definimos as funções que nos permitem avaliar a boa formação de um programa.

Antes deste conceito temos de apresentar uma outra função que iremos utilizar mais adiante. Esta função constrói um conjunto com todas as "labels" invocadas em algum dos procedimentos de um dado programa.

Definição 32. *Seja* $\mathsf{inv} : \mathcal{P} \to \wp(\mathcal{R})$ *a função definida do seguinte modo*

$$\begin{aligned} \mathsf{inv}([\,]) &= \{\} \\ \mathsf{inv}((l := \mathtt{proc}(\vec{x})\{S\}) : P') &= \mathsf{inv}(S) \cup \mathsf{inv}(P') \end{aligned}$$

onde $\mathsf{inv} : \mathcal{S} \to \wp(\mathcal{R})$ *é a função definida do seguinte modo*

$$\begin{aligned} \mathsf{inv}(\mathtt{ret}(E)) &= \{\} \\ \mathsf{inv}(x := E; S) &= \mathsf{inv}(S) \\ \mathsf{inv}(x := l(\vec{E}); S) &= \{l\} \cup \mathsf{inv}(S) \\ \mathsf{inv}(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}) &= \mathsf{inv}(S_0) \cup \mathsf{inv}(S_1) \end{aligned}$$

Definição 33. P *diz-se* bem formado *quando*

$$\forall l := \mathtt{proc}(\vec{x})\{S\} \in P, \;\; \mathsf{wf}(\{l\}, P, S) = \textbf{\textit{True}}$$

onde

$\mathsf{wf} : \wp(\mathcal{R}) \times \mathcal{P} \times \mathcal{S} \to \mathrm{Bool}$ *é definida da seguinte forma:*

$$\mathsf{wf}(L, P, \mathtt{ret}(E)) = \textbf{\textit{True}}$$

$$\mathsf{wf}(L, P, x := E; S) = \mathsf{wf}(L, P, S)$$

$$\mathsf{wf}(L, P, x := l(\vec{E}); S) = \begin{cases} \textbf{\textit{False}}, \; \textit{se } l \in L \\ \textbf{\textit{False}}, \; \textit{se } l \notin LB(P) \\ (\mathsf{wf}(l : L, P, S') \wedge \mathsf{wf}(L, P, S)) \\ \qquad \textit{se } l := \mathtt{proc}(\vec{x})\{S'\} \in P \textit{ e } l \notin L \end{cases}$$

$$\mathsf{wf}(L, P, \mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}) = \mathsf{wf}(L, P, S_0) \wedge \mathsf{wf}(L, P, S_1)$$

Informalmente, a definição anterior garante-nos que um programa bem formado não tem recursividade (direta ou indireta), i. e., que não existem invocações de funções não definidas nem procedimentos com o mesmo nome.

Um outro que carece de explicação relativamente à função wf, prende-se com a medida que fundamenta o esquema de recursão que está a ser usado, e que garante que as invocações recursivas eventualmente teminam. Dados um conjunto de "labels" L, um programa P e um comando S, a medida corresponde a uma ordem lexicográfica sobre o par $(\#(LB(P) \setminus L), S)$. A ordem para as primeiras componentes deste par é a ordem em $\mathbb{N}_0$ e a ordem para as segundas componentes do par é a ordem associada à estrutura de comandos. A utilização destes pares deve-se especialmente ao caso $\mathsf{wf}(L, P, x := l(\vec{E}); S)$ com $l := \mathtt{proc}(\vec{x})\{S'\} \in P$ e $l \notin L$, onde o objecto que sofre uma redução é o primeiro elemento do par (aumentando L com uma "label" de P reduzimos o tamanho do conjunto $LB(P) \setminus L$). Em todos os outros casos observa-se uma redução do segundo elemento do par. Esta medida servir-nos-á para demonstrar propriedades sobre o predicado wf. Uma outra propriedade sobre função wf está expressa no Lema 11.

Lema 11.

$$\mathsf{wf}(\{l\}, P, S) \implies l \notin \mathsf{inv}(S)$$

Demonstração. Por indução em S. □

A boa formação de um programa implica que não exista recursão (direta ou indireta) e que o programa seja finito. Este facto é de extrema importância para a definição do conceito de *programa compilado*. Neste sentido, comecemos por construir o nosso método de expansão de procedimentos. Este método conta com 3 funções de substituição, todas elas designadas por subs (Def. 34, Def. 35 e Def. 36), retirando-se o contexto onde são aplicadas pelo tipo dos parâmetros.

Definição 34. *Seja* $\mathsf{subs} : \mathcal{S} \times \mathit{Vars} \times \mathcal{S} \to \mathcal{S}$ *a função definida da seguinte forma:*

$$\mathsf{subs}(\mathtt{ret}(E), x, S) = x := E; S$$
$$\mathsf{subs}(y := E; S', x, S) = y := E; \mathsf{subs}(S', x, S)$$
$$\mathsf{subs}(y := l(\vec{E}); S', x, S) = y := l(\vec{E}); \mathsf{subs}(S', x, S)$$
$$\mathsf{subs}(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}, x, S) =$$
$$\quad \mathtt{if}(E)\{\mathsf{subs}(S_0, x, S)\}\mathtt{else}\{\mathsf{subs}(S_1, x, S)\}$$

Definida por recursão estrutural no primeiro elemento, esta função, $\mathsf{subs}(S, x, S')$, permite-nos substituir todos os comandos de "retorno" de S pela sequência de comandos S', o que em termos de execução significa que, cada vez que em S é encontrado um comando de "retorno" a execução continuará com S', sendo o valor de "retorno" associado à variável x.

Definição 35. *Seja* subs : ***Proc*** $\times \mathcal{S} \to \mathcal{S}$*, a função definida da seguinte forma:*

$$\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, \mathtt{ret}(E)) = \mathtt{ret}(E)$$
$$\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, x := E; S') = x := E; \mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, S')$$
$$\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, x := g(\vec{E}); S') =$$
$$\begin{cases} z_1 := E_1; \ldots; z_n := E_n; \mathsf{subs}(S, x, \mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, S')) \\ \qquad \textit{caso } g = l \\ x := g(\vec{E}); \mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, S') \\ \qquad \textit{caso } g \neq l \end{cases}$$
$$\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, \mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}) =$$
$$\mathtt{if}(E)\{\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, S_0)\}\mathtt{else}\{\mathsf{subs}(l := \mathtt{proc}(\vec{z})\{S\}, S_1)\}$$

Note que ***Proc*** *denota o conjunto de todos os procedimentos.*

Definida por recursão estrutural no último argumento, esta função substitui todas as invocações de um procedimento específico (numa sequência de comandos) por uma sequência de comandos, recorrendo à função de substituição anterior.

Definição 36. *Seja* subs : ***Proc*** $\times \mathcal{P} \to \mathcal{P}$ *a função definida da seguinte forma:*

$$\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, []) = []$$
$$\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, (g := \mathtt{proc}(\vec{y})\{S'\}) : P') =$$
$$(g := \mathtt{proc}(\vec{y})\{\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S')\}) : \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P')$$

Esta última função de substituição efetua, recorrendo às funções de substituição anteriores, a substituição das invocações de um procedimento específico em todos os procedimentos de um programa.

Posto isto, estamos agora em condições de definir a função de compilação para programas $\mathcal{LA}$:

Definição 37. *Seja* cp : $\mathcal{P} \to \mathcal{P}$ *a função definida da seguinte forma:*

$$\mathsf{cp}(P) = \mathsf{cpL}(LB(P), P)$$

$$\mathsf{cpL} : [\mathcal{R}] \times \mathcal{P} \to \mathcal{P}$$
$$\mathsf{cpL}([], P) = P$$
$$\mathsf{cpL}(l : L, P) = \begin{cases} \mathsf{cpL}(L, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P)) \\ \quad \textit{se } l := \mathtt{proc}(\vec{x})\{S\} \in P \\ \mathsf{cpL}(L, P), \textit{ se } l \notin LB(P) \end{cases}$$

Um *programa compilado* é um programa que não contém invocações de funções, i. e., não existe nenhum comando da forma `x:=l(E1,...,EN)` em nenhum dos procedimentos que constituem o programa.

A sequência de lemas e corolários que se seguem têm como objetivo a demonstração do Teorema da Compilação (Teor. 4). Este demonstra que, dado um programa bem formado, o processo de compilação remove, de facto, todas as invocações de procedimentos presentes no programa.

Lema 12.

$$l \notin \mathsf{inv}(S') \implies \mathsf{subs}(l := \mathrm{proc}(\vec{x})\{S\}, S') = S'$$

Demonstração. Por indução em S'.

Caso: S' da forma $\mathrm{ret}(E)$.

$$\begin{aligned} & \mathsf{subs}(l := \mathrm{proc}(\vec{x})\{S\}, \mathrm{ret}(E)) = \mathrm{ret}(E) \\ \Leftrightarrow & [\text{Def. de } \mathsf{subs}] \\ & \mathrm{ret}(E) = \mathrm{ret}(E) \end{aligned}$$

Caso: S' da forma $x := g(\vec{E}); S''$.

$$\begin{aligned} & \mathsf{subs}(l := \mathrm{proc}(\vec{x})\{S\}, x := g(\vec{E}); S'') = x := g(\vec{E}); S'' \\ \Leftrightarrow & [\text{Def. de } \mathsf{subs}] \\ & x := g(\vec{E}); \mathsf{subs}(l := \mathrm{proc}(\vec{x})\{S\}, S'') = x := g(\vec{E}); S'' \\ \Leftrightarrow & [\text{Hipótese de indução }] \\ & x := g(\vec{E}); S'' = x := g(\vec{E}); S'' \end{aligned}$$

Restantes casos: seguem de forma simples da hipótese de indução, analogamente ao caso anterior.

□

Lema 13.

$$\mathsf{wf}(L, P, S) \wedge L' \subseteq L \implies \mathsf{wf}(L', P, S)$$

Demonstração. Por indução no par $(\#(\mathrm{LB}(P) \setminus L), S)$, que é a medida de indução associada à função wf.

Caso: S da forma $x := l(\vec{E}); S'$.

Queremos $\mathsf{wf}(L', P, x := l(\vec{E}); S')$.

Da hipótese $\mathsf{wf}(L, P, x := l(\vec{E}); S')$, temos que,

- $l := \mathrm{proc}(\vec{y})\{S''\} \in P$ para algum $\vec{y}$ e S''; e
- $l \notin L$; e
- $\mathsf{wf}(\{l\} \cup L, P, S'')$; e
- $\mathsf{wf}(L, P, S')$.

Ora, se $l \notin L$ então $l \notin L'$. Logo, por definição de wf, temos

$$\mathsf{wf}(L', P, x := l(\vec{E}); S') \Leftrightarrow$$
$$\quad \mathsf{wf}(\{l\} \cup L', P, S'') \wedge \mathsf{wf}(L', P, S')$$

Por um lado, como $\#(LB(P) \setminus (L \cup \{l\})) < \#(LB(P) \setminus L)$, $\mathsf{wf}(\{l\} \cup L, P, S'')$ e $L' \cup \{l\} \subseteq L \cup \{l\}$ da hipótese de indução podemos concluir $\mathsf{wf}(\{l\} \cup L', P, S'')$. Por outro lado, como "$S' < S''$" e $\mathsf{wf}(L, P, S'')$, da hipótese de indução segue $\mathsf{wf}(L', P, S')$

Restantes casos: saem de forma mais simples da hipótese de indução e da definição de wf. □

Lema 14.

$$\mathsf{wf}(L, P, S) \wedge \mathsf{wf}(L, P, S')$$
$$\quad \Longrightarrow \mathsf{wf}(L, P, \mathsf{subs}(S, x, S'))$$

Demonstração. Por indução em S.

Caso: S da forma $\mathtt{ret}(E)$,

$$\begin{aligned}
&\mathsf{wf}(L, P, \mathsf{subs}(\mathtt{ret}(E), x, S')) \\
\Leftrightarrow&[\text{Def. de subs}] \\
&\mathsf{wf}(L, P, x := E; S') \\
\Leftrightarrow&[\text{Def. de wf}] \\
&\mathsf{wf}(L, P, S')
\end{aligned}$$

O que é verdade por hipótese.

Caso: S da forma $\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}$,

$$\begin{aligned}
&\mathsf{wf}(L, P, \mathsf{subs}(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}, x, S')) \\
\Leftrightarrow&[\text{Def. de subs}] \\
&\mathsf{wf}(L, P, \mathtt{if}(E)\{\mathsf{subs}(S_0, x, S')\}\mathtt{else}\{\mathsf{subs}(S_1, x, S')\}) \\
\Leftrightarrow&[\text{Def. de wf}] \\
&\mathsf{wf}(L, P, \mathsf{subs}(S_0, x, S')) \wedge \mathsf{wf}(L, P, \mathsf{subs}(S_1, x, S'))
\end{aligned}$$

O que é verdade por hipótese de indução, dado que a hipótese $\mathsf{wf}(L, P, S)$ implica $\mathsf{wf}(L, P, S_0)$ e $\mathsf{wf}(L, P, S_1)$.

Caso: S da forma $y := E; S_0$,

$$\begin{aligned}
&\mathsf{wf}(L, P, \mathsf{subs}(y := E; S_0, x, S')) \\
\Leftrightarrow&[\text{Def. de subs}] \\
&\mathsf{wf}(L, P, y := E; \mathsf{subs}(S_0, x, S')) \\
\Leftrightarrow&[\text{Def. de wf}] \\
&\mathsf{wf}(L, P, \mathsf{subs}(S_0, x, S'))
\end{aligned}$$

O que é verdade por hipótese de indução, dado que a hipótese $\mathsf{wf}(L, P, S)$ segue ainda $\mathsf{wf}(L, P, S_0)$.

Caso: S da forma $y := l(\vec{E}); S_0$,

$$
\begin{aligned}
& \mathsf{wf}(L, P, \mathsf{subs}(y := l(\vec{E}); S_0, x, S')) \\
\Leftrightarrow & [\text{Def. de subs}] \\
& \mathsf{wf}(L, P, y := l(\vec{E}); \mathsf{subs}(S_0, x, S')) \\
\Leftrightarrow & [\text{Da hipótese } \mathsf{wf}(L, P, S), \text{ tomando } l := \mathtt{proc}(\vec{z})\{S''\}] \\
& \mathsf{wf}(\{l\} \cup L, P, S'') \wedge \mathsf{wf}(L, P, \mathsf{subs}(S_0, x, S')) \wedge l \notin L
\end{aligned}
$$

Ora, $\mathsf{wf}(\{l\} \cup L, P, S'')$ e $l \notin L$ seguem da hipótese $\mathsf{wf}(L, P, S)$ e $\mathsf{wf}(L, P, \mathsf{subs}(S_0, x, S'))$ é verdade por hipótese de indução, dado que da hipótese $\mathsf{wf}(L, P, S)$ segue ainda $\mathsf{wf}(L, P, S_0)$. □

Lema 15.

$$
\begin{aligned}
& \mathsf{wf}(L, P, S') \wedge (l := \mathtt{proc}(\vec{x})\{S\}) \in P \wedge l \notin \mathsf{inv}(S) \\
& \implies \mathsf{wf}(L, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S')))
\end{aligned}
$$

Demonstração. Por indução no par $(\#(LB(P) \setminus L), S')$.

Caso: S' da forma $y := l(\vec{E}); S''$. Queremos,

$$\mathsf{wf}(L, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), \mathsf{subs}(S, x, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S'')))$$

Pelo Lema 14 basta que:

(a) $\mathsf{wf}(L, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), S)$; e

(b) $\mathsf{wf}(L, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S''))$.

(b) segue de imediato pela hipótese de indução ($S'' < S'$).

Da hipótese $\mathsf{wf}(L, P, S')$, conseguimos $\mathsf{wf}(L \cup \{l\}, P, S) \wedge l \notin L$. Daqui, por hipótese de indução ($\#(P \setminus (L \cup \{l\})) < \#(P \setminus L)$), segue

(i) $\mathsf{wf}(L \cup \{l\}, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S))$

Como $l \notin \mathsf{inv}(S)$, $\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S) = S$ (Lema 12).

Logo, de (i) e do Lema 13 segue (a).

Restantes casos: saem de forma mais simples da definição de wf e das respectivas hipóteses de indução. □

Corolário 3 (Preservação da boa formação pela substituição)**.**

1. $$\mathsf{wf}(P) \wedge l := \mathtt{proc}(\vec{x})\{S\}, l' := \mathtt{proc}(\vec{y})\{S'\} \in P \implies \mathsf{wf}(\{l'\}, \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P), \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S'))$$

2. $$\mathsf{wf}(P) \wedge l := \mathtt{proc}(\vec{x})\{S\} \in P \implies \mathsf{wf}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P))$$

Demonstração.

1. Segue de Lema 15. Note-se que:
   - $\mathsf{wf}(P)$ e $l' := \mathtt{proc}(\vec{y})\{S'\} \in P$ implicam que $\mathsf{wf}(\{l\}, P, S)$;
   - $\mathsf{wf}(P)$ e $l := \mathtt{proc}(\vec{x})\{S\} \in P$ implicam $\mathsf{wf}(\{l\}, P, S)$.

   que pelo Lema 11 implica $l \notin \mathsf{inv}(S)$.

2. Segue do ponto 1. Note-se que em 1 podemos tomar, em particular, $l = l'$.

□

Lema 16.

1. $\mathsf{inv}(\mathsf{subs}(S, x, S')) \subseteq \mathsf{inv}(S) \cup \mathsf{inv}(S')$

2. $\mathsf{inv}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, S')) \subseteq (\mathsf{inv}(S') \setminus \{l\}) \cup \mathsf{inv}(S)$

3. $\mathsf{inv}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P)) \subseteq (\mathsf{inv}(P) \setminus \{l\}) \cup \mathsf{inv}(S)$

Demonstração.

1. Por indução em S.

   Caso: S da forma $\mathtt{ret}(E)$.

$$
\begin{aligned}
& \mathsf{inv}(\mathsf{subs}(\mathtt{ret}(E), x, S')) \subseteq \mathsf{inv}(\mathtt{ret}(E)) \cup \mathsf{inv}(S') \\
\Leftrightarrow & [\text{Def. de } \mathsf{subs}] \\
& \mathsf{inv}(x := E; S') \subseteq \mathsf{inv}(\mathtt{ret}(E)) \cup \mathsf{inv}(S') \\
\Leftrightarrow & [\text{Def. de } \mathsf{inv}] \\
& \mathsf{inv}(S') \subseteq \{\} \cup \mathsf{inv}(S')
\end{aligned}
$$

   Caso: S da forma $y := f(\vec{E}); S_0$.

$$
\begin{aligned}
& \mathsf{inv}(\mathsf{subs}(y := f(\vec{E}); S_0, x, S')) \subseteq \mathsf{inv}(y := f(\vec{E}); S_0) \cup \mathsf{inv}(S') \\
\Leftrightarrow & [\text{Def. de } \mathsf{subs}] \\
& \mathsf{inv}(y := f(\vec{E}); \mathsf{subs}(S_0, x, S')) \subseteq \mathsf{inv}(y := f(\vec{E}); S_0) \cup \mathsf{inv}(S') \\
\Leftrightarrow & [\text{Def. de } \mathsf{inv}] \\
& \{f\} \cup \mathsf{inv}(\mathsf{subs}(S_0, x, S')) \subseteq \{f\} \cup \mathsf{inv}(S_0) \cup \mathsf{inv}(S') \\
\Leftrightarrow & [\text{Por hipótese de indução }] \\
& \{f\} \cup \mathsf{inv}(S_0) \cup \mathsf{inv}(S') \subseteq \{f\} \cup \mathsf{inv}(S_0) \cup \mathsf{inv}(S')
\end{aligned}
$$

   Restantes casos: Saem mais facilmente a partir das hipóteses de indução.

2. Por indução em S'.

   Caso: S' da forma $y := l(\vec{E}); S_0$. Por hipótese que indução sabemos que,

   $$\mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, S_0)) \subseteq \mathsf{inv}(S_0) \setminus \{l\} \cup \mathsf{inv}(S)$$

   Queremos mostrar que,

   $$\mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, S')) \subseteq (\mathsf{inv}(S') \setminus \{l\}) \cup \mathsf{inv}(S)$$

   Ora,

   $$\begin{aligned}
   & \mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, S')) \\
   \Leftrightarrow & [\text{Def. de subs}] \\
   & \mathsf{inv}(x_1 = E_1; \ldots; x_n = E_n; \mathsf{subs}(S, y, \mathsf{subs}(l := proc(\vec{x})\{S\}, S_0))) \\
   \Leftrightarrow & [\text{Def. de inv}] \\
   & \mathsf{inv}(\mathsf{subs}(S, y, \mathsf{subs}(l := proc(\vec{x})\{S\}, S_0))) \\
   \Leftrightarrow & [\text{Por hipótese de indução e pelo ponto anterior }] \\
   & \mathsf{inv}(S) \cup ((\mathsf{inv}(S_0) \setminus \{l\}) \cup \mathsf{inv}(S))
   \end{aligned}$$

   Logo é verdade que,

   $$\mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, S')) \subseteq (\mathsf{inv}(S') \setminus \{l\}) \cup \mathsf{inv}(S)$$

   pois $(\mathsf{inv}(S_0) \setminus \{l\}) \subseteq (\mathsf{inv}(S') \setminus \{l\})$.

   Restantes casos: Saem mais facilmente da hipótese de indução.

3. Por indução em P.

   Caso: $P = []$. Trivial

   Caso: $P = (g := proc(\vec{y})\{S'\}) : P'$. Assumimos por hipótese de indução que a propriedade é verdade para P', ou seja,

   $$\mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, P')) \subseteq (\mathsf{inv}(P') \setminus \{l\}) \cup \mathsf{inv}(S)$$

   Por definição de subs e inv temos que,

   $$\begin{aligned}
   & \mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, (g := proc(\vec{y})\{S'\}) : P')) \\
   \Leftrightarrow & [\text{Defs. de subs e inv}] \\
   & \mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, S')\}) \cup \mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, P')) \\
   \subseteq & [\text{Pelo ponto anterior}] \\
   & (\mathsf{inv}(S') \setminus \{l\}) \cup \mathsf{inv}(S) \cup \mathsf{inv}(\mathsf{subs}(l := proc(\vec{x})\{S\}, P')) \\
   \subseteq & [\text{Por hipótese de indução}] \\
   & (\mathsf{inv}(S') \setminus \{l\}) \cup \mathsf{inv}(S) \cup (\mathsf{inv}(P') \setminus \{l\}) \cup \mathsf{inv}(S) \\
   = & [\text{Propriedades de conjuntos}] \\
   & (\mathsf{inv}(S') \cup \mathsf{inv}(P')) \setminus \{l\} \cup \mathsf{inv}(S) \\
   = & [\text{Def. de inv}] \\
   & \mathsf{inv}(g := proc(\vec{y})\{S'\} : P') \setminus \{l\} \cup \mathsf{inv}(S)
   \end{aligned}$$

□

Lema 17.

$$\mathsf{wf}(P) \implies \mathsf{inv}(P) \subseteq LB(P)$$

Demonstração. Vamos assumir por contradição que existe $l \in \mathsf{inv}(P)$ e que $l \notin LB(P)$. Então existe pelo menos um comando da forma $x := l(\ldots); S$ em algum procedimento de P. Então, para algum L, temos que ter $\mathsf{wf}(L, P, x := l(\ldots); S) = \textbf{True}$, mas também não é possivél pois $l \notin LB(P)$. E então não é verdade que existe l nas condições acima. □

Abaixo, iremos algumas vezes sobrecarregar a notação $LB(P)$ e usá-la para significar também o conjunto de "labels" que dão nome aos procedimentos de P. Note-se que, no caso de não existirem procedimentos com o mesmo nome em P(o que está garantido quando se assume $\mathsf{wf}(P)$), a lista $LB(P)$ (Def. 31), produz uma enumeração do conjunto $LB(P)$.

Lema 18.

$$\mathsf{wf}(P) \wedge \mathsf{inv}(P) \subseteq X \subseteq LB(P) \wedge L \textit{ enumeração de } X \\ \implies \mathsf{inv}(\mathsf{cpL}(L, P)) = \varnothing$$

Demonstração. Por indução em #X.
Caso: $\#X = 0$. Trivial.
Caso: $\#X > 0$.
Sejam

- $L = l : L'$ uma enumeração de X; e
- $l := \mathtt{proc}(\vec{x})\{S\} \in P$.

Por definição,

$$\mathsf{cpL}(L, P) = \mathsf{cpL}(L', \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P))$$

Pelo Lema 16,

$$\mathsf{inv}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P)) \subseteq \mathsf{inv}(P) \setminus \{l\} \cup \mathsf{inv}(S)$$

1. De $\mathsf{wf}(P)$ e $l := \mathtt{proc}(\vec{x})\{S\} \in P$ seque $l \notin \mathsf{inv}(S)$. Logo $\mathsf{inv}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P)) \subseteq X \setminus \{l\}$.
2. Como o processo de substituição não afeta as "labels" de um programa e $X \subseteq LB(P)$, $X \setminus \{l\} \subseteq LB(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P))$.
3. Usando novamente $\mathsf{wf}(P)$ e $l := \mathtt{prco}(\vec{x})\{S\} \in P$, pelo Corol. 3 temos a garantia que $\mathsf{wf}(\mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P))$.

De 1, 2 e 3 e como L' é uma enumeração de $X \setminus \{l\}$ por hipótese de indução (note-se que $\#(X \setminus \{l\}) < \#(X)$), segue $\mathsf{inv}(\mathsf{cpL}(L', \mathsf{subs}(l := \mathtt{proc}(\vec{x})\{S\}, P))) = \varnothing$, com pretendido. □

Ainda relativamente ao processo de compilação, apresentamos agora o resultado que garante, de facto, que um programa compilado é um programa sem invocações de procedimentos.

Teorema 4 (Teorema da Compilação $\mathcal{LA}$)**.**

$$\mathsf{wf}(P) \implies \mathsf{inv}(\mathsf{cp}(P)) = \varnothing$$

Demonstração. Consequência do Lema 18, tomando para X o conjunto $LB(P)$ e para L a lista $LB(P)$. Note-se que $\mathsf{inv}(\mathsf{cpL}(LB(P), P)) = \mathsf{inv}(\mathsf{cp}(P))$ e que, de $\mathsf{wf}(P)$ segue $\mathsf{inv}(P) \subseteq LB(P)$ (Lema 17), mais ainda a lista $LB(P)$ é uma enumeração do conjunto $LB(P)$. □

Corolário 4 (Preservação da Boa Formação pela compilação $\mathcal{LA}$)**.**

$$\mathsf{wf}(P) \implies \mathsf{wf}(\mathsf{cp}(P))$$

Demonstração. Pelo Teor. 4, temos que $\mathsf{inv}(\mathsf{cp}(P)) = \varnothing$. Ora por definição, o único caso em que a função wf poderá retornar **False** é no caso de encontrar um comando de invocação. Como o programa compilado não contem invocações então $\mathsf{wf}(P) \implies \mathsf{wf}(\mathsf{cp}(P))$. □

No início desta secção sobre procedimentos referimos algumas técnicas e aspetos a ter em conta quando queremos introduzir procedimentos respeitar o formato *SA*. Resta então a questão: o que acontece durante o processo de compilação no que toca ao formato *SA*?

Para dar resposta à questão levantada, iremos considerar o caso "extremo" em que um programa se diz no formato *SA* quando cada um dos procedimento está no formato *SA* e não existe nenhuma variável com "nome" comum em múltiplos procedimentos. A Figura 21 serve como exemplo para um programa com procedimentos no formato *SA* obedecendo a estas condições.

```
g := proc(c,d){
a := c;
b := f(d,a);
if(E) {
          x := a;
          y := x;
          ret(y)
        } else {
          k := b;
          j := k;
          ret(j)
        }
}

f := proc(h,i){
    w := h;
    e := i;
    if(E1){
        ret(e)
    } else {
        ret(w)
    }
}
```

Figura 21: Ex. Programa com procedimentos no formato *SA*

No entanto, é preciso ter em conta que a forma como definimos a compilação não preserva o formato *SA* estático (considerado no Capítulo 2). Tomando como exemplo a Figura 21 sabemos que o processo de compilação irá replicar o código após o comando de invocação da função `f` (`b := f(d,a)`) tantas vezes quantos comandos `ret` existem no procedimento `f` (neste caso duas vezes). Na Figura 22 surge então, o resultado da compilação do programa na Figura 21.

```
g := proc(c,d){
a := c;
h := d;
i := a;
w := h;
e := i;
if(E1){
    b := e;
    if(E) {
          x := a;
          y := x;
          ret(y)
        } else {
          k := b;
          j := k;
          ret(j)
        }
} else {
    b := w;
    if(E) {
          x := a;
          y := x;
          ret(y)
        } else {
          k := b;
          j := k;
          ret(j)
        }
}
}

f := proc(h,i){
    w := h;
    e := i;
    if(E1){
        ret(e)
    } else {
        ret(w)
    }
}
```

Figura 22: Ex. Programa da Figura 21 após compilação

Observando o resultado da compilação da Figura 22, podemos afirmar que o código não está no formato *SA* estático (por exemplo, existem duas atribuições de x). Sendo isto originado pela replicação de código gerado através dos múltiplos comandos `ret`, que por sua vez

são gerados pelo divisão do fluxo que o comando `if` oferece. Contudo, podemos afirmar que o programa se encontra no formato *SA* dinâmico. Um facto que não iremos demonstrar é o de que um programa *SA* após compilado poderá gerar um *DSA*. Existem algumas alternativas para resolver este problema e fazer com que o programa após compilado, se mantenha no formato $\mathcal{SA}$ estático. Eis duas delas:

1. Redefinir o processo de expansão de invocação de procedimentos que definimos para efetuar uma renomeação de variáveis (uma espécie de micro tradução *SSA*), sempre que existe duplicação de código.

2. No final do processo de compilação que definimos, efetuar uma tradução *SSA*.

Ao contrário da primeira solução, que permite que o processo de compilação preserve a estrutura *SA* ao expandir as diversas invocações de procedimentos, a segunda solução consiste numa mera composição de traduções.

Na nossa opinião, seria possível incorporar a primeira solução no nosso processo de compilação, por intremédio do aumento do nível de complexidade do estudo.

Uma observação final que não podemos deixar de salientar prende-se com a inexistência das chamadas funções ϕ. Esta função surge usualmente na literatura ([13]) para representar a estrutura auxiliar responsável por manter informação sobre o caminho que está a ser percorrido na árvore de execução do programa. Este estrutura encarrega-se de fazer atribuições às variáveis conforme o ramo da execução que foi computado. No caso deste trabalho, dado que não existe composição de comandos "if" e como também não pretendemos analisar casos que envolvam recursão (direta ou indireta), não necessitamos de recorrer a esta estrutura auxiliar.

3.2.2 *Procedimentos em $\mathcal{CPS}$*

Tal como fizemos para a linguagem $\mathcal{LA}$, também para linguagem $\mathcal{CPS}$ iremos introduzir procedimentos, tendo sempre em consideração que as relação precisas entre $\mathcal{CPS}$ e $\mathcal{LA}$ expressas no Capítulo 2 se mantêm.

Assim, podemos considerar que um procedimento em λ-termos consiste numa abstração que recebe parâmetros de entrada, ou seja:

$$\lambda_{proc}\vec{x}.\ M$$

Note-se que a palavra "proc" serve para distinguir uma abstração que é um λ-procedimento. $\vec{x}$ é, tal como anteriormente, um vetor de variáveis

$$\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.M))$$

que corresponde a uma função com n parâmetros.

Aplicando a transformação CPS obtemos

$$(\lambda k.k(\lambda x_1.(\lambda k.k(\lambda x_2.(\ldots(\lambda k.k(\lambda x_n.M))\ldots))))$$

Mantendo a mesma consistência e linha de raciocínio, é então necessário passar os parâmetros à "esquerda" e sob a forma de "returns", i. e., da forma $(\lambda k.kB)$. Deste modo, sempre que desejarmos passar parâmetros a uma função, obtemos termos da seguinte forma:

$$\begin{aligned} & ((\lambda k.kB_n)(\ldots((\lambda k.kB_1)(\lambda_{proc}\vec{x}.M))) \\ \equiv\ & ((\lambda k.kB_n)(\ldots((\lambda k.kB_1)(\lambda k.k(\lambda x_1.(\ldots M))))) \end{aligned}$$

Quanto às reduções, serão feitas da forma usual, como se segue.

$$\begin{aligned} & ((\lambda k.kB_n)(\ldots((\lambda k.kB_1)(\lambda_{proc}\vec{x}.M))) \\ \equiv & ((\lambda k.kB_n)(\ldots((\lambda k.kB_1)(\lambda k.k(\lambda x_1.(\ldots M))))) \\ \to_\beta & ((\lambda k.kB_n)(\ldots(\lambda k.k(\lambda x_1.(\ldots M))B_1))) \\ \to_\beta & ((\lambda k.kB_n)(\ldots B_1(\lambda x_1.(\ldots M))) \end{aligned}$$

De seguida, e tal como procedemos para a linguagem $\mathcal{LA}$, apresentaremos para a linguagem $\mathcal{CPS}$ o mesmo método que permite referenciar procedimentos "externos", rotulando-os. Vejamos o seguinte exemplo:

$$f = \lambda_{proc}\vec{x}.M$$

Figura 23: Ex. Procedimento na Linguagem $\mathcal{CPS}$

Sempre que quisermos invocar um procedimento rotulado como f usamos a seguinte sintaxe:

$$(f\ B_1\ B_2\ \ldots\ B_n)$$

Também a ideia do pré-processamento segue a mesma apresentação que para a linguagem $\mathcal{LA}$. Atentemos no seguinte exemplo: Seja

$f = \lambda_{proc}\vec{x}.(\lambda k.kx_1)(\lambda x_2.(if(\lambda k.kx_2)(\lambda k.k(\lambda k.kx_2))(\lambda k.k(\lambda k.kx_1))))$, com $\vec{x} = (x_1, x_2)$. Informalmente temos,

1. x_2 "toma o valor de" x_1
2. Se x_2 for "verdade", x_2 é "retornado"
3. Senão x_1 é "retornado"

Suponhamos que queremos pré-processar o seguinte programa:

$$\lambda_{proc}\vec{y}.(f\ y_1\ y_2)(\lambda y_3.(\lambda k.k(\lambda k.ky_3)))$$

com $\vec{y} = (y_1, y_2)$. Informalmente temos,

1. y_3 "toma o valor de" $(f\ y_1\ y_2)$
2. y_3 é "retornado"

Então, em analogia com o que acontece para $\mathcal{LA}$, temos a seguinte expansão:

$$\begin{aligned}
&\lambda_{proc}\vec{y}.(\lambda k.ky_1)(\lambda x_1.((\lambda k.ky_2)(\lambda x_2.\\
&\qquad\qquad (if(\lambda k.kx_2)\\
&\qquad\qquad ((\lambda k.kx_2)(\lambda y_3.(\lambda k.k(\lambda k.ky_3))))\\
&\qquad\qquad ((\lambda k.kx_1)(\lambda y_3.(\lambda k.k(\lambda k.ky_3))))\\
&\qquad))))
\end{aligned}$$

O programa pode ser lido da seguinte forma:

1. x_1 "toma o valor de" y_1
2. x_2 "toma o valor de" y_2
3. x_2 "toma o valor de" x_1
4. Se x_2 for "verdade", y_3 "toma o valor de" x_2 e é "retornado"
5. Senão y_3 "toma o valor de" x_1 e é "retornado"

Ou seja, temos um comportamento idêntico ao da invocação em $\mathcal{LA}$, em que a passagem de parâmetros e o return são convertidos em atribuições. Sendo assim a nossa gramática, incluindo os comandos "if-else" é agora a seguinte:

$$\begin{aligned}
T &::= [\,] \mid (l := \lambda_{proc}\vec{x}.M) : T\\
M &::= B(\lambda x.M)\\
&\quad \mid\ (l\ \vec{B})(\lambda x.M)\\
&\quad \mid\ (if\ B\ M\ M)\\
&\quad \mid\ (\lambda k.kB)\\
B &::= \lambda k.kx\\
&\quad \mid\ \lambda k.kc
\end{aligned}$$

Com l um "nome" de um procedimento e $\vec{B}$ um vetor de "expressões base". Vamos usar $\mathcal{T}$ para representar o conjunto de todos os programas.

Tal como fizemos para a linguagem $\mathcal{LA}$ temos agora de definir o conceito de *compilação*. Como já referido, o processo de compilação em $\mathcal{CPS}$ não será mais do que uma réplica do processo de compilação em $\mathcal{LA}$(, vice-versa).

Comecemos por definir as funções inv e LB para o contexto funcional.

Definição 38. *Seja* $\mathsf{inv} : \mathcal{T} \to \wp(\mathcal{R})$ *a função definida do seguinte modo*

$$\begin{aligned} \mathsf{inv}([]) &= \{\} \\ \mathsf{inv}((l := \lambda_{proc}\vec{x}.M) : T') &= \mathsf{inv}(M) \cup \mathsf{inv}(T') \end{aligned}$$

onde $\mathsf{inv} : \mathcal{M} \to \wp(\mathcal{R})$ *é a função definida do seguinte modo*

$$\begin{aligned} \mathsf{inv}((\lambda k.kB)) &= \{\} \\ \mathsf{inv}(B(\lambda x.M)) &= \mathsf{inv}(M) \\ \mathsf{inv}((l\ \vec{B})(\lambda x.M)) &= \{l\} \cup \mathsf{inv}(M) \\ \mathsf{inv}((\mathtt{if}\ B\ M_0\ M_1)) &= \mathsf{inv}(M_0) \cup \mathsf{inv}(M_1) \end{aligned}$$

Definição 39. *Seja* $\mathsf{LB} : \mathcal{T} \to [\mathcal{R}]$ *a função definida da seguinte forma*

$$\begin{aligned} \mathsf{LB}([]) &= [] \\ \mathsf{LB}((l := \lambda_{proc}\vec{x}.M) : T') &= l : \mathsf{LB}(T') \end{aligned}$$

Definição 40. T *diz-se* bem formado *se* $\mathsf{wf}(T) = \textbf{\textit{True}}$*, com* wf *definido da seguinte forma*

$$\forall l := \lambda_{proc}\vec{x}.M \in T, \quad \mathsf{wf}(\{l\}, T, M) = \textbf{\textit{True}}$$

onde
$\mathsf{wf} : \wp(\mathcal{R}) \times \mathcal{P} \times \mathcal{S} \to \mathtt{Bool}$ *é definida da seguinte forma:*

$$\mathsf{wf}(L, T, (\lambda k.kB)) = \textbf{\textit{True}}$$

$$\mathsf{wf}(L, T, B(\lambda x.M)) = \mathsf{wf}(L, T, M)$$

$$\mathsf{wf}(L, T, (l\ \vec{B})(\lambda x.M)) = \begin{cases} \textbf{\textit{False}}, & se\ l \in L \\ \textbf{\textit{False}}, & se\ l \notin \mathsf{LB}(T) \\ (\mathsf{wf}(l : L, T, M') \wedge \mathsf{wf}(L, T, M)) & \\ \quad se\ l := \lambda_{proc}\vec{x}.M' \in T\ e\ l \notin L & \end{cases}$$

$$\mathsf{wf}(L, T, (\mathtt{if}\ B\ M_0\ M_1)) = \mathsf{wf}(L, T, M_0) \wedge \mathsf{wf}(L, T, M_1)$$

Passemos então e definir as funções de substituição necessárias para o processo de compilação:

Definição 41. *Seja* $\mathsf{subs} : \mathcal{M} \times \mathtt{Vars} \times \mathcal{M} \to \mathcal{M}$ *a função definida da seguinte forma:*

$$\begin{aligned} \mathsf{subs}((\lambda k.kB), x, M) &= B(\lambda x.M) \\ \mathsf{subs}(B(\lambda y.M'), x, M) &= B(\lambda y.\mathsf{subs}(M', x, M)) \\ \mathsf{subs}((f\ \vec{B})(\lambda y.M'), x, M) &= (f\ \vec{B})(\lambda y.\mathsf{subs}(M', x, M)) \\ \mathsf{subs}((\mathtt{if}\ B\ M_0\ M_1), x, M) &= (\mathtt{if}\ B\ \mathsf{subs}(M_0, x, M)\ \mathsf{subs}(M_1, x, M)) \end{aligned}$$

Definição 42. *Seja* subs : ***Proc*** $\times \mathcal{M} \to \mathcal{M}$*, a função definida da seguinte forma:*

$$
\begin{aligned}
&\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, (\lambda k.kB)) = (\lambda k.kB)\\
&\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, B(\lambda x.M')) = B(\lambda x.\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, M'))\\
&\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, (g\ \vec{B})(\lambda x.M')) =\\
&\quad\begin{cases}
B_1(\lambda z_1.(\ldots B_n(\lambda z_n.\mathsf{subs}(M, x, \mathsf{subs}(l := \lambda_{proc}\vec{z}.M, M')))\ldots))\\
\qquad \textit{caso } g = l\\
(g\ \vec{B})(\lambda x.\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, M'))\\
\qquad \textit{caso } g \neq l
\end{cases}\\
&\mathsf{subs}(l := \lambda_{proc}\vec{z}.M, (\mathrm{if}\ B\ M_0\ M_1)) =\\
&\qquad (\mathrm{if}\ B\ \mathsf{subs}(l := \lambda_{proc}\vec{z}.M, M_0)\ \mathsf{subs}(l := \lambda_{proc}\vec{z}.M, M_1))
\end{aligned}
$$

Definição 43. *Seja* subs : ***Proc*** $\times \mathcal{P} \to \mathcal{P}$*, a função definida da seguinte forma:*

$$
\begin{aligned}
&\mathsf{subs}(l := \lambda_{proc}\vec{x}.M, []) = []\\
&\mathsf{subs}(l := \lambda_{proc}\vec{x}.M, (g := \lambda_{proc}\vec{y}.M') : T') =\\
&\quad (g := \lambda_{proc}\vec{y}.\mathsf{subs}(l := \lambda_{proc}\vec{x}.M, M')\}) : \mathsf{subs}(l := \lambda_{proc}\vec{x}.M, T')
\end{aligned}
$$

Podemos, finalmente, definira a função de compilação, e analogia à compilação para $\mathcal{LA}$.

Definição 44. *Seja* cp : $\mathcal{T} \to \mathcal{T}$ *a função definida da seguinte forma:*

$$
\mathsf{cp}(T) = \mathsf{cpL}(\mathsf{LB}(T), T)
$$

$$
\begin{aligned}
&\mathsf{cpL} : [\mathcal{R}] \times \mathcal{T} \to \mathcal{T}\\
&\mathsf{cpL}([], T) = T\\
&\mathsf{cpL}(l : L, T) = \begin{cases}
\mathsf{cpL}(L, \mathsf{subs}(l := \lambda_{proc}\vec{x}.M, T))\\
\quad \textit{se } l := \lambda_{proc}\vec{x}.M \in T\\
\mathsf{cpL}(L, T), \textit{ se } l \notin \mathsf{LB}(T)
\end{cases}
\end{aligned}
$$

Teorema 5 (Teorema da Compilação $\mathcal{CPS}$)**.**

$$
\mathsf{wf}(T) \implies \mathsf{inv}(\mathsf{cp}(T)) = \varnothing
$$

Demonstração. Demonstração análoga à do Teor. 4, que, portanto, requer, em particular, lemas análogos ao lemas 11 a 18 estabelecidos para $\mathcal{LA}$. □

Teorema 6 (Teorema da Preservação da Boa Formação de $\mathcal{CPS}$)**.**

$$
\mathsf{wf}(T) \implies \mathsf{wf}(\mathsf{cp}(T))
$$

Demonstração. Análoga à justificação na prova do Corol. 4 □

3.3 ESTENDENDO A RELAÇÃO ENTRE $\mathcal{LA}$ E $\mathcal{CPS}$

3.3.1 *Relação sintática entre programas*

Temos em primeiro lugar de expandir as funções **F** e **I**, de forma a abranger as novas construções, ou seja, iremos criar mecanismos de tradução idênticos aos do capítulo anterior sobrecarregando os nomes de funções usados. Iremos também utilizar os nomes $\mathcal{LA}$ e $\mathcal{CPS}$ para as linguagens estendidas que apresentamos nas secções anteriores.

Definição 45. *A função* $\boldsymbol{F}$: $\mathcal{CPS} \to \mathcal{LA}$ *está definida do seguinte modo:*

$$\begin{aligned}
\boldsymbol{F}([]) &= [] \\
\boldsymbol{F}(l := \lambda_{\mathtt{proc}}\vec{x}.M \ : \ T') &= (l := \mathtt{proc}(\vec{x})\{\boldsymbol{F}(M)\}) \ : \ \boldsymbol{F}(T') \\
\boldsymbol{F}((l\ \vec{x})(\lambda y.M)) &= y := l(\vec{x}); \boldsymbol{F}(M) \\
\boldsymbol{F}((\mathtt{if}\ B\ M_1\ M_2)) &= \mathtt{if}(\boldsymbol{F}(B))\{\boldsymbol{F}(M_1)\}\mathtt{else}\{\boldsymbol{F}(M_2)\} \\
\boldsymbol{F}(B(\lambda x.M)) &= x := \boldsymbol{F}(B);\ \boldsymbol{F}(M) \\
\boldsymbol{F}(\lambda k.kB) &= \mathtt{ret}(\boldsymbol{F}(B)) \\
\boldsymbol{F}(\lambda k.kx) &= x \\
\boldsymbol{F}(\lambda k.kc) &= c
\end{aligned}$$

Definição 46. *A função* $\boldsymbol{I} : \mathcal{LA} \to \mathcal{CPS}$ *está definida do seguinte modo:*

$$\begin{aligned}
\boldsymbol{I}([]) &= [] \\
\boldsymbol{I}(l := \mathtt{proc}(\vec{x})\{S\} \ : \ P) &= (l := \lambda_{\mathtt{proc}}\vec{x}.\boldsymbol{I}(S)) \ : \ \boldsymbol{I}(P) \\
\boldsymbol{I}(y := l(\vec{x}); S) &= (l\ \vec{x})(\lambda y.\boldsymbol{I}(S)) \\
\boldsymbol{I}(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}) &= (\mathtt{if}\ \boldsymbol{I}(E)\ \boldsymbol{I}(S_0)\ \boldsymbol{I}(S_1)) \\
\boldsymbol{I}(x := E;\ S) &= \boldsymbol{I}(E)(\lambda x.\boldsymbol{I}(S)) \\
\boldsymbol{I}(\mathtt{ret}(E)) &= (\lambda k.k\ \boldsymbol{I}(E)) \\
\boldsymbol{I}(x) &= \lambda k.kx \\
\boldsymbol{I}(c) &= \lambda k.kc
\end{aligned}$$

As proposições 7 e 8 trazem-nos tal como no capítulo anterior (Prop. 4) a preservação da da sintaxe pelas tradução.

Proposição 7.

1. $\boldsymbol{I}(\boldsymbol{F}(T)) = T$
2. $\boldsymbol{I}(\boldsymbol{F}(M)) = M$
3. $\boldsymbol{I}(\boldsymbol{F}(B)) = B$

Demonstração. Nos pontos 2 e 3, a demonstração segue por indução estrutural em M e T, respectivamente. Em todos os pontos, a demonstração é análoga e "estende" a demonstração da Prop. 4. Vamos apenas mostrar o resultado para as novas construções:

1.

Caso: $T = []$. Trivial pois é igual à lista vazia.

Caso: T da forma $l := \lambda_{proc}.\vec{x}.M : T'$.

Por hipótese de indução, sabemos que, $\mathbf{I}(\mathbf{F}(T')) = T'$ e pelo ponto 2 sabemos também que $\mathbf{I}(\mathbf{F}(M)) = M$. Logo

$$\mathbf{I}(\mathbf{F}(l := \lambda_{proc}.\vec{x}.M)) = l := \lambda_{proc}.\vec{x}.\mathbf{I}(\mathbf{F}(M)) \text{ e } \mathbf{I}(\mathbf{F}(T)) = T$$

2.

Caso M da forma $(\text{if } B\ M_1\ M_2)$.

$$\begin{aligned}\mathbf{I}(\mathbf{F}((\text{if } B\ M_1\ M_2))) &= \mathbf{I}(\text{if}(\mathbf{F}(B))\{\mathbf{F}(M_1)\}else\{\mathbf{F}(M_2)\}) \\ &= (\text{if } \mathbf{I}(\mathbf{F}(B))\ \mathbf{I}(\mathbf{F}(M_1))\ \mathbf{I}(\mathbf{F}(M_2)))\end{aligned}$$

Por motivos idênticos ao apresentados na Prop. 4 é verdade que

$$\begin{aligned}&\mathbf{I}(\mathbf{F}(B)) = B \\ &\mathbf{I}(\mathbf{F}(M_1)) = M_1 \\ &\mathbf{I}(\mathbf{F}(M_2)) = M_2\end{aligned}$$

Então $\mathbf{I}(\mathbf{F}((\text{if } B\ M_1\ M_2))) = (\text{if } B\ M_1\ M_2)$.

Caso M da forma $(f\ \vec{x})(\lambda y.M')$.

$$\begin{aligned}\mathbf{I}(\mathbf{F}((f\ \vec{x})(\lambda y.M'))) &= \mathbf{I}(y := f(\vec{x}); \mathbf{F}(M')) \\ &= (f\ \vec{x})(\lambda y.\mathbf{I}(\mathbf{F}(M')))\end{aligned}$$

Por motivos idênticos aos apresentados na Prop. 4 é verdade que

$$\mathbf{I}(\mathbf{F}(M)) = M$$

□

Proposição 8.

1. $\boldsymbol{F}(\boldsymbol{I}(P)) = P$
2. $\boldsymbol{F}(\boldsymbol{I}(S)) = S$
3. $\boldsymbol{F}(\boldsymbol{I}(E)) = E$

Demonstração. Análoga à anterior □

O Teor. 7 apresenta-nos a preservação da sintaxe pelo processo de compilação.

Teorema 7 (Comutatividade da Compilação com as traduções em $\mathcal{LA}$ e $\mathcal{CPS}$)**.**

1. $\mathsf{wf}(P) \implies \boldsymbol{I}(\mathsf{cp}(P)) = \mathsf{cp}(\boldsymbol{I}(P))$
2. $\mathsf{wf}(T) \implies \boldsymbol{F}(\mathsf{cp}(T)) = \mathsf{cp}(\boldsymbol{F}(T))$

Não faremos uma demonstração detalhada dado que, além das proposições 7 e 8 (acerca de comutatividade das traduções **F** e **I**), esta também requer diversos lemas acerca da comutatividade das traduções **F** e **I** assim como das diversas funções usadas nos processos de compilação. Note-se no entanto que o desenho do processo de compilação para $\mathcal{CPS}$ foi concebido de forma a ser uma "cópia" do processo de compilação para $\mathcal{LA}$.

A Figura 24 ilustra este resultado.

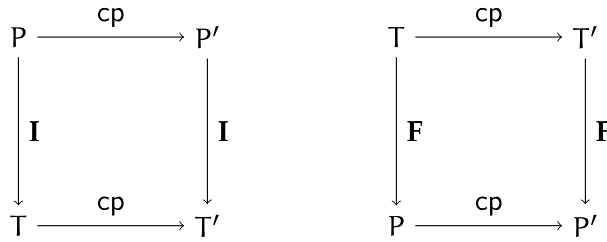

Figura 24: Diagrama da comutação da Compilação c/ as traduções **F** e **I**

3.3.2 *Preservação da semântica para a extensão da linguagem*

Nesta seção vamos verificar a preservação da semântica se mantém para as linguagens estendidas. Sempre que nos referirmos a um programa compilado, estaremos a fazê-lo assumindo que existe um programa bem formado que foi sujeito ao processo de compilação/expansão, i. e., o processo de compilação irá atuar sempre sobre programas bem formados.

Agora, o objectivo passa por utilizar conceitos idênticos aos do Capítulo 2 para falar sobre a relação entre a execução de programas compilados. As proposições e teoremas que se seguem constituem então a demonstração de que a semântica de um programa compilado é preservada pelas traduções.

Proposição 9. *Sejam* P *um programa compilado,* $l := \mathtt{proc}(\vec{x})\{S\} \in P$ *e* σ *um estado.*

$$X = var(S) \wedge (S,\sigma) \rightsquigarrow (S',\sigma') \implies (\boldsymbol{I}(S), \mathsf{sc}(\sigma, X)) \rightarrow_\beta (\boldsymbol{I}(S'), \mathsf{sc}(\sigma', X))$$

Demonstração. Comecemos por notar que sendo P um programa compilado, S não tem invocações. Mostraremos o caso do construtor "if-else" (todos os outros seguem o mesmo padrão de demonstração apresentado em Teor. 1).

Caso: S da forma $\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}$.

Sub-caso: $[\![E]\!]_\sigma = \mathbf{v}$. Assim, $(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\},\sigma) \rightsquigarrow (S_0,\sigma)$, e portanto $S' = S_0$ e $\sigma' = \sigma$.
Ora

$$\begin{aligned} & (\mathbf{I}(\mathtt{if}(E)\{S_0\}\mathtt{else}\{S_1\}), \mathsf{sc}(\sigma, X)) \\ \Leftrightarrow & [\text{Def. } \mathbf{I}] \\ & ((\mathtt{if}\ \mathbf{I}(E)\ \mathbf{I}(S_0)\ \mathbf{I}(S_1)), \mathsf{sc}(\sigma, X)) \\ \to_\beta & [\text{De } [\![E]\!]_\sigma = \mathbf{v} \text{ segue } [\![\mathbf{I}(E)]\!]_{\mathsf{sc}(\sigma,X))} = \mathbf{v}, \text{pelo Lema 4}] \\ & (\mathbf{I}(S_0), \mathsf{sc}(\sigma, X)) \end{aligned}$$

Sub-caso: $[\![E]\!]_\sigma = \mathbf{f}$. Análogo ao sub-caso anterior. □

Proposição 10. *Sejam* T *um programa compilado,* $l := \lambda_{proc}\vec{x}.M$ *e* δ *um fecho para* M.

$$(M,\delta) \to_\beta (M',\delta') \implies (\boldsymbol{F}(M), \mathsf{cs}(\delta)) \rightsquigarrow (\boldsymbol{F}(M'), \mathsf{cs}(\delta'))$$

Demonstração. Análogo à demonstração da proposição anterior, usando desta vez as ideias apresentadas em Teor. 2 (em vez do Teor. 1). □

Proposição 11. *Sejam* P *um programa compilado,* $l := \mathtt{proc}(\vec{x})\{S\} \in P$ *e* σ *um estado.*

$$\begin{aligned} & X = var(S) \wedge (S,\sigma) \rightsquigarrow^n (S',\sigma') \\ & \qquad \implies (\boldsymbol{I}(S), \mathsf{sc}(\sigma, X)) \to_\beta^n (\boldsymbol{I}(S'), \mathsf{sc}(\sigma', X)) \end{aligned}$$

Demonstração. Recorre-se à proposição Prop. 9 e às ideias apresentadas no Corol. 1. □

Proposição 12. *Sejam* T *um programa compilado,* $l := \lambda_{proc}\vec{x}.M$ *e* δ *um fecho para* M.

$$(M,\delta) \to_\beta^n (M',\delta') \implies (\boldsymbol{F}(M), \mathsf{cs}(\delta)) \rightsquigarrow^n (\boldsymbol{F}(M'), \mathsf{cs}(\delta'))$$

Demonstração. Recorre-se à proposição Prop. 10 e às ideias apresentadas no Corol. 2. □

Teorema 8 (Preservação da semântica por **I**)**.** *Sejam* P *um programa compilado,* $l := \mathtt{proc}(\vec{x})\{S\} \in P$ *e* σ *um estado.*

$$(S,\sigma) \textit{ avalia em } v \Leftrightarrow (\boldsymbol{I}(S), \mathsf{sc}(\sigma, X)) \textit{ avalia em } v$$

Demonstração. Imediata das proposições anteriores e das ideias apresentadas na Teor. 3. □

Teorema 9 (Preservação da semântica por **F**)**.** *Sejam* T *um programa compilado,* $l := \lambda_{proc}\vec{x}.M$ *e* δ *um fecho para* M.

$$(M,\delta) \textit{ avalia em } v \Leftrightarrow (\boldsymbol{F}(M), \mathsf{cs}(\delta)) \textit{ avalia em } v$$

Demonstração. Análoga à anterior. □

4 IMPLEMENTAÇÃO

No presente capítulo procede-se à descrição da implementação dos conceitos apresentados ao longo de todo o trabalho. A linguagem escolhida para a implementação foi o *Haskell*, dado ser uma linguagem funcional e, consequentemente, aproxima-se por construção do formato dos conteúdos aqui presentes. Para além disso, esta linguagem agiliza a aplicação dos conceitos sem perder a robustez e performance nas operações que executa.

A implementação foi construída sob a forma de biblioteca o que, no caso específico da linguagem escolhida, consiste num conjunto de módulos, são eles:

LANGUAGES estruturas para as linguagens

UTILITIES predicados sobre as linguagens

CONVERTER conversões entre linguagens

COMPILER compilação de programas (tal como definido em Capítulo 3).

VM máquina virtual, onde se define a semântica para executar um programa.

PARSER permite ler um programa a partir de um ficheiro ou string.

O código segue em anexo e está também disponível em `https://github.com/tacf/saCPS`.

4.1 MÓDULOS

Encontram-se nesta secção detalhados os módulos que desenvolvemos. A saber:

LANGUAGES

Este módulo contém as estruturas que definem as linguagens. De notar a convenção usada para a nomeação dos construtores dos tipos de dados e funções. Notação esta onde recorremos ao mesmo nome com o acréscimo da letra “L” para distinguir funções que trabalham sobre λ-termos. Por exemplo o construtor `INV` representa uma invocação na linguagem $\mathcal{LA}$, enquanto que, `LINV` representa uma invocação na linguagem $\mathcal{CPS}$.

```
type Label = String
```

O tipo `Label` é usado para representar nomes de variáveis em ambas as linguagens. Como se pode observar no comando acima é apenas açúcar sintático para o tipo `String`.

```
data E = VAR Label | CNST Int

data B = ID Label  | LCNST Int
```

Estas duas estruturas de dados representam variáveis para $\mathcal{LA}$ e $\mathcal{CPS}$, respectivamente, através dos construtores `VAR` e `ID`, e constantes, usando os construtores `CNST` e `LCNST`.

```
data S = ATRB Label E S
       | INV Label Label [E] S
       | IF E S S
       | RET E

data M  = BNDR Label B M
        | LINV Label Label [B] M
        | LIF B M M
        | LRET B
```

As estruturas `S` e `M` representam os corpos de procedimentos para $\mathcal{LA}$ e $\mathcal{CPS}$, respectivamente. Na linguagem $\mathcal{LA}$, `ATRB` representa um comando de atribuição, `INV` um comando de invocação e `RET` um comando de retorno. Na linguagem $\mathcal{CPS}$, recorremos às mesmas denominações com exceção do construtor `BNDR` que representa um "binder"(ou atribuição).

```
data Proc = PROC Label [Label] S

data LProc = LPROC Label [Label] M
```

As estruturas `Proc` e `LProc` representam procedimentos. Os parâmetros dos construtores `PROC` e `LPROC` são idênticos e representam, por ordem, o nome do procedimento, a lista de parâmetros formais e o seu corpo.

```
type Program = [Proc]

type LProgram = [LProc]
```

Os tipos de dados acima, são apenas açúcar sintático para descrevermos um *Programa* como uma lista de procedimentos.

As estruturas que enunciamos foram extraídas das definições apresentadas neste trabalho no que concerne às gramáticas das linguagens.

De notar que, dada a bijecção sintática entre as linguagens, existe a possibilidade de reduzir o código aqui da implementação, utilizando apenas uma estrutura de dados para ambas as linguagens, removendo assim, a necessidade de definir as conversões e a criação de um parser distinto para cada uma das linguagens. A opção de manter os

conceitos em separado deveu-se à opção de manter a implementação o mais fiel possível aos conceitos descritos.

UTILITIES

Este módulo introduz algumas funções que que consistem em predicados sobre a linguagem, sendo as mais relevantes as funções `isSAProc`, `isWF` e `isLWF`, que verificam, respetivamente, se um procedimento se encontra no formato *SA*, se um programa $\mathcal{LA}$ ou $\mathcal{CPS}$ é bem formado.

Neste módulo encontram-se também definidas algumas funções auxiliares que nos permitem trabalhar com as estruturas definidas no módulo *Languages*. Por exemplo, a função `getProcBody` permite, dado um procedimento em $\mathcal{LA}$, obter o corpo desse mesmo procedimento (ou seja, um elemento `S` da estrutura da gramática).

CONVERTER

Converter, como o próprio nome indica, define funções que permitem converter entre linguagens. A funções `la2cps` e `cps2la` correspondem às definições 46 e 45, respetivamente.

COMPILER

Este módulo define duas funções principais - `compile` e `compileL` - que são nada mais que a implementação dos conceitos de expansão apresentados no capítulo anterior e formalmente representados pelas definições 37 e 44, respetivamente.

VM

VM é a abreviatura de *Virtual Machine*, módulo no qual estão definidos os aspetos semânticos das linguagens. Este exporta duas funções: `runProgram` e `runLProgram` - sendo que ambas recebem um corpo de procedimento (`S` ou `M`, respetivamente) e um estado ou fecho, respetivamente.

PARSER

Este módulo define parsers para as linguagens apresentadas nos capítulos anteriores. Os detalhes sobre a sintaxe escolhida e sobre a utilização encontram-se expressos nos exemplos da secção seguinte.

Note-se que o parser para $\mathcal{CPS}$ tem algumas restrições no tratamento de espaços em branco e quebras de linha. No entanto seguindo os exemplos disponíveis é possível criar novos programas.

4.2 EXEMPLOS

Os exemplos que se seguem assumem que se está a utilizar o *ghci* (interpretador do Haskell) e que os respetivos módulos foram carregados e que a pasta atual é a *src* (onde se encontro os módulos).

Os exemplos foram também disponibilizados online. No repositório, optou-se por se disponibilizar todos os exemplos em ambas as linguagens, a linguagem utilizada em cada ficheiro está patente na extensão do mesmo, sendo que essa extensão não influência o resultado final. Por exemplo, o ficheiro "program1.la" apresenta, na linguagem $\mathcal{LA}$, mesmo programa que o ficheiro "program1.cps", apresenta na linguagem $\mathcal{CPS}$.

Exemplo 1. Conversão de um programa na linguagem $\mathcal{LA}$ para a linguagem $\mathcal{CPS}$.

```
f := proc(x,y){
    if(x) {
        ret(y)
    } else {
        x:=1;
        if(y){
            ret(x)
        } else {
            y:=x;
            ret(y)
        }
    }
}
```

Figura 25: Ficheiro "examples/program1.la"

```
*Parser>fmap (map la2cps) (readFromFile parseProgram "../examples
    /program1.la")
```

A utilização de "fmap (map la2cps)" deve-se ao facto de estarmos a trabalhar directamente com o resultado da função *parseFromFile* que é *IO Program*.

Alternativamente podemos utilizar os seguintes comandos:

```
*Parser>y <- readFromFile parseProgram "../examples/program1.la"
*Parser>maps la2cps y
```

O resultado deste comando - o programa "examples/program1.cps", é o seguinte:

```
[f:=\proc x,y).(if \k.kx ((\k.k(\k.ky))) ((\k.k1)(\x.(if \k.ky
    ((\k.k(\k.kx))) ((\k.kx)(\y.(\k.k(\k.ky)))))))
]
```

Figura 26: Conversão de "examples/program1.la" para $\mathcal{CPS}$

Exemplo 2. Execução do ficheiro "examples/program1.la" , através dos comandos que se seguem:

```
*Main> y <- readFromFile parseProgram "../examples/program1.la"
*Main> runProgram ((getProcBody.head) y, [])
```

Note-se que neste caso o "Main" refere-se ao conjunto de todos os módulos, ou seja, é apenas um módulo que importa todos os outros.

A utilização da função "head" deve-se ao facto de o conjunto de procedimentos ser representado por uma lista. Neste caso, sabemos que a lista tem apenas um elemento.

Efetuar estas operações representa a aplicação das regras de semântica para a linguagem $\mathcal{LA}$. Note-se que neste caso estamos a ignorar os parâmetros de entrada, começando com um estado "vazio". Segue então o resultado da aplicação das operações:

```
###Command###
x:=1;
if(y){
ret(x)
}else{
y:=x;
ret(y)
}
####State####
[]
#############
---------------------
###Command###
if(y){
ret(x)
}else{
y:=x;
ret(y)
}
####State####
[("x",1)]
#############
---------------------
```

```
###Command###
y:=x;
ret(y)
####State####
[("x",1)]
#############
--------------------
###Command###
ret(y)
####State####
[("x",1),("y",1)]
#############
--------------------
Computation Ended
```

Exemplo 3. Verificações de boa formação e do predicado sa.

Considere-se o seguinte programa presente no ficheiro "examples/-program2.la":

```
f := proc(x,y){
    if(x) {
        ret(y)
    } else {
        x:=g(2);
        if(y){
            ret(x)
        } else {
            y:=x;
            ret(y)
        }
    }
}

g := proc(w){
    z := w;
    z := 2;
    ret(z)
}
```

Figura 27: Ficheiro "examples/program2.la"

Para verificar a boa formação deste programa é necessário executar os seguintes comandos:

```
*Main> y <- readFromFile parseProgram "../examples/program2.la"
```

```
*Main> isWF y
True
```

O resultado do último comando permite-nos concluir que o programa é bem formado. Poderemos também verificar se o programa pertence à linguagem $\mathcal{SA}$ da seguinte forma:

```
*Main> y <- readFromFile parseProgram "../examples/program2.la"
*Main> map isSAProc y
[True,False]
```

O resultado permite-nos concluir que o procedimento `f` pertence à linguagem $\mathcal{SA}$, o que não acontece com procedimento `g`. É importante ressalvar que apenas falamos da verificação individual dos procedimentos. Para considerarmos o programa como estando no formato $\mathcal{SA}$ teríamos que selecionar em primeiro lugar, qual das restrições (cf. Capítulo 3) que gostaríamos que o programa obedecesse.

Exemplo 4. Compilação/expansão de um programa $\mathcal{LA}$

```
f := proc(x,y){
    if(x) {
        ret(y)
    } else {
        x := g(2);
        if(y){
            x := 3;
            ret(x)
        } else {
            x := y;
            ret(x)
        }
    }
}

g := proc(w){
    z := w;
    z := 2;
    ret(z)
}
```

Figura 28: Ficheiro "examples/program3.la"

Executando os comandos:

```
*Main> y <- readFromFile parseProgram "../examples/program3.la"
*Main> compile y
```

O resultado é o seguinte:

```
[f:=proc(x,y){
if(x){
ret(y)
}else{
w:=2;
z:=w;
z:=2;
x:=z;
if(y){
x:=3;
ret(x)
}else{
x:=y;
ret(x)
}
}
}
,g:=proc(w){
z:=w;
z:=2;
ret(z)
}
]
```

5 CONCLUSÃO

O paradigmas imperativo e funcional, aparentemente muito distantes, podem ser relacionados através da correspondência entre programas *SA* e *CPS*. Esta relação foi observada no início dos anos 90 e permite transferir ideias entre estes dois paradigmas. Contudo, a literatura acerca deste tema parece ser escassa. Apesar de, à partida, a relação entre os estilos *SA* e *CPS* aparentar simplicidade, esta impõe algumas escolhas e alguns compromissos para poder ser concretizada. Nesta dissertação concretizamos esta relação estabelecendo um isomorfismo entre linguagens correspondentes a um fragmento da linguagem *SA* considerada em [13] e um fragmento de λ-calculus no formato *CPS*.

Este trabalho, inspirado no de outros autores ([16], [13]), compreende a construção das linguagens, a definição dos diversos conceitos que lhes estão associados e os resultados da relação entre essas mesmas linguagens, culminando assim numa análise que prima pela originalidade.

Uma diferença importante em relação ao trabalho apresentado em [13] é o facto do presente trabalho para além de integrar a correspondência ao nível de programas, contempla também a relação a nível relação ao nível de execução de programas em linguagens *LA* (entenda-se, do mesmo modo, *SA*) e *CPS*.

O estudo aqui apresentado prima pela maneira como é detalhada a relação. No entanto, alguns destes detalhes não surgem de uma forma tão simples como poderia ser inicialmente esperado. O próprio conceito de *fecho* em si não é o conceito mais usual quando queremos atribuir semântica ao λ-calculus, usualmente recorre-se ao conceito mais simples de β-redução para tal. Porém, esta escolha justifica-se pelo facto de os modelos computacionais das máquinas físicas imporem a utilização de uma estrutura externa para armazenamento de valores (conceito de estado), tal como apontado em [1]. Neste sentido, o conceito de fecho é necessário e pertinente na medida em que aproxima a semântica do λ-calculus ao modelo de memória.

Todos estes aspetos particulares, permitem atentar e refletir acerca das dificuldades que surgem e dos cuidados que são necessários quando se pretende detalhar uma relação, tal como aqui foi realizado. Um dos cuidados a referir é o de que todas as definições (em particular apontamos para as de compilação e de boa formação) foram construídas de forma a assentarem em esquemas de "boa recursão". As dificuldades inerentes a este tipo de cuidado são refletidas, por

exemplo, na necessidade de utilização de formas de indução mais complexas, como a apresentada para a função `wf`.

Uma contribuição particularmente relevante deste trabalho prende-se com o facto de relacionar α-equivalência e o formato *SA* estático. Este aspeto revela que a relação aqui estabelecida facilita o transporte de diversas análises entre os dois universos. Neste caso, em particular, permitiu transportar o conceito de α-equivalência para o mundo imperativo e, no mundo funcional, observar a possibilidade de gerar termos α-equivalentes utilizando as técnicas de conversão usuais na transformação de programas imperativos para o formato *SA*.

Uma outra contribuição que não podemos deixar de mencionar é a implementação dos conceitos envolvidos nesta dissertação. Apesar desta implementação poder ser extraída com relativa facilidade, em parte devido o cuidado adicional que se colocou na criação desses mesmos conceitos, a implementação permite consolidar, testar e animar muitas das ideias aqui presentes, facilitando a percepção dos resultados e a obtenção de algumas conclusões. Em suma, a ferramenta, apesar de simples, constituiu um elemento fundamental para a construção do isomorfismo *SA*-*CPS*.

Comparativamente a outras linguagens, aquelas aqui apresentadas têm claras limitações e, como foi apresentado no Capítulo 3, algumas dessas limitações foram introduzidas de forma a ser possível recorrer às propriedades apresentadas no Capítulo 2. O âmbito deste estudo apenas permitiu observar superficialmente estes dois universos tão vastos que são os paradigmas imperativo e funcional.

Assim, seria importante que trabalhos futuros visassem a extensão de ambas as linguagens para, por exemplo, permitir ciclos e/ou desenvolver uma semântica para procedimentos permitindo recursividade direta e indireta. No caso dos ciclos já apontamos o estudo realizado por Kelsey que apresenta algumas dificuldades e eventuais soluções para lidar com esse tipo de construções.

Um outro aspecto que fica por analisar no nosso estudo são as conversões das linguagens "primitivas" para as intermédias. Por exemplo, seria interessante estudar, até que ponto, traduções conhecidas de $\mathcal{LA}$ para *SA* e de λ-calculus para *CPS* podem ser combinadas com as traduções que estudamos entre *SA* e *CPS* e se tais combinações trazem alguma ideia inovadora.

Uma outra linha de investigação que poderia ser seguida diz respeito à formalização dos conceitos aqui patentes com recurso a ferramentas computacionais como o `Coq` (Bertot and Castéran [5]). Tal como vimos através da implementação, todos os conceitos envolvidos no nosso trabalho podem ser facilmente implementados e seria relevante sermos capazes de utilizar tais ferramentas para reforçar e validar os resultados aqui apresentados.

Deste modo, apesar das limitações enumeradas e tendo em consideração as questões passíveis de se investigar posteriormente, as con-

tribuições do presente estudo constituem uma perspectiva diferente sobre a relação entre os formatos *SA* e *CPS*, possibilitando uma nova direção na análise desta relação.

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

[2] Andrew W. Appel. Ssa is functional programming. *ACM SIGPLAN NOTICES*, 33(4):17–20, 1998.

[3] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984. http://www.cs.ru.nl/ henk/Personal Webpage.

[4] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, March 2014. ISSN 0164-0925. doi: 10.1145/2579080. URL http://doi.acm.org/10.1145/2579080.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.

[6] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, May 2003. ISSN 0004-5411. doi: 10.1145/765568.765573. URL http://doi.acm.org/10.1145/765568.765573.

[7] Alonzo Church. A Set of Postulates for the Foundation of Logic. URL http://www.jstor.org/stable/1968337.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75280. URL http://doi.acm.org/10.1145/75277.75280.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[10] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[11] Michael J. Fischer. Lambda-calculus schemata, 1993.

[12] Timothy G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96714. URL http://doi.acm.org/10.1145/96709.96714.

[13] Richard Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, pages 13–22. ACM Press, 1995.

[14] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975. ISSN 03043975. doi: 10.1016/0304-3975(75)90017-1. URL http://dx.doi.org/10.1016/0304-3975(75)90017-1.

[15] JohnC. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3-4):233–247, 1993. ISSN 0892-4635. doi: 10.1007/BF01019459. URL http://dx.doi.org/10.1007/BF01019459.

[16] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Lisp and Symbolic Computation*, pages 288–298.

[17] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.

[18] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

[19] Peter Vanbroekhoven. Dynamic single assignment. ESSES PhD Student Meeting at the 9th International Static Analysis Symposium, Madrid, Spain, September 15-21, 2002, 2002. URL https://lirias.kuleuven.be/handle/123456789/167099.

[20] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4), September 2007. ISSN 1084-4309. doi: 10.1145/1278349.1278353. URL http://doi.acm.org/10.1145/1278349.1278353.

ANEXOS

CÓDIGO DA IMPLEMENTAÇÃO

A.1 MÓDULO DE LINGUAGENS

```
{-
 Module Languages

 Description: module that defines language structures
-}
module Languages where

type Label = String

-- Imperative expressions
data E = VAR Label | CNST Int
-- Functional expressions
data B = ID Label  | LCNST Int

-- Imperative comand sequences
data S  = ATRB Label E S
        | INV Label Label [E] S
        | IF E S S
        | RET E

-- Functional command sequences
data M  = BNDR Label B M
        | LINV Label Label [B] M
        | LIF B M M
        | LRET B

-- Imperative Procedure
data Proc  = PROC Label [Label] S
-- Functional Procedure
data LProc = LPROC Label [Label] M

-- Imperative Program
type Program = [Proc]
-- Functional Program
type LProgram = [LProc]
```

```
instance Show E where
        show e = case e of
                VAR x    -> x
                CNST c   -> show c

instance Show S where
        show s = case s of
            RET e -> list2string
                     ["ret(",show e,")"]
            ATRB x e s -> list2string
                          [x,":=",show e,";\n"
                          ,show s]
            INV x f vec s -> list2string
                             [x,":=",f,"("
                             ,printVector show vec,");\n"
                             ,show s]
            IF e s1 s2  -> list2string
                           ["if(",show e,"){\n"
                           ,show s1
                           ,"\n}else{\n"
                           ,show s2
                           ,"\n}"]

instance Show Proc where
show (PROC l vec s) = list2string
                      [l,":=proc("
                      ,printVector id vec,"){\n"
                      ,show s
                      ,"\n}\n"]


instance Show B where
        show b = case b of
                ID x     -> list2string ["\\k.k", x]
                LCNST c  -> list2string ["\\k.k", show c]

instance Show M where
        show m = case m of
                LRET b -> list2string
                          ["(\\k.k(",show b,"))"]
                BNDR x b m -> list2string
                              ["(",show b
                              ,")(\\"
                              ,x,"."
                              ,show m,")"]
                LINV x f vec m -> list2string
```

```
                                   ["(",f,"",printVector show vec," )("
                                   ,show m
                                   ,")"]
                LIF b m1 m2 -> list2string
                                 ["(if ",show b
                                 ," (",show m1,")"
                                 ," (",show m2,"))"]

instance Show LProc where
    show (LPROC l vec m) = list2string
                             [l,":=\\proc "
                             , printVector id vec,")."
                             ,show m
                             ,"\n"]

printVector:: (Show a) => (a -> String) -> [a] -> String
printVector f e = case e of
          []    -> ""
          e1:[] -> f e1
          _ -> (foldr (++) ((f.last) e)
                    (map ((++",").f)
                        ((reverse.tail.reverse) e)))

list2string :: [String] -> String
list2string = foldr (++) ""
```

A.2 MÓDULO DE FERRAMENTAS

```
{-
 Module Utilities

 Description: module  the define tools and predicates over
              the languages
-}
module Utilities (
            -- LA Functions
            isSAProc,
            isWF,
            getProcBody,
            getProgramRefList,
            getProcRef,
            getProcPara,
            findProc,
            -- CPS Functions
            isLWF,
            getLProcBody,
```

```haskell
            getLProgramRefList,
            getLProcRef,
            getLProcPara,
            findLProc) where

import Languages
import Data.List

-- LA Language Functions


-- Indicates whether a LA procedure is in Single Assignment or not
isSAProc    :: Proc -> Bool
isSAProc p = snd (isSAaux [] (getProcBody p))

-- Auxiliar function for isSAproc
isSAaux :: [Label] -> S -> ([Label],Bool)
isSAaux l (RET _) = (l,True)
isSAaux l (ATRB x _ s) | (elem x l) = (l,False)
                       | otherwise  = isSAaux (x:l) s
isSAaux l (INV x _ _ s)| (elem x l) = (l,False)
                       | otherwise  = isSAaux (x:l) s
isSAaux l (IF _ s1 s2) =
                        let (l1, b1) = isSAaux l s1
                            (l2, b2) = isSAaux l1 s2
                        in  (l2, b1 && b2 )

-- Indicates whether a LA program is well formed or not
isWF :: Program -> Bool
isWF pr =
  and (map (\p -> isWFProc [(getProcRef p)] pr (getProcBody p)) pr)

-- Indicates whether a LA procedure is well formed or not
isWFProc:: [Label] -> Program -> S -> Bool
isWFProc _ _ (RET _)                 = True
isWFProc l p (ATRB _ _ s)     = isWFProc l p s
isWFProc l p (INV x f vec s)
        | (elem f l) = False  -- Direct/Indirect recursion found
        | otherwise =
        case (findProc p f) of
            Nothing -> False --Inexistent Procedure invoked
            (Just pr) ->
                let (r,b) = (getProcRef pr, getProcBody pr)
                in        (isWFProc l p s) && (isWFProc (r:l) p b)
isWFProc l p (IF _ s1 s2) =
        (isWFProc l p s1) && (isWFProc l p s2)
```

```haskell
-- Indicates whether a CPS program is well formed or not
isLWF :: LProgram -> Bool
isLWF pr =
    and (map (\p -> isLWFProc [(getLProcRef p)] pr (getLProcBody p)) pr)

-- Indicates whether a CPS procedure is well formed or not
isLWFProc:: [Label] -> LProgram -> M -> Bool
isLWFProc _ _ (LRET _)          = True
isLWFProc l p (BNDR _ _ m)      = isLWFProc l p m
isLWFProc l p (LINV x f vec m)
    | (elem f l) = False        -- Direct or Indirect recursion found
    | otherwise =
    case (findLProc p f) of
        Nothing -> False --Inexistent Procedure invoked
        (Just pr) ->
            let (r,b) = (getLProcRef pr, getLProcBody pr)
            in        (isLWFProc l p m) && (isLWFProc (r:l) p b)
isLWFProc l p (LIF _ m1 m2) =
            (isLWFProc l p m1) && (isLWFProc l p m2)

-- Finds if a LA procedure with a specific reference exists
findProc :: Program -> Label -> Maybe Proc
findProc [] lf = Nothing
findProc ((PROC l vec s):ps) lf | lf == l = Just (PROC l vec s)
                                | otherwise = findProc ps lf

-- Finds if a CPS procedure with a specific reference exists
findLProc :: LProgram -> Label -> Maybe LProc
findLProc [] lf = Nothing
findLProc ((LPROC l vec m):ps) lf | lf == l = Just (LPROC l vec m)
                                  | otherwise = findLProc ps lf


-- Returns a list with all the references from a program
getProgramRefList :: Program -> [Label]
getProgramRefList = map (\(PROC l _ _) -> l)

-- Returns a list with all the references from a program
getLProgramRefList :: LProgram -> [Label]
getLProgramRefList = map (\(LPROC l _ _) -> l)

-- Returns LA procedure parameters list
getProcPara :: Proc -> [Label]
getProcPara (PROC _ l _ ) = l
```

```
-- Returns CPS procedure parameters list
getLProcPara :: LProc -> [Label]
getLProcPara (LPROC _ l _ ) = l

-- Returns LA procedure body
getProcBody :: Proc -> S
getProcBody (PROC _ _ s) = s

-- Returns CPS procedure body
getLProcBody :: LProc -> M
getLProcBody (LPROC _ _ m) = m

-- Returns LA procedure reference
getProcRef :: Proc -> Label
getProcRef (PROC l _ _) = l

-- Returns CPS procedure reference
getLProcRef :: LProc -> Label
getLProcRef (LPROC l _ _) = l
```

A.3 MÓDULO DE CONVERSÕES

```
{-
 Module Converter

 Description: module that define language convertions
-}
module Converter(la2cps, cps2la) where

import Languages

-- LA -> CPS
la2cps  :: Proc -> LProc
la2cps (PROC l vec s) = (LPROC l vec (la2cpsS s))

la2cpsS :: S -> M
la2cpsS s = case s of
 (RET e)          -> LRET (la2cpsE e)
 (ATRB l e s)     -> BNDR l (la2cpsE e) (la2cpsS s)
 (INV l f vec s) -> LINV l f (map la2cpsE vec) (la2cpsS s)
 (IF e s1 s2)     -> LIF (la2cpsE e) (la2cpsS s1) (la2cpsS s2)

la2cpsE :: E -> B
la2cpsE e = case e of
 (VAR l)   -> ID l
 (CNST x) -> LCNST x
```

```
-- CPS -> LA
cps2la  :: LProc -> Proc
cps2la (LPROC l vec m) = PROC l vec (cps2laM m)

cps2laM :: M -> S
cps2laM m = case m of
 (LRET b)          -> RET (cps2laB b)
 (BNDR l b m)      -> ATRB l (cps2laB b) (cps2laM m)
 (LINV l f vec m) -> INV l f (map cps2laB vec) (cps2laM m)
 (LIF b m1 m2)     -> IF (cps2laB b) (cps2laM m1) (cps2laM m2)

cps2laB :: B -> E
cps2laB b = case b of
 (ID l)     -> VAR l
 (LCNST x) -> CNST x
```

A.4 MÓDULO DE COMPILAÇÃO

```
{-
  Module Compiler

  Description: Module that defines compiler functions
-}

module Compiler(compile, compileL) where

import Languages
import Utilities

-- Imperative Compiler
invPara :: [Label] -> [E] -> S -> S
invPara [] [] s = s
invPara [] (e:es) s = s -- Malformed invocation
invPara (l:ls) [] s = s -- Malformed invocation
invPara (l:ls) (e:es) s = (ATRB l e (invPara ls es s))

subs :: S -> Label -> S -> S
subs (RET e) x s           = ATRB x e s
subs (ATRB y e sp) x s     = ATRB y e (subs sp x s)
subs (INV y f vec sp) x s = INV y f vec (subs sp x s)
subs (IF e s1 s2) x s      = IF e (subs s1 x s) (subs s2 x s)

subsF :: Label -> S -> S -> [Label] -> S
subsF _ _ (RET e) v  = RET e
```

```
subsF f s (ATRB x e sp) v  = ATRB x e (subsF f s sp v)
subsF f s (INV x g vec sp) v
        | f == g =
           invPara v vec (subs s x (subsF f s sp v))
        | otherwise = INV x g vec (subsF f s sp v)
subsF f s (IF e s1 s2) v = IF e (subsF f s s1 v) (subsF f s s2 v)

compile :: Program -> Program
compile p = compileAux ( getProgramRefList p ) p

compileAux :: [Label] -> Program -> Program
compileAux [] p = p
compileAux (l:ls) p = let
        replaceIn (Just x) =
          map (\(PROC lp vec s)
           -> PROC lp vec (subsF l (getProcBody x) s (getProcPara x))) p
        replaceIn Nothing  = p
        in compileAux ls (replaceIn (findProc p l))



-- Functional Compiler
invLPara :: [Label] -> [B] -> M -> M
invLPara [] [] m         = m
invLPara [] (b:bs) m     = m -- Malformed invocation
invLPara (l:ls) [] m     = m -- Malformed invocation
invLPara (l:ls) (b:bs) m = (BNDR l b (invLPara ls bs m))

subsL :: M -> Label -> M -> M
subsL (LRET b) x m          = BNDR x b m
subsL (BNDR y b mp) x m     = BNDR y b (subsL mp x m)
subsL (LINV y f vec mp) x m  = LINV y f vec (subsL mp x m)
subsL (LIF b m1 m2) x m     = LIF b (subsL m1 x m) (subsL m2 x m)

subsLF :: Label -> M -> M -> [Label] -> M
subsLF _ _ (LRET b) v          = LRET b
subsLF f m (BNDR x b mp) v     = BNDR x b (subsLF f m mp v)
subsLF f m (LINV x g vec mp) v
        | f == g =
          invLPara v vec (subsL m x (subsLF f m mp v))
        | otherwise = LINV x g vec (subsLF f m mp v)
subsLF f m (LIF b m1 m2) v = LIF b (subsLF f m m1 v) (subsLF f m m2 v)

compileL:: LProgram -> LProgram
compileL p = compileLAux ( getLProgramRefList p ) p
```

```
compileLAux :: [Label] -> LProgram -> LProgram
compileLAux [] p = p
compileLAux (l:ls) p = let
        replaceIn (Just x) =
         map (\(LPROC lp vec s) ->
          LPROC lp vec (subsLF l (getLProcBody x) s (getLProcPara x))) p
        replaceIn Nothing  = compileLAux ls p
        in compileLAux ls (replaceIn (findLProc p l))
```

A.5 MÓDULO DE EXECUÇÃO

```
module VM (
                -- Imperative Execution
        runProgram,
                -- Functional Execution
        runLProgram
                ) where

import Data.List

import Languages


type Assoc = [(Label,Int)]

type State   = Assoc
type Closure = Assoc

type Config = (S,State)
type LConfig = (M,Closure)

evalInState :: E -> State -> Int
evalInState (CNST c) _ = c
evalInState (VAR l) s = case (find ((==l).fst)  s) of
                             Nothing -> 0
                             (Just x) -> snd x

evalInClosure :: B -> Closure -> Maybe Int
evalInClosure (LCNST c) _ = Just c
evalInClosure (ID l) c = fmap snd (find ((==l).fst) c)


updateState :: Label -> Int -> State -> State
updateState  = updateAssoc

updateClosure :: Label -> Int -> Closure -> Closure
```

```
updateClosure = updateAssoc


updateAssoc :: Label -> Int -> Assoc -> Assoc
updateAssoc l c [] = [(l,c)]
updateAssoc l c ((x,v):xs) | x==l = (x,c):xs
                           | otherwise = (x,v):(updateAssoc l c xs)



isFinalConfig :: Config -> Bool
isFinalConfig (RET _, _) = True
isFinalConfig (_,_) = False

isFinalLConfig :: LConfig -> Bool
isFinalLConfig (LRET _, _) = True
isFinalLConfig (_,_) = False

step :: Config -> Config
step (RET e, st) = (RET e, st)
step (ATRB x e s, st) = (s,updateState x (evalInState e st) st)
-- step (INV x l vec s, st) = no specific semantics
step (IF e s1 s2, st) = case (evalInState e st) of
                            0 -> (s2, st)
                            _ -> (s1, st)



stepL :: LConfig -> LConfig
stepL (LRET b, cl) = (LRET b, cl)
stepL (BNDR x b m, cl) =
            case evalInClosure b cl of
                (Just b1) -> (m,updateClosure x b1 cl)
                -- Special Config representing error in closure
                Nothing -> (LRET (ID "error"),[])
-- stepL (LINV x l vec m, cl) = no specific semantics
stepL (LIF b m1 m2, cl) =
            case (evalInClosure b cl) of
                (Just 0) -> (m2, cl)
                (Just x) -> (m1, cl)
                Nothing -> (LRET (ID "error"), [])

runProgram :: Config -> IO ()
runProgram c = do
                let x = step c
                putStrLn ("###Command###\n"
                         ++show (fst x)
                         ++"\n####State####\n"
```

```
                 ++show (snd x)
                 ++"\n############\n"
                 ++"---------------------")
        if (isFinalConfig x)
        then putStrLn "Computation Ended"
        else runProgram x

runLProgram :: LConfig -> IO ()
runLProgram c = do
        let x = stepL c
        putStrLn ("###Command###\n"
                 ++show (fst x)
                 ++"\n####Closure####\n"
                 ++show (snd x)
                 ++"\n############\n"
                 ++"---------------------")
        if (isFinalLConfig x)
        then putStrLn "Computation Ended"
        else runLProgram x
```