Universidade de Aveiro Departamento de Matemática

2009

Alexandra Isabel Fernandes Espinha Abreu

Fundamentos Matemáticos da Programação Funcional

Universidade de Aveiro Departamento de Matemática
2009

Alexandra Isabel Fernandes Espinha Abreu

Fundamentos Matemáticos da Programação Funcional

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Matemática, realizada sob a orientação científica do Dr. Dirk Hofmann, Professor Auxiliar Convidado do Departamento de Matemática da Universidade de Aveiro

o júri

presidente

Prof. Dr. António Manuel Rosa Pereira Caetano
professor associado com agregação da Universidade de Aveiro

Prof. Dr. Gonçalo Gutierres da Conceição
professor auxiliar da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Prof. Dr. Dirk Hofmann
professor auxiliar da Universidade de Aveiro

palavras-chave

programação funcional, cálculo-λ, semântica denotacional, semântica operacional, LCF, PCF

resumo

O objectivo desta tese é estudar os contributos matemáticos que estiveram na origem da, agora tão falada, programação funcional.
Este trabalho está organizado em duas fases. Numa primeira fase irá estudar-se o cálculo-λ como sendo a primeira linguagem de programação funcional, especialmente no que diz respeito à notação utilizada por ela, e a sua contribuição nas ciências da computação. Na segunda fase irá estudar-se a linguagem de programação de funções computáveis, PCF, inicialmente desenvolvida por Dana Scott e posteriormente estudada por Gordon Plotkin. A sua sintaxe e outras propriedades serão alvo de estudo neste trabalho.

keywords

functional programming, λ-calculus, denotational semantics, operacional semantics, LCF, PCF

abstract

The aim of this thesis is to study the mathematical contributions that resulted in the functional programming.
This work is organized in two phases. Initially it will examine the λ-calculus as the first functional programming language, especially in what concerns to the notation used by it and its contribution in the Computer Science. In the second phase it will study the programming language of computable functionals, PCF, originally developed by Dana Scott and later studied by Gordon Plotkin. Its syntax and other properties will be subject of study in this work.

Conteúdo

Capítulo I - Introdução 1

1.1. A linguagem de programação PCF 2

1.2. Sintaxe de PCF 3

Capítulo II - Programação Funcional e Cálculo-λ 5

2.1. Expressões em cálculo-λ 5

2.2. Funções em cálculo-λ 6

2.3. Redução 8

2.4. Funções de ordem superior 11

2.5. Iteração e recursão no cálculo-λ 11

2.6. O cálculo-λ com tipos 13

2.7. Normalização forte 16

Capítulo III - Teoria de Domínios 23

3.1. Teoria de ordem 23

3.2. Conjuntos direccionados 23

3.3. Funções monótonas 24

3.4. Funções contínuas 25

3.5. Princípios de Indução para Pontos Fixos Mínimos 26

Capítulo IV - A Linguagem de Programação PCF 29

4.1. Os termos de PCF 29

4.2. Semântica operacional de PCF 31

4.3. Equivalência observacional 32

4.4. Semântica denotacional 34

4.5. Adequação Computacional 36

4.6. Computabilidade 40

4.7. Abstracção Completa para PCF 42

Bibliografia 49

Capítulo I

Introdução

Os primeiros computadores foram construídos nos anos 40. Naquela altura as linguagens de programação reflectiam a arquitectura do computador. Um computador consistia numa unidade central de processamento e numa memória e, portanto, um programa consistia em instruções que modificavam a memória e que eram executadas pela unidade de processamento. E é assim que surge o estilo de programação imperativa (como PASCAL, FORTRAN e C). A base teórica da programação imperativa foi fundada nos anos 30 por Alan Turing (na Inglaterra) e Jonh von Neuman (nos EUA).

No inicio dos anos 50, as funções, que sempre tiveram um papel importante na Matemática, começam a ser vistas como um bom modo de especificar um cálculo. As funções expressam a ligação entre os parâmetros ou valores de entrada (o *input*) e o resultado (o *output*) de determinados processos. Esta é a base do estilo de programação funcional, que utiliza uma linguagem mais próxima do "mundo humano" do que do "mundo do computador". Assim, um programa é escrito como uma função que é definida à custa de outras funções, que, por sua vez, são definidas em termos de ainda mais funções.

O *cálculo lambda* pode ser considerado como a primeira linguagem de programação funcional, embora nunca tenha sido projectada para ser realmente executada num computador. É um modelo de computação que foi desenvolvido por Alonzo Church nos anos 30 e que oferece um modo muito formal de descrever o cálculo de uma função.

A primeira linguagem de programação funcional criada para computadores foi o *LISP*, desenvolvida por John McCarthy no Instituto de Tecnologia de Massachusetts (MIT) no fim dos anos 50. Mesmo não sendo uma linguagem de programação puramente funcional, o *LISP* introduziu a maioria das características hoje encontradas nas modernas linguagens de programação funcional. *Scheme* foi uma tentativa posterior de simplificar e melhorar o *LISP*. Nos anos 70 a linguagem *ML* foi criada pela Universidade de Edimburgo,

e David Turner desenvolveu a linguagem *Miranda* na Universidade de Kent. A linguagem *Haskell* foi lançada no fim dos anos 80 numa tentativa de juntar ideias resultantes da pesquisa em programação funcional.

A programação funcional trata a computação como uma avaliação de funções matemáticas que evita estados ou dados mutáveis. A definição de uma função descreve como esta será avaliada em termos de outras funções, devendo a linguagem oferecer funções básicas que não requeiram definições adicionais.

Na programação funcional parecem faltar diversas construções frequentemente consideradas essenciais nas linguagens imperativas. Por exemplo, numa programação estritamente funcional, não há alocação explícita de memória, nem declaração explícita de variáveis. No entanto, essas operações podem ocorrer automaticamente quando a função é invocada. A alocação de memória cria espaço para os parâmetros e para o resultado e a declaração ocorre aquando da cópia dos parâmetros para este espaço recém-alocado e, eventualmente, da cópia do resultado como parâmetro de outra função. Nos programas funcionais não existem expressões de atribuição e, por isso, o valor de uma variável nunca irá mudar. Deste modo, evitam-se efeitos colaterais e, por isso, esta é uma linguagem que oferece transparência referencial. Isso assegura que o resultado de uma função é sempre o mesmo, para um dado conjunto de parâmetros, não importando onde ou quando seja avaliada.

Nas linguagens funcionais não existe noção de estado ou comandos que alteram estados. Os seus conceitos básicos são: a aplicação de uma função a um argumento e a definição de funções explicitamente e recursivamente. Estas linguagens permitem, não só, a definição recursiva de funções, mas também a definição recursiva de tipos de dados. Nas linguagens imperativas, a definição recursiva de tipos tem que ser feita à custa de ponteiros o que é uma tarefa delicada e uma fonte de erros subtis difíceis de eliminar.

1.1. A linguagem de programação PCF

Em 1969, Dana Scott apresentou uma lógica para funções computáveis, mais tarde conhecida como LCF. Em 1977, Gordon Plotkin publicou um texto no qual considerava os

termos da lógica de Scott como uma linguagem de programação chamada PCF – (*Programming Computable Functionals* - Linguagem de programação de funções computáveis).

PCF pode ser vista como uma versão simplificada de linguagens de programação funcional modernas como Haskell, Miranda e ML.

1.2. Sintaxe de PCF

Gordon Plotkin apresentou PCF explicitamente como uma linguagem de programação e estudou a relação entre as suas semânticas operacional e denotacional.

A sintaxe de PCF é essencialmente idêntica à do cálculo lambda com tipos, mas ampliada com uma noção de recursão, na forma de um operador do ponto fixo, e com alguns operadores aritméticos básicos.

Sumariamente tem-se que, os tipos mais simples, tipos-básicos, são os booleanos e os números naturais. Os tipos mais complexos obtêm-se do seguinte modo: se σ e τ são tipos, então é possível obter um novo tipo $(\sigma \to \tau)$ onde $\to$ é um operador binário associativo à direita.

A primeira ligação entre a semântica operacional e a semântica denotacional de PCF é a relação entre o comportamento de um programa e a natureza da sua denotação. Um programa será um termo de tipo-básico e o seu comportamento será dado por uma *função (relação) de avaliação* que especifica se o programa termina e qual o seu valor caso termine. Esse valor será o especificado pela sua denotação e, quando um programa não termina, a sua denotação será $\perp$.

Numa semântica denotacional, o significado dos termos e frases de uma linguagem é definido por meio de uma função que associa a cada termo, um elemento de um domínio matemático adequado para a interpretação da linguagem. Esse elemento é chamado *denotação* ou *significado* do termo.

A denotação de um termo que possui variáveis livres depende do significado atribuído a essas variáveis. Por isso, a função semântica é parametrizada por outra função, denominada *ambiente*, que associa a cada variável um elemento do domínio.

A segunda relação entre as duas semânticas diz respeito à *equivalência denotacional*. Dois termos são denotacionalmente equivalentes se e só se, em qualquer ambiente, têm a mesma denotação. Por outro lado, dois termos são *operacionalmente equivalentes* se puderem ser substituídos, um pelo outro, num programa sem alterarem o comportamento deste.

Capítulo II

Programação Funcional e Cálculo-λ

O nosso sistema numérico actual foi introduzido na Europa durante o Renascimento (por volta do ano 1200) por Fibonacci, entre outros matemáticos.

Uma notação para expressões e equações só seria desenvolvida durante o século XVI por François Viète, quando este começou a usar espaços para representar parâmetros e abreviações para as operações aritméticas.

Só ao fim de 250 anos se desenvolveu uma notação para funções arbitrárias. Foi Alonzo Church que, na década de 30, introduziu tal notação que é chamada de cálculo-lambda. O cálculo-λ é um sistema matemático formal que investiga funções e a aplicação de funções.

Nos anos 60, o cálculo-λ foi redescoberto como uma ferramenta versátil em Ciências da Computação, por McCarthy, Strachey, Landin, Scott, entre outros. O cálculo lambda permitiu chegar à base de uma linguagem de programação funcional na qual cada objecto é encarado como uma função.

2.1. Expressões em cálculo-λ

Existem três tipos de expressões-λ ($M, N, M_1, M_2, \ldots$ designarão expressões-λ arbitrárias). Elas são:

Variáveis usualmente representadas pelas letras $x, y, z, u, v, \ldots$.

Aplicações $(M_1 M_2)$. Tal representa a aplicação da expressão M_1 a M_2.

Abstracções $(\lambda x.M)$. Tal representa a função que devolve o valor M quando dado o parâmetro formal x.

O uso de parênteses, ou melhor, a ausência destes, obedece a determinadas regras que aqui se apresentam na forma de **convenções sintácticas**:

C1 A aplicação prende mais fortemente que a abstracção, ou seja, $\lambda x.xy$ significa $\lambda x.(xy)$ e não $(\lambda x.x)y$.

C2 A aplicação associa à esquerda, ou seja, xyz denota $(xy)z$ e não $x(yz)$.

C3 A expressão $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.M$ representa $\lambda x_1.\left(\lambda x_2.\ldots(\lambda x_n.M)\right)$

Os operadores são escritos antes dos argumentos. Além disso, a função e o argumento são escritos lado a lado sem parênteses à volta do argumento. Portanto, em vez de "$sen\ (x)$", no cálculo-λ escreve-se simplesmente "$sen\ x$". Se uma função requer mais do que um argumento, então estes são escritos de forma alinhada logo após a função. Portanto, "$x + 3$" escreve-se "$+\ x\ 3$", e "x^2" escreve-se "$*\ x\ x$". Os parênteses só são usados para agrupar determinadas expressões. Por exemplo, normalmente escreveríamos "$sen\ (x) + 4$" e no cálculo-λ fica "$+\ (sen\ x)\ 4$".

2.2. Funções em cálculo-λ

A essência do cálculo-λ é o mecanismo da abstracção. A expressão $\lambda x.M$ é a forma geral que as funções têm neste sistema. Para especificar uma função temos que dizer qual é o seu parâmetro formal, neste caso x, e qual é o seu resultado, aqui será M.

Na Matemática é usual escrever funções como equações, $f(x) = 3x$, e por vezes como mapeamentos, $x \mapsto 3x$. A notação que é usada no cálculo-λ liberta-nos da necessidade de dar um nome à função. No exemplo considerado anteriormente, em cálculo-λ a expressão "$3x$" escreve-se "$*\ 3\ x$" e para torná-la numa função precede-se por "$\lambda x.$". Obtém-se assim: "$\lambda x.*\ 3\ x$". A letra grega λ alerta o leitor para o facto de a letra que se segue não fazer parte de uma expressão, sendo apenas um parâmetro formal da função. O ponto que aparece a seguir ao parâmetro introduz o corpo da função. Esta forma de representar o parâmetro formal não nos é de todo desconhecida. O mesmo acontece em expressões do tipo $\forall x\ \ldots$ e $\int \ldots\ dx$.

A questão que se coloca aqui é, como se associam os parâmetros actuais aos formais. Tal é feito através de aplicações, $(\lambda x.M)N$. Para calcular esta aplicação procede-se à substituição do parâmetro formal pelo actual no corpo da função, $M[N/x]$.

Por exemplo, a aplicação da função anterior ao valor 4 é escrita do seguinte modo: $(\lambda x.* \ 3 \ x) \ 4$.

Embora não seja estritamente necessário, será conveniente introduzir abreviações para termos-λ. Assim, se abreviarmos a nossa função para F:

$$F \stackrel{\text{def}}{=} \lambda x.* \ 3 \ x$$

podemos escrever $F \ 4$ em vez de $(\lambda x.* \ 3 \ x) \ 4$.

Suponhamos agora que o corpo de uma função consiste numa outra função como no seguinte exemplo:

$$N \stackrel{\text{def}}{=} \lambda y.(\lambda x.* \ y \ x)$$

Se aplicarmos esta função ao valor 3, então obtemos a função anterior $\lambda x.* \ 3 \ x$. Por outras palavras, N é uma função que, quando aplicada a um número, devolve outra função (ou seja, $N \ 3$ comporta-se como F). No entanto, também é possível interpretar N como sendo uma função com dois argumentos e, nesse caso, podemos tirar os parênteses ficando $\lambda y.\lambda x.* \ y \ x$. Se abreviarmos esta expressão obtemos ainda. $\lambda y \ x.* \ y \ x$.

No entanto, é necessário algum cuidado no que diz respeito às substituições. Antes de mais é preciso compreender o que são variáveis livres e ligadas.

Definição 2.1. A ocorrência de uma variável x numa expressão do tipo $\lambda x.M$ é **ligada**, todas as outras são **livres**. A ocorrência de x em $\lambda x.$ é a ocorrência **ligante** que introduz a variável, outras ocorrências são chamadas **aplicadas**.

Definição 2.2. O conjunto das **variáveis livres** de um termo M, denota-se por $\mathrm{FV}(M)$ e é definido por:

1. $\mathrm{FV}(x) = \{x\}$
2. $\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$
3. $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) - \{x\}$

Definição 2.3. Uma expressão diz-se **fechada** se não contém variáveis livres, caso contrário, diz-se **aberta**.

A mesma variável pode ocorrer livre e ligada na mesma expressão. Vejamos o seguinte exemplo:

$$(\lambda x.\lambda y.yx)((\lambda z.zx)x)$$

A primeira ocorrência aplicada de x é ligada, mas a segunda e terceira ocorrências aplicadas são livres.

Vejamos agora uma definição formal de substituição.

Definição 2.4. A **substituição** de N pelas ocorrências livres de x em M, denota-se por $M[N/x]$ e define-se do seguinte modo:

- $x[M/x] \stackrel{\text{def}}{=} M$ e para uma variável $y \neq x$, $y[M/x] \stackrel{\text{def}}{=} y$;
- No caso das aplicações, a substituição é feita em duas partes:
$$(M_1 M_2)[N/x] \stackrel{\text{def}}{=} (M_1[N/x]\ M_2[N/x]);$$
- Se $M \equiv \lambda x.N$ então $M[P/x] \stackrel{\text{def}}{=} M$.

  Se y é uma variável diferente de x, e $M \equiv \lambda y.N$ então,
  - Se y não ocorre livre em P, $M[P/x] \stackrel{\text{def}}{=} \lambda y.N[P/x]$,
  - Se y ocorre livre em P, $M[P/x] \stackrel{\text{def}}{=} \lambda z.(N[z/y][P/x])$, onde z é uma variável que não aparece em P ou N.
- Em geral, se x não é livre em M então $M[P/x] = M$.

2.3. Redução

A semântica operacional do cálculo-λ é usualmente dada por uma relação de redução, onde o significado de um termo é a sua forma normal, ou seja, um termo que não admite mais reduções.

Definição 2.5. Uma regra de redução escrita na forma $M \longrightarrow M'$, define que o termo M **avalia** ou **reduz** para o termo M', num passo. Nesse caso, dizemos que M é um **redex** e M' é o seu **reduto**.

No coração da relação de redução encontra-se a regra de **redução-β**, pela qual um termo $(\lambda x.M)N$ é reduzido a $M[N/x]$:

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

Vejamos dois exemplos:

$(\lambda x.* 3\ x)\ 4 \longrightarrow_\beta * 3\ 4$

$$(\lambda y.\, y\ 5)(\lambda x.{*}\ 3\ x)\ \longrightarrow_\beta\ (\lambda x.{*}\ 3\ x)\ 5\ \longrightarrow_\beta {*}\ 3\ 5$$

Definição 2.6. Considere-se que $M \longrightarrow M'$. Podemos definir as seguintes **regras de inferência**:

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]}\ \text{beta}$$

$$\frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'}\ \text{abst}$$

$$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}\ \text{ape}$$

$$\frac{N \longrightarrow N'}{MN \longrightarrow MN'}\ \text{apd}$$

A primeira regra, beta, é a regra de redução-β propriamente dita. As restantes três permitem-nos aplicar a redução-β dentro de qualquer termo. Note-se que na regra abst impusemos que a variável ligada se chamasse x em ambos os lados.

Vejamos, agora, um exemplo de dedução:

$$\frac{\dfrac{}{(\lambda x.\lambda y.x)z \longrightarrow \lambda y.z}\ \text{beta}}{(\lambda x.\lambda y.x)z\ z \longrightarrow (\lambda y.z)\ z}\ \text{ape}$$

Definição 2.7. Escreve-se que $\boldsymbol{M \longrightarrow^{*} N}$ se existe uma sequência de zero ou mais reduções tal que,

$$M \equiv M_0 \longrightarrow_\beta \cdots \longrightarrow_\beta M_n \equiv N$$

representa uma sequência de passos de redução e onde $n \geq 0$. Dizemos que N é um **reduto** de M.

Seria de esperar que um termo, ao fim de um determinado número de reduções, alcançasse uma forma onde não fosse possível reduzir mais. Surpreendentemente, isto nem sempre acontece. Segue-se o menor contra-exemplo:

$$\Omega \stackrel{\text{def}}{=} (\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

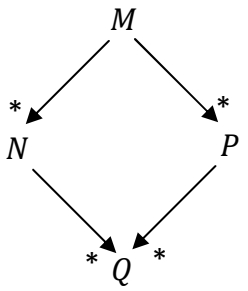
O termo Ω reduz sempre para ele próprio.

Definição 2.8. Se um termo for reduzido até a um ponto onde não são possíveis mais reduções, dizemos que o termo foi **reduzido à forma normal**. Tal como Ω, nem todos os termos têm uma forma normal.

Definição 2.9. Dizemos que M' é uma **forma normal** de M se $M \longrightarrow^* M'$ e M' está na forma normal.

Uma questão que aqui se coloca é a seguinte: diferentes sequências de redução dão origem ao mesmo resultado final? O seguinte teorema dá resposta a esta pergunta:

Teorema 2.1. (Church-Rosser) *Para todo M, N e P, se $M \longrightarrow^* N$ e $M \longrightarrow^* P$, então existe Q tal que $N \longrightarrow^* Q$ e $P \longrightarrow^* Q$.*



Demonstração. A demonstração completa deste teorema pode ser encontrada em [15][1]. Esta prova é feita por indução sobre a estrutura sintáctica dos termos, normalmente chamada de indução estrutural. O método de **indução estrutural** diz que, para provar o resultado $P(M)$ para toda a expressão-λ M, basta provar:

- $P(x)$ para todas as variáveis x.
- $P(MN)$ assumindo que $P(M)$ e $P(N)$ se verificam.
- $P(\lambda x.M)$ assumindo que $P(M)$ se verifica.

■

Corolário 2.1. *Se um termo-λ tem uma forma normal, então esta é única.*

Demonstração. Por redução ao absurdo, assuma-se que existem duas formas normais N e P, para as quais um certo termo M reduz.

[1] Henk P. Barendregt. *The lambda calculus – its syntax and semantics*, volume 103 de *Studies in Logic and Foundations of Mathematics.* North-Holland, 1984.

Pelo teorema de Church e Rosser, existe um termo Q para o qual N e P podem ser reduzidos. No entanto, N e P estão na forma normal, logo não admitem mais reduções. Então, a única interpretação possível é a de que $N = P = Q$.

■

2.4. Funções de ordem superior

No cálculo-λ não há distinção entre entidades simples, como os números, e outras mais complexas, como as funções. Tudo o que possa ser expresso como um termo-λ pode ser manipulado por outros termos-λ.

Vejamos um exemplo. Um termo para o quadrado de um número é dado por

$$Q \stackrel{\text{def}}{=} \lambda x.* \; x \; x$$

Se quisermos obter x^8 tal pode ser feito elevando x ao quadrado três vezes: $x^8 = ((x^2)^2)^2$. Com a notação do cálculo-λ podemos escrever a função "potência de 8" do seguinte modo:

$$P_8 \stackrel{\text{def}}{=} \lambda x. \, Q(Q(Q \; x))$$

Analogamente e para generalizar, podemos escrever um termo que aplica qualquer função três vezes, do seguinte modo:

$$T \stackrel{\text{def}}{=} \lambda f. (\lambda x. f(f \; (f \; x)))$$

Aqui os parênteses servem apenas para realçar o facto de T tomar uma função f como argumento e devolver outra função com argumento x.

Operadores como T são ditos de **ordem superior** porque eles operam sobre funções em vez de números.

2.5. Iteração e recursão no cálculo-λ

Como vimos anteriormente com o termo T, a combinação de termos-λ pode expressar a aplicação repetida de uma função. É possível generalizar esta ideia para obter o comportamento de um ciclo-for, onde o número de repetições é controlado por um contador. Vejamos como tal é possível.

Em primeiro lugar precisamos de uma constante que nos permita distinguir o zero dos outros números positivos. Chamemos a esta constante de "$zero$". O seu comportamento é semelhante ao de uma oração if-then-else dependendo do valor de um número:

$$zero(\ 0\ , x, y)\ \longrightarrow x$$
$$zero(\ n, x, y)\ \longrightarrow y\ \ \ (n \neq 0)$$

Consideremos ainda as constantes suc e $pred$ para as funções sucessor e predecessor, respectivamente.

Considere-se um termo I (para "iteração") o qual toma como argumentos um número n, uma função f e um valor x e que aplica n vezes a função f a x:

$$I\ n\ f\ x = f(f(f\ \ldots(f\ x)\ \ldots))$$

Se $n = 0$ então $I\ 0\ f\ x$ devolverá x, sem aplicar a função f. Podemos então definir a função I do seguinte modo:

$$I = \ \lambda n\ f\ x.zero\ (n, x, (I\ pred(n), f, (f\ x)))$$

A ideia é a seguinte: Se $n = 0$ então $zero(n, x, M)$ irá avaliar para x qualquer que seja M. Se $n > 0$ então f é iterado $(n - 1)$-vezes no argumento $(f\ x)$, o que devolve f aplicado a x n-vezes.

Observe-se que I é definida à custa de si própria, ou seja, I aparece no corpo da sua definição. Vamos ver como podemos eliminar a circularidade da definição.

Observe-se novamente a definição de I:

$$I = \ \lambda n\ f\ x.zero\ (n, x, (I\ pred(n), f, (f\ x)))$$

Podemos alterar o termo da direita por uma função S definida do seguinte modo:

$$S\ \stackrel{\text{def}}{=}\ \lambda M.(\lambda n\ f\ x.zero\ (n, x, (M\ pred(n), f, (f\ x))))$$

Esta definição já não é circular. Pretende-se agora encontrar um termo I tal que:

$$I = S\ I$$

ou seja, procuramos um termo que seja um ponto fixo de S.

Surpreendentemente, o ponto fixo pode sempre ser encontrado. Existem termos Y que constroem um ponto fixo para qualquer termo N, ou seja, eles satisfazem

$$Y\ N = N(Y\ N)$$

Uma vez encontrado tal Y o problema da definição circular estará resolvido e poderemos então dizer que $I\ \stackrel{\text{def}}{=}\ Y\ S$.

O dito Y chama-se **operador do ponto fixo**. Neste caso iremos usar o operador do ponto fixo de Turing (existem infinitos operadores do ponto fixo), cuja definição é a seguinte:

$$Y \stackrel{\text{def}}{=} \Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)\Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)$$

De facto:

$$\begin{aligned} YS &= \Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)\Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)S \\ &\longrightarrow S\Big(\Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)\Big(\lambda x.\lambda y.\big(y\ (x\ x\ y)\big)\Big)S\Big) \\ &\longrightarrow S(YS) \end{aligned}$$

Os pontos fixos são também utilizados para criar ciclos while. Vejamos o seguinte exemplo no qual se pretende encontrar o menor número para o qual uma determinada função devolve zero. Podemos implementar este problema como uma equação do ponto fixo, do seguinte modo:

$$Z = \lambda\, f\ n.zero((f\ n), n, (Z\ f\ (suc\ n))$$

Ou seja, *se* $f(n) = 0$ *então devolve* n*, caso contrário continua a procura para* $n + 1$. Transformando isto numa função para um qualquer termo M vem:

$$L \stackrel{\text{def}}{=} \lambda M.\Big(\lambda\, f\ n.zero\ \big((f\ n), n, (M\ f\ (suc\ n))\big)\Big)$$

Portanto, o procurador de raízes é dado por:

$$Z \stackrel{\text{def}}{=} Y\ L$$

A menor raiz de uma função, se existir, é dada por $Y\ L\ f\ 0$.

2.6. O cálculo-λ com tipos

No cálculo que tem vindo a ser desenvolvido, neste trabalho, falta ainda uma noção de **tipo**. O tipo de um termo deve dizer-nos qual o tipo do argumento que o termo aceita e qual o tipo do resultado que esse termo produz. Deste modo, os tipos impõem restrições que evitam paradoxos.

Em computação, existem diversas linguagens não tipadas como, por exemplo: LISP, λ-calculus, Self, Perl e Tcl. Essas linguagens não dispõem de nenhum mecanismo para a detecção de falhas devidas a operações aplicadas a argumentos impróprios. A ocorrência

de um erro dessa natureza não interrompe a execução do programa, sendo possível que o erro seja detectado somente após uma sequência bastante grande de operações subsequentes à ocorrência do mesmo.

Considere-se o seguinte termo "$sen\ log$". Aqui a função seno está a ser aplicada à função logaritmo. Ora, este termo não faz qualquer sentido, pois o argumento da função seno deveria ser um número e não uma função. Se quiséssemos expressar o tipo da função seno, este seria qualquer coisa como: "aceita números reais e produz números reais".

É, portanto, necessário definir um sistema com tipos para as funções (e alguns tipos básicos) ao qual irão ser progressivamente adicionados outros tipos e objectos.

Definição 2.10.

(i) Denote-se por $\boldsymbol{B}$ um alfabeto infinito cujos elementos serão chamados **tipo das variáveis** ou **tipos básicos**. O conjunto $Tipos$ de **tipos simples** é o conjunto das expressões definidas por:

$$Tipos ::= \boldsymbol{B} \mid (Tipos \longrightarrow Tipos)$$

As letras σ, τ, … denotam tipos arbitrários e se σ e τ são tipos, então $(\sigma \longrightarrow \tau)$ também é um tipo. Assume-se que "⟶" é associativo à direita e omitem-se os parênteses de acordo com tal.

(ii) O conjunto C de **contextos** é o conjunto de todos os conjuntos de pares da forma:

$$\{x_1 : \tau_1, \dots, x_n : \tau_n\}$$

onde $\tau_1, \dots, \tau_n \in Tipos$, $x_1, \dots, x_n$ são variáveis e $x_i \neq x_j$ para $i \neq j$.

(iii) O **domínio** de um contexto $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ é definido por:

$$dom(\Gamma) = \{x_1, \dots, x_n\}$$

(iv) O **tipo** de uma expressão é definido por:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

onde se impõe que $x \notin dom(\Gamma)$ na primeira e segunda regras. A notação $\Gamma, x{:}\,\sigma$ significa $\Gamma \cup \{x{:}\,\sigma\}$ com a condição de que x não ocorra em Γ.

Podemos agora denotar o tipo da função seno como "real → real" e o mesmo acontece com a função logaritmo. Portanto a função logaritmo não pode ser argumento da função seno pois não tem tipo "real".

Definição 2.11. Se $\Gamma \vdash M{:}\,\sigma$, então podemos dizer que M **tem tipo σ em Γ**. Dizemos que M é **tipável** se existe um Γ e σ tais que $\Gamma \vdash M{:}\,\sigma$.

Um contexto é uma presunção de que alguns elementos $x_1, \ldots, x_n$ têm determinados tipos $\sigma_1, \ldots, \sigma_n$, respectivamente. Além disso, se x tem tipo σ e M tem tipo τ, então $\lambda x.M$ tem tipo $\sigma \to \tau$. Do mesmo modo, se M tem tipo $\sigma \to \tau$ e N tem tipo σ, então MN tem tipo τ.

O Lema que se segue mostra que apenas o tipo das variáveis livres de um termo interessa na escolha dos contextos.

Lema 2.1. *Considere-se que $\Gamma \vdash M{:}\,\sigma$. Então:*

(i) *$\Gamma \subseteq \Gamma'$ implica $\Gamma' \vdash M{:}\,\sigma$;*

(ii) *$FV(M) \subseteq dom(\Gamma)$;*

(iii) *$\Gamma' \vdash M{:}\,\sigma$ onde $dom(\Gamma') = FV(M)$ e $\Gamma' \subseteq \Gamma$.*

Demonstração. (i) A prova vai ser feita por indução sobre a dedução de $\Gamma \vdash M{:}\,\sigma$.

1. Sabemos que

$$\frac{}{\Delta, x{:}\,\sigma \vdash x{:}\,\sigma}$$

onde $\Gamma = \Delta, x{:}\,\sigma$, $x \notin dom(\Delta)$ e $M = x$. Uma vez que $\Gamma \subseteq \Gamma'$ e Γ' é um contexto, então $\Gamma' = \Delta', x{:}\,\sigma$ para algum Δ' tal que $x \notin dom(\Delta')$. Portanto, $\Delta', x{:}\,\sigma \vdash x{:}\,\sigma$.

2. Podemos deduzir que

$$\frac{\Gamma, x{:}\,\tau_1 \vdash P{:}\,\tau_2}{\Gamma \vdash \lambda x.P{:}\,\tau_1 \to \tau_2}$$

onde $x \notin dom(\Gamma)$, $\sigma = \tau_1 \to \tau_2$ e $M = \lambda x.P$. Sem perda de generalidade, podemos assumir que $x \notin dom(\Gamma')$. Então $\Gamma, x{:}\,\tau_1 \subseteq \Gamma', x{:}\,\tau_1$ e pela hipótese de indução tem-se que $\Gamma', x{:}\,\tau_1 \vdash P{:}\,\tau_2$. Então também se tem $\Gamma' \vdash \lambda x.P{:}\,\tau_1 \to \tau_2$, como pretendido.

3. Tem-se que:

$$\frac{\Gamma \vdash P:\tau \to \sigma \quad \Gamma \vdash Q:\tau}{\Gamma \vdash PQ:\sigma}$$

onde $M = PQ$. Pela hipótese de indução $\Gamma' \vdash P:\tau \to \sigma$ e $\Gamma' \vdash Q:\tau$ e, por isso, $\Gamma' \vdash PQ:\sigma$, como pretendíamos.

A prova de (ii) e (iii) faz-se de forma análoga por indução na dedução de $\Gamma \vdash M:\sigma$.

■

Muitos dos resultados vistos anteriormente continuam a ser válidos neste sistema com tipos como é o exemplo do teorema de Church e Rosser.

Segue-se uma proposição que mostra que a relação de redução preserva tipos.

Proposição 2.1. *Se* $\Gamma \vdash M:\sigma$ *e* $M \longrightarrow_\beta N$, *então* $\Gamma \vdash N:\sigma$.

Corolário 2.2. *Se* $\Gamma \vdash M:\sigma$ *e* $M \longrightarrow^* N$ *então* $\Gamma \vdash N:\sigma$.

Teorema 2.2. (Teorema de Church-Rosser para termos tipáveis) *Suponha-se que* $\Gamma \vdash M:\sigma$. *Se* $M \longrightarrow^* N$ *e* $M \longrightarrow^* P$, *então existe um* Q *tal que* $N \longrightarrow^* Q$ *e* $P \longrightarrow^* Q$ *e* $\Gamma \vdash Q:\sigma$.

No cálculo-λ com tipos, uma expressão não pode ser aplicada a si própria. Assim, é impossível definir o termo Ω e o operador do ponto fixo, vistos anteriormente. Além disso, o cálculo-λ com tipos possui a propriedade de *terminação*, também chamada *normalização forte*.

2.7. Normalização forte

Teorema 2.3. (Normalização forte) *Para toda a expressão* M *do cálculo lambda com tipos, todas as sequências de redução que comecem em* M *são finitas.*

A demonstração deste teorema irá ser feita por indução sobre a estrutura dos tipos. Seguem-se algumas definições e resultados que irão ser necessários para a demonstração do Teorema 2.3.

Definição 2.12. O método de **indução sobre tipos** diz que, para provar o resultado $P(\tau)$ para todos os tipos τ, basta provar que:

- $P(\sigma)$ para todo $\sigma \in \boldsymbol{B}$. Este é chamado o caso base.

- $P(\sigma \longrightarrow \tau)$ assumindo que $P(\sigma)$ e $P(\tau)$ se verificam. Este é chamado o passo de indução.

Como é costume em provas por indução, para provar uma propriedade $R(M)$ para toda a expressão M, provamos uma propriedade R' mais forte que R. Tal deve-se ao facto da hipótese de indução não ser suficientemente forte para ser utilizada no passo de indução. Este é o caso da propriedade de M ser **fortemente normalizável** ($M \in$ FN) pois, dois termos M e M' podem ser fortemente normalizáveis sem que isso garanta que a aplicação $(M\ M')$ o seja.

Definição 2.13. Dizemos que uma expressão M de tipo τ é **estável**, e escreve-se $M \in \|\tau\|$, se

- Se M tem tipo básico e $M \in$ FN, ou
- Se M tem tipo $\sigma \longrightarrow \tau$ e para todo o M' em $\|\sigma\|$, a aplicação $(M\ M') \in \|\tau\|$.

Lema 2.2. *Se x é uma variável, então*

a) $x \in FN$.

b) Se $M_1, \ldots, M_k \in FN$ então $xM_1 \ldots M_k \in FN$.

c) Se $Mx \in FN$ então $M \in FN$.

d) Se $M \in FN$ então $(\lambda x.M) \in FN$.

Demonstração. (a) Uma vez que as variáveis não contêm redexes, então são fortemente normalizáveis.

(b) Toda a sequência de redução de $xM_1 \ldots M_k$ terá a forma:

$$xM_1 \ldots M_k \longrightarrow_\beta \ldots \longrightarrow_\beta xN_1 \ldots N_k \longrightarrow_\beta xP_1 \ldots P_k \longrightarrow_\beta \ldots$$

onde, em cada passo, para exactamente um índice j, $N_j \longrightarrow_\beta P_j$ e para todos os outros índices $N_i \equiv P_i$. Isto significa que, se existir uma sequência de redução infinita de $xM_1 \ldots M_k$, então existe uma sequência de redução infinita de um dos M_i's, o que contradiz o facto de eles serem FN.

(c) Uma sequência de redução de Mx poderá ter uma das seguintes formas:

$$Mx \longrightarrow_\beta M_1x \longrightarrow_\beta M_2x \longrightarrow_\beta \ldots \longrightarrow_\beta M_nx \longrightarrow_\beta \ldots$$

ou

$$Mx \longrightarrow_\beta M_1x \longrightarrow_\beta \ldots \longrightarrow_\beta (\lambda y.N)x \longrightarrow_\beta N[x/y] \longrightarrow_\beta N_1[x/y] \longrightarrow_\beta N_2[x/y] \longrightarrow_\beta \ldots$$

onde

$$\lambda y.N \longrightarrow_\beta \lambda y.N_1 \longrightarrow_\beta \lambda y.N_2 \longrightarrow_\beta \ldots$$

é uma sequência de redução de $M \longrightarrow_\beta M_1 \longrightarrow_\beta \ldots$. Nos dois casos, uma sequência infinita com início em Mx dá origem a uma com início em M.

(d) Uma sequência de redução começando em $\lambda x.M$ terá a forma

$$\lambda y.M \longrightarrow_\beta \lambda y.M_1 \longrightarrow_\beta \lambda M_2 \longrightarrow_\beta \ldots$$

onde

$$M \longrightarrow_\beta M_1 \longrightarrow_\beta M_2 \longrightarrow_\beta \ldots$$

e, portanto, uma sequência infinita começando em $\lambda y.M$ dá origem a outra começando em M.

■

Lema 2.3. *(a) Se $M \in \|\tau\|$, então $M \in FN$.*

(b) Se $xM_1 \ldots M_n : \tau$ e $M_1 \ldots M_n \in FN$, então $xM_1 \ldots M_n \in \|\tau\|$.

(c) Se $x : \tau$, então $x \in \|\tau\|$.

Demonstração A demonstração irá ser feita por indução sobre o tipo τ. As três alíneas serão provadas simultaneamente.

Caso Básico. τ é um tipo básico. A propriedade (a) é verdadeira por definição de estabilidade para um tipo básico.

Para (b), se $M_1, \ldots, M_n \in FN$, então pelo Lema 2.2, parte (b), $xM_1 \ldots M_k$ será fortemente normalizável, e, uma vez que, τ é um tipo básico, a expressão é estável.

Para provar (c), observe-se que toda a variável é fortemente normalizável e, portanto, é estável se for de tipo básico.

Passo de indução. Assuma-se que τ é o tipo $(\sigma \longrightarrow \rho)$ e que as propriedades (a), (b) e (c) se verificam para σ e ρ.

Para provar (a) assuma-se que $M \in \|\tau\|$. Pretende-se provar que M é FN. Tome-se x de tipo σ. Pela alínea (c) para σ, x é estável e, pela definição de estabilidade, Mx também é estável. Como Mx tem tipo ρ, pela alínea (a) para ρ, Mx é FN. Usando o Lema 2.2, parte (c), vem que M é FN.

Para provar (b), ou seja, para provar que $xM_1 \dots M_n \in \|\tau\|$ é necessário provar que $xM_1 \dots M_nN$ está em $\|\rho\|$, se $N \in \|\sigma\|$. Por hipótese $M_1, \dots, M_n \in FN$ e por (a) para σ, N também é FN. A expressão $xM_1 \dots M_nN$ é do tipo ρ e, portanto, por (b) para ρ tem-se que $xM_1 \dots M_nN \in \|\rho\|$ como era pretendido.

Finalmente, para provar (c), suponhamos que $N \in \|\sigma\|$. Por (a) para σ, N é FN e, uma vez que a expressão (xN) tem tipo ρ, pela alínea (b) para ρ, (xN) está em $\|\rho\|$, logo x é estável.

■

Pretende-se agora mostrar que todas as expressões são estáveis. Tal irá ser provado por indução estrutural sobre as expressões. Sabemos, pelo resultado anterior, que as variáveis são estáveis e, pela definição, as aplicações preservam a estabilidade. Vejamos o caso das abstracções. Pretende-se provar que $\lambda x.N$ é estável, se N for estável, e portanto tem que se provar que, para P estável e com tipo apropriado $(\lambda x.N)P$ é estável. Esta expressão reduz para $N[P/x]$. É necessário deduzir a estabilidade da primeira a partir da segunda expressão. Uma generalização deste caso aparece na alínea b) do seguinte lema, mas segue-se primeiro uma definição.

Definição 2.14. Uma **s-instância** M' de uma expressão M é uma substituição $M' \equiv M[N_1/x_1, \dots, N_r/x_r]$ onde os $N_i's$ são expressões estáveis.

Lema 2.4. *(a) Se M e N são estáveis, então (MN) também é estável.*

(b) Para todo $k \geq 0$ se $M[N/x]P_1 \dots P_k \in \|\tau\|$ e $N \in FN$, então

$$(\lambda x.M)NP_1 \dots P_k \in \|\tau\|$$

(c) Toda as s-instâncias M' de expressões M são estáveis.

Demonstração. Cada alínea será demonstrada separadamente.

(a) Se $M \in \|\sigma \to \tau\|$ e $N \in \|\sigma\|$, então, pela definição de estabilidade, $(MN) \in \|\tau\|$, ou seja, (MN) é estável.

(b) A prova desta alínea será feita por indução sobre o tipo τ.

Caso base: Suponhamos que τ é um tipo básico. Pretende-se provar que $(\lambda x.M)NP_1 \dots P_k$ é fortemente normalizável assumindo que $M[N/x]P_1 \dots P_k$ e N são fortemente normalizáveis.

Considere-se a forma geral de uma sequência de redução começando em $(\lambda x.M)NP_1 \dots P_k$.

Todas as sequências terão uma das duas seguintes formas:

$$\begin{aligned}(\lambda x.M)NP_1 \dots P_k &\longrightarrow^* (\lambda x.M')N'P'_1 \dots P'_k \\ &\longrightarrow_\beta M'[N'/x]P'_1 \dots P'_k \\ &\longrightarrow_\beta \dots\end{aligned}$$

ou

$$\begin{aligned}(\lambda x.M)NP_1 \dots P_k &\longrightarrow^* (\lambda x.M')N'P'_1 \dots P'_k \\ &\longrightarrow_\beta \dots\end{aligned}$$

Em ambos os casos, o topo da redex não é reduzido em computações subsequentes. No primeiro caso, uma vez que $M[N/x]P_1 \dots P_k \longrightarrow^* M'[N'/x]P'_1 \dots P'_k$, a sequência tem de ser finita porque $M[N/x]P_1 \dots P_k$ é fortemente normalizável. No segundo caso, podemos considerar esta sequência como a junção de duas sequências, uma começando em N e outra que não contém reduções de N. A sequência das reduções de N é finita porque N é FN, e a outra sequência também é finita porque pode ser transformada na correspondente sequência de reduções do termo $M[N/x]P_1 \dots P_k$, no qual não haverá reduções de N, e o termo $(\lambda x.M)NP_1 \dots P_k$ é FN.

Passo de indução. Suponhamos que τ é o tipo $(\sigma \to \rho)$. Para provar que $(\lambda x.M)NP_1 \dots P_k \in \|\tau\|$ temos que provar que

$$(\lambda x.M)NP_1 \dots P_k P \in \|\rho\|$$

para todo $P \in \|\sigma\|$. Uma vez que $M[N/x]P_1 \dots P_k \in \|\tau\|$, então

$$M[N/x]P_1 \dots P_k P \in \|\rho\|$$

então, por (b) para o tipo ρ tem-se que $(\lambda x.M)NP_1 \dots P_k P \in \|\rho\|$, como pretendíamos.

(c) A prova desta alínea vais ser feita por indução estrutural sobre expressões M. Existem três casos.

Variáveis. Considere-se uma variável x. Uma instância x' de x pode ter a forma da expressão estável $[\dots, N/x, \dots]$, ou, quando x não é o alvo de uma substituição, x' será x o qual é estável pelo Lema 2.3, parte (c).

Aplicação. Se M tem a forma $(M_1 M_2)$, então uma s-instância de M terá a forma $(M'_1 M'_2)$ onde cada M'_i é uma s-instância de M_i. Por indução, cada M'_i é estável, e pelo Lema 2.4 parte (a), a aplicação $(M'_1 M'_2)$ é estável.

Abstracção. Suponhamos que M tem a forma $\lambda x.N$. Pretende-se provar que qualquer instância substituição é estável. Estas instâncias têm a forma $\lambda x.N'$, onde N' é uma instância substituição de N. Pretende-se provar que $\lambda x.N'$ é estável, ou seja, pretende-se provar que $(\lambda x.N')P$ é estável, para P estável. Pelo Lema 2.3 parte (a), P é FN logo, aplicando o Lema 2.4 parte (b) com $k = 0$, $N'[P/x]$ é estável. Esta é também uma s-instância de N e, por indução é estável.

■

Demonstração: (Teorema 2.3)

Pretende-se provar que todas as sequências de redução que comecem em M são finitas, ou seja, que toda a expressão é fortemente normalizável.

Pela parte (c) do Lema 2.4, toda a expressão é estável e, pelo Lema 2.3 parte (a), toda a expressão estável é fortemente normalizável.

■

Para servir como modelo para linguagens de programação, o cálculo-λ com tipos tem de ser estendido com a introdução de operadores de ponto fixo (um para cada tipo funcional $\sigma \to \tau$), de modo a prover definições de funções recursivas. Desta forma, a linguagem torna-se "universal" (toda a função recursiva pode ser expressa na linguagem). Antes de podermos tratar desta questão terei que fazer algumas considerações sobre teoria de domínios

Capítulo III

Teoria de Domínios

3.1. Teoria de ordem

Definição 3.1. Uma **ordem parcial** (**po**) é um par $(D, \sqsubseteq)$, onde D é um conjunto e $\sqsubseteq$ é uma relação reflexiva, transitiva e anti-simétrica em D, ou seja, para todo $x, y, z \in D$:

- $x \sqsubseteq x$ (reflexividade)
- $x \sqsubseteq y$ e $y \sqsubseteq z \implies x \sqsubseteq z$ (transitividade)
- $x \sqsubseteq y$ e $y \sqsubseteq x \implies x = y$ (anti-simetria)

Definição 3.2. Seja $(D, \sqsubseteq)$ uma ordem parcial.

1. Um elemento $x \in D$ é chamado **limite superior** de um subconjunto $A \subseteq D$, se x está acima de todos os elementos de A e escreve-se $\uparrow A = x$. O conceito dual é o de **limite inferior**.
2. Um elemento $x \in D$ é **maximal** se não existir nenhum outro elemento em D acima dele. A definição dual é a de elemento **minimal**.
3. Se todos os elementos de D estão acima de um único elemento $x \in D$, então x diz-se o **elemento menor**, também chamado *bottom*, e denota-se frequentemente por $\bot$.
4. Se, para um subconjunto $A \subseteq D$, o conjunto dos limites superiores tem um elemento menor x, então x chama-se **supremo** e escreve-se $x = \bigsqcup A$. Se $A = \{x, y\}$, então o supremo escreve-se $x \sqcup y$. O conceito dual é o de **ínfimo** e escreve-se $x = \bigsqcap A$.

3.2. Conjuntos direccionados

Definição 3.3. Seja $(D, \sqsubseteq)$ uma po. Um subconjunto A de D diz-se **direccionado**, se for não-vazio e todos os pares de elementos de A tiverem um limite superior em A. Se um conjunto direccionado A tiver supremo, então este denota-se por $\bigsqcup^{\uparrow} A$.

Um exemplo simples de conjuntos direccionados é o caso das cadeias.

Definição 3.4. Para uma po $(D, \sqsubseteq)$, um conjunto $X \subseteq D$ diz-se uma **cadeia** se, para todo o par de elementos x, y em X se tem que $x \sqsubseteq y$ ou $y \sqsubseteq x$.

Definição 3.5. Uma ordem parcial $(D, \sqsubseteq)$ é chamada de **pré-domínio** ou **ordem parcial completa** (**cpo**) se e só se qualquer subconjunto direccionado de D tiver um supremo.

Definição 3.6. Um cpo $(D, \sqsubseteq)$ é chamado **domínio** ou **cpo pontiagudo** se e só se tiver um menor elemento $\perp$.

Definição 3.7. Um elemento d de um cpo $(D, \sqsubseteq)$ diz-se **compacto** se, para todo o subconjunto direccionado X de D, se $d \sqsubseteq \bigsqcup X$ então existe $e \in X$ tal que $d \sqsubseteq e$.

Definição 3.8. Um cpo D diz-se **algébrico** se, para todo o elemento $d \in D$, o subconjunto $\{x \sqsubseteq d : x \text{ é compacto}\}$ de D é direccionado, e o seu supremo é d.

Definição 3.9. Um subconjunto X de um cpo D diz-se **consistente** se tiver um limite superior em D.

Definição 3.10. Um cpo D diz-se **consistentemente completo** se, para todos os seus subconjuntos não vazios, o supremo e o ínfimo existem.

Definição 3.11. Um cpo $(D, \sqsubseteq)$ diz-se um **domínio de Scott** se D for algébrico e todos os subconjuntos de D que tenham limite superior tiverem supremo.

3.3. Funções monótonas

Definição 3.12. Sejam $(D, \sqsubseteq_D)$ e $(E, \sqsubseteq_E)$ ordens parciais. Uma **função** $f : D \to E$ diz-se **monótona** se, para todo $x, y \in D$ tais que $x \sqsubseteq_D y$, então $f(x) \sqsubseteq_E f(y)$ em E.

Proposição 3.1. *Se L é consistentemente completo, então toda a função monótona de L para L tem um ponto fixo. O menor ponto fixo é dado por:*

$$\bigsqcap \{x \in L : f(x) \sqsubseteq x\}$$

Demonstração. Seja $A = \{x \in L: f(x) \sqsubseteq x\}$ e $a = \sqcap A$. Para cada $x \in A$ tem-se que $a \sqsubseteq x$ e $f(a) \sqsubseteq f(x) \sqsubseteq x$. Tomando o ínfimo, obtém-se que $f(a) \sqsubseteq \sqcap f(A) \sqsubseteq \sqcap A = a$ e $a \in A$. Por outro lado, $x \in A$ o que implica que $f(x) \in A$ por definição de função monótona. Aplicando isto a a vem que $f(a) \in A$ e portanto, $a \sqsubseteq f(a)$.

■

3.4. Funções contínuas

Definição 3.13. Se D e E são cpo's, então uma **função** $f: D \longrightarrow E$ é **contínua (de Scott)** se for monótona e se, para todo o subconjunto direccionado $X \subseteq D$ se tem que $\bigsqcup^{\uparrow}\{f(x): x \in X\} = f(\bigsqcup^{\uparrow} X)$.

Portanto, uma função contínua é uma função que preserva supremos de conjuntos direccionados.

Definição 3.14. Uma **função** entre domínios é chamada **estrita** se e só se preserva elementos mínimos.

Definição 3.15. O **conjunto das funções contínuas de D para E**, onde D e E são cpo's, denota-se por $[D \longrightarrow E]$.

Teorema 3.1. *Se D e E são cpo's, então* $E^D = [D \longrightarrow E]$ *é um cpo com a seguinte relação de ordenação:*

$$f \sqsubseteq g \qquad \text{se e só se} \qquad \forall x \in D \quad f(x) \sqsubseteq g(x)$$

Demonstração. Seja F um subconjunto direccionado de $[D \longrightarrow E]$. Pretende-se provar que a função g definida por $g(x) = \bigsqcup_{f \in F}^{\uparrow} f(x)$ é o supremo de F, o que se verifica pela construção de g. Resta verificar que g é contínua. Seja $A \subseteq D$ direccionado.

$$\begin{aligned} g(\bigsqcup^{\uparrow} A) &= \bigsqcup_{f \in F}^{\uparrow} f(\bigsqcup^{\uparrow} A) \\ &= \bigsqcup_{f \in F}^{\uparrow} \bigsqcup_{a \in A}^{\uparrow} f(a) \\ &= \bigsqcup_{a \in A}^{\uparrow} \bigsqcup_{f \in F}^{\uparrow} f(a) \\ &= \bigsqcup_{a \in A}^{\uparrow} g(a). \end{aligned}$$

■

O teorema que se segue garante a existência de um operador do ponto fixo mínimo $Y:[D \longrightarrow D] \longrightarrow D$.

Teorema 3.2. *Seja D um domínio.*

1. *Toda a função contínua f em D tem um ponto fixo mínimo dado por $\bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)$.*
2. *A atribuição $Y:[D \longrightarrow D] \longrightarrow D$ que a cada $f \longmapsto \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)$ é contínua.*

Demonstração. (1) O conjunto $\{f^n(\bot): n \in \mathbb{N}\}$ é uma cadeia. Tal deve-se ao facto de $\bot \sqsubseteq f(\bot)$ e à monotonia de f. Pelo facto de f ser contínua tem-se que $f(\bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)) = \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^{n+1}(\bot) = \bigsqcup_{n\in\mathbb{N}}^{\uparrow} f^n(\bot)$.

Se x for qualquer outro ponto fixo de f, então de $\bot \sqsubseteq x$ obtém-se que $f(\bot) \sqsubseteq f(x) = x$. Portanto, x é um limite superior de todos os $f^n(\bot)$, logo tem de estar acima de $Y(f)$.

(2) Primeiro consideremos o n-ésimo operador iteração $it_n:[D \longrightarrow D] \longrightarrow D$ o qual a cada f faz corresponder $f^n(\bot)$. Vamos provar que este operador é contínuo por indução. Para $n = 0$ este operador fica igual à função constante que devolve sempre $\bot$. Para o passo de indução, considere-se F uma família direccionada de funções contínuas em D.

$$\begin{aligned} it_{n+1}(\bigsqcup^{\uparrow} F) &= (\bigsqcup^{\uparrow} F)(it_n(\bigsqcup^{\uparrow} F)) \\ &= (\bigsqcup^{\uparrow} F)(\bigsqcup_{f\in F}^{\uparrow} it_n(f)) \\ &= \bigsqcup_{g\in F}^{\uparrow} g(\bigsqcup_{f\in F}^{\uparrow}(it_n(f))) \\ &= \bigsqcup_{g\in F}^{\uparrow} \bigsqcup_{f\in F}^{\uparrow} g(it_n(f)) \\ &= \bigsqcup_{f\in F}^{\uparrow} f^{n+1}(\bot) \end{aligned}$$

O supremo de todos operadores iteração é precisamente Y, logo, este último também é contínuo. ■

3.5. Princípios de Indução para Pontos Fixos Mínimos

O operador do ponto fixo mínimo é o correspondente matemático das declarações recursivas e iterativas das linguagens de programação e irá garantir definições recursivas em PCF.

A definição que se segue introduz uma classe de propriedades sobre domínios nos quais os princípios de indução posteriores são "admissíveis".

Definição 3.16. Um subconjunto P de um cpo D diz-se uma **propriedade admissível** em D se e só se P é fechado sobre supremos de conjuntos direccionados.

Teorema 3.3. (Indução Computacional) *Seja D um domínio, $P \subseteq D$ uma propriedade admissível em D e $f: D \longrightarrow D$ uma função contínua. Então $P(Y(f))$ sempre que $P(f^n(\perp))$ para todo o $n \in \mathbb{N}$.*

Segue-se uma consequência imediata do teorema anterior.

Teorema 3.4. (Indução do Ponto Fixo) *Seja D um domínio, $f: D \longrightarrow D$ uma função contínua e $P \subseteq D$ uma propriedade admissível em D. Então P satisfaz $Y(f)$ sempre que $P(\perp)$ e $\forall x \in D$ se $P(x) \Rightarrow P(f(x))$.*

Segue-se um teorema que se deve a David Park e que é útil na demonstração de que funções definidas recursivamente divergem para alguns argumentos.

Teorema 3.5. (Indução de Park) *Seja D um domínio e $f: D \longrightarrow D$ uma função contínua. Então $Y(f) \sqsubseteq d$ sempre que $f(d) \sqsubseteq d$.*

Capítulo IV

A Linguagem de Programação PCF

Nesta secção irei apresentar uma linguagem de programação funcional PCF, a sua semântica operacional e denotacional, a relação existente entre elas e algumas características desta linguagem, tais como, adequabilidade computacional e abstracção completa.

4.1. Os termos de PCF

Os termos de PCF são os considerados no cálculo-λ com tipos.

Relativamente ao conjunto **Tipo** dos tipos, tem-se que:

(1) **nat** é o tipo básico de números naturais;

(2) **bool** é o tipo básico de valores de verdade;

(3) Se σ_1 e σ_2 são tipos, então $(\sigma_1 \to \sigma_2)$ é um tipo.

É usual representar tipos por σ, σ_1, σ_2, …, e é frequente omitir os parênteses assumindo que $'\to'$ associa à direita. Deste modo $\sigma_1 \to \sigma_2 \to \sigma_3$ é uma abreviação de $(\sigma_1 \to (\sigma_2 \to \sigma_3))$.

As letras M e N denotam termos e, para indicar que um termo M tem tipo σ, escreve-se $M{:}\,\sigma$ ou M_σ.

Existem quatro grupos de **termos** de PCF:

1. Aritméticos básicos;
2. Condicionais;
3. Recursão (Operador do ponto fixo);
4. Cálculo lambda.

Os dois primeiros são as constantes de PCF.

Os termos de PCF e os seus tipos são dados pelas seguintes regras de inferência:

1. $\dfrac{}{\Gamma, x{:}\sigma, \Delta \vdash x{:}\sigma}$ $\qquad$ $\dfrac{}{\Gamma \vdash zero{:}\textbf{nat} \longrightarrow \textbf{bool}}$ $\qquad$ $\dfrac{}{\Gamma \vdash true{:}\textbf{bool}}$

$\dfrac{}{\Gamma \vdash false{:}\textbf{bool}}$ $\qquad$ $\dfrac{\Gamma \vdash M{:}\textbf{nat}}{\Gamma \vdash suc(M){:}\textbf{nat}}$ $\qquad$ $\dfrac{\Gamma \vdash M{:}\textbf{nat}}{\Gamma \vdash pred(M){:}\textbf{nat}}$

2. $\dfrac{\Gamma \vdash M_i{:}\textbf{nat}\ \ (i = 1,2,3)}{\Gamma \vdash ifz(M_1, M_2, M_3){:}\textbf{nat}}$ $\qquad$ $\dfrac{\Gamma \vdash M{:}\textbf{bool} \quad \Gamma \vdash M_1{:}\sigma \quad \Gamma \vdash M_2{:}\sigma}{\Gamma \vdash if_\sigma(M, M_1, M_2){:}\sigma}$ $(\sigma \in \{\textbf{nat},\textbf{bool}\})$

3. $\dfrac{\Gamma \vdash M{:}\sigma \longrightarrow \sigma}{\Gamma \vdash Y_\sigma(M){:}\sigma}$

4. $\dfrac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x.M{:}\sigma \to \tau}$ $\qquad$ $\dfrac{\Gamma \vdash M{:}\sigma \to \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash (MN){:}\tau}$

Podemos então dizer que, os candidatos a termos de PCF são as constantes, as variáveis, as aplicações, as abstracções e o operador do ponto fixo.

É usual escrever Ω_σ como uma abreviação de $Y_\sigma(\lambda x{:}\sigma.x)$.

Definição 4.1. **Contextos** são termos que têm "buracos" como subtermos. Esses buracos são representados por variáveis-buraco $X, Y, Z, \ldots$. Os contextos representam-se usualmente por $C[\cdot, \ldots, \cdot]$ e são definidos pelas seguintes regras:

(1) Toda a variável, x, é um contexto.

(2) Toda a constante, c, é um contexto.

(3) Toda a variável-buraco, X, é um contexto.

(4) Se C é um contexto, então $(\lambda x.C)$ é um contexto.

(5) Se C é um contexto, então (CC) é um contexto.

(6) Se C é um contexto, então $\boldsymbol{Y}_\sigma(C)$ é um contexto.

A **substituição em contextos** define-se recursivamente do seguinte modo:

$$C[M_1, \ldots, M_n] = \begin{cases} C & \text{se } C \text{ é uma variável ou constante} \\ M_i & \text{se } C \text{ é o buraco } X_i \\ (C_1[M_1, \ldots, M_n]C_2[M_1, \ldots, M_n] & \text{se } C = (C_1C_2) \\ (\lambda x.C'[M_1, \ldots, M_n]) & \text{se } C = (\lambda x.C') \\ \boldsymbol{Y}(C'[M_1, \ldots, M_n]) & \text{se } C = \left(\boldsymbol{Y}(C')\right) \end{cases}$$

A principal diferença entre a substituição em contextos e a substituição em termos é que na primeira as variáveis podem aparecer ligadas no resultado, enquanto que na outra

as variáveis nunca são ligadas. Por exemplo, seja $C[X]$ igual a $\lambda x.Xx$, então $C[xy] \equiv \lambda x.(xy)x$ e portanto x é ligada como resultado; mas $C[w][(xy)/w] \equiv \lambda z.(xy)z \equiv \lambda z'.(xy)z'$ (a variável ligada x em $C[w] \equiv \lambda x.wx$ é renomeada para z para evitar que fique ligada como resultado da substituição.

4.2. Semântica operacional de PCF

Definição 4.2. A semântica operacional é dada por uma **função de avaliação**, $\Downarrow$, definida à custa de uma **relação de redução imediata** entre termos, $\longrightarrow$, do seguinte modo:

$$M \Downarrow c \text{ se e só se } M \longrightarrow^* c, \text{ para algum programa } M \text{ e constante } c.$$

A **relação de redução**, $\longrightarrow$, foi vista na secção sobre Redução no capítulo II. Vamos agora adicionar algumas regras de inferência de acordo com a linguagem PCF:

$$\frac{}{zero(\underline{0}) \longrightarrow true} \qquad \frac{}{zero(\underline{n+1}) \longrightarrow false}$$

$$\frac{}{pred(\underline{n+1}) \longrightarrow \underline{n}} \qquad \frac{}{pred(\underline{0}) \longrightarrow \underline{0}}$$

$$\frac{M \longrightarrow N}{pred(M) \longrightarrow pred(N)} \qquad \frac{M \longrightarrow N}{suc(M) \longrightarrow suc(N)}$$

$$\frac{}{ifz(\underline{0}, M, N) \longrightarrow M} \qquad \frac{}{ifz(\underline{n+1}, M, N) \longrightarrow N}$$

$$\frac{}{Y_\sigma(M) \longrightarrow M(Y_\sigma(M))} \qquad \frac{}{(\lambda x{:}\sigma.M)(N) \longrightarrow M[N/x]}$$

$$\frac{M_1 \longrightarrow M_2}{if_\sigma(M_1, N_1, N_2) \longrightarrow if_\sigma(M_2, N_1, N_2)}$$

Aqui $\underline{n}$ é o número natural n, definido por recursão sobre k do seguinte modo: $\underline{0} \equiv$ zero e $\underline{k+1} = suc(\underline{k})$.

A **relação de avaliação**, também conhecida como Semântica do Grande Passo (Big Step Semantics), pode ser definida de acordo com as seguintes regras:

$$\frac{}{x \Downarrow x} \qquad \frac{}{\underline{0} \Downarrow \underline{0}}$$

$$\frac{}{true \Downarrow true} \qquad \frac{}{false \Downarrow false}$$

$$\frac{M \Downarrow \underline{0}}{zero(M) \Downarrow true} \qquad \frac{M \Downarrow \underline{n+1}}{zero(M) \Downarrow false}$$

$$\frac{M \Downarrow \underline{n}}{suc(M) \Downarrow \underline{n+1}} \qquad \frac{M \Downarrow \underline{n+1}}{pred(M) \Downarrow \underline{n}}$$

$$\frac{M \Downarrow \underline{0}}{pred(M) \Downarrow \underline{0}} \qquad \frac{M(Y_\sigma(M)) \Downarrow V}{Y_\sigma(M) \Downarrow V}$$

$$\frac{M \Downarrow \underline{n+1} \qquad M_2 \Downarrow V}{ifz(M, M_1, M_2) \Downarrow V} \qquad \frac{M \Downarrow \underline{0} \quad M_1 \Downarrow V}{ifz(M, M_1, M_2) \Downarrow V}$$

$$\frac{M \Downarrow true \qquad M_1 \Downarrow V}{if_\sigma(M, M_1, M_2) \Downarrow V} \qquad \frac{M \Downarrow false \qquad M_2 \Downarrow V}{if_\sigma(M, M_1, M_2) \Downarrow V}$$

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.E \qquad E[N/x] \Downarrow V}{M(N) \Downarrow V}$$

4.3. Equivalência observacional

Uma vez que os termos só têm interesse enquanto parte integrante dos programas, podemos dizer que dois termos são operacionalmente equivalentes se for possível substitui-los livremente, um pelo outro, num programa sem afectar o comportamento deste.

Definição 4.3. Suponhamos que M e N são termos do mesmo tipo. Define-se **equivalência observacional**, $\approx$ por:

$M \approx N$ se , em qualquer contexto $C[X]$ tal que $C[M]$ e $C[N]$ são programas, e para qualquer valor V, $C[M] \Downarrow V$ se e só se $C[N] \Downarrow V$.

Definição 4.4. Sejam M e N termos do mesmo tipo $\sigma = (\sigma_1, \dots, \sigma_n, \tau)$.

(i) Dizemos que M se **aproxima aplicativamente** de N, escreve-se $M \precsim N$, se, para todos os termos fechados $P_1 : \sigma_1, \dots, P_n : \sigma_n$ e para algum V:

$$M \precsim N \quad \text{sse} \quad MP_1 \dots P_n \Downarrow V \;\Rightarrow\; NP_1 \dots P_n \Downarrow V$$

(ii) Dizemos que M se **aproxima observacionalmente** de N, escreve-se $M \lesssim N$, se em qualquer contexto $C[X]$ tal que $C[M]$ e $C[N]$ são programas e para algum V:

$$M \lesssim N \quad \text{sse} \quad C[M] \Downarrow V \;\Rightarrow\; C[N] \Downarrow V$$

Portanto,

$$M \approx N \;\text{ sse }\; M \lesssim N \text{ e } N \lesssim M.$$

Definição 4.5. Para todo o tipo σ, seja Prg_σ o conjunto $\{M : \vdash M : \sigma\}$ de termos fechados de PCF, também chamados **programas de tipo σ**. Aos programas de tipo básico chamaremos apenas programas.

Por indução sobre a estrutura de σ, podemos dar definições alternativas das relações $\precsim_\sigma$ e $\lesssim_\sigma$ em Prg_σ:

(i) **aproximação aplicativa**:

para o tipo básico **nat**:

$$M \precsim_{\mathbf{nat}} N \quad \text{sse} \quad \forall n \in \mathbb{N} \;\; M \Downarrow \underline{n} \Rightarrow N \Downarrow \underline{n}$$

para o tipo funcional $\sigma \longrightarrow \tau$:

$$M \precsim_{\sigma\to\tau} N \quad \text{sse} \quad \forall P \in Prg_\sigma \;\; M(P) \precsim_\tau N(P).$$

(ii) **aproximação observacional**:

$$M \lesssim N \;\text{ sse }\; \forall P \in Prg_{\sigma\to\mathbf{nat}} \;\; P(M) \precsim_{\mathbf{nat}} P(N)$$

Será provado, mais tarde, que as relações $\precsim$ e $\lesssim$ coincidem para todos os tipos (Lema do Contexto de Milner).

4.4. Semântica denotacional

Uma semântica denotacional para PCF associa a cada tipo σ um domínio D_σ e a cada termo (num contexto) $x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n \vdash M{:}\,\sigma$ uma função

$$[\![x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n]\!]{:}\, D_{\sigma_1} \times \ldots \times D_{\sigma_n} \longrightarrow D_\sigma$$

assumindo que o produto cartesiano entre domínios está definido.

Para vermos como definir domínios que interpretem os tipos de PCF precisamos da seguinte definição.

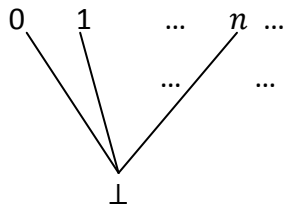
Definição 4.6. Seja X um conjunto. Então $X_\perp$ é a po cujos elementos são $X \cup \{\perp\}$ onde $\perp \notin X$ e cuja relação de ordenação é:

$$x \sqsubseteq y \quad sse \quad x = \perp \vee x = y$$

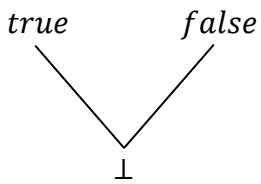
A relação " $\sqsubseteq$ " é conhecida como **ordem de informação** na qual $\perp$ é o menor elemento e todos os outros são incomparáveis.

Estamos agora em condições de definir os domínios associados aos tipos de PCF, dotados da referida ordem de informação:

- $D_{\textbf{nat}} = \mathbb{N}_\perp$
- $D_{\textbf{bool}} = \mathbb{B}_\perp$, com $\mathbb{B} = \{true, false\}$
- $D_{\sigma \to \tau} = [D_\sigma \longrightarrow D_\tau]$



Ordem em $\mathbb{N}_\perp$



Ordem em $\mathbb{B}_\perp$

Definição 4.7. Um **modelo standard** de PCF é uma família $\{D_\sigma\}$ de cpo's, uma para cada tipo σ, onde $D_{\textbf{nat}} = \mathbb{N}_\perp$, $D_{\textbf{bool}} = \mathbb{B}_\perp$ e $D_{\sigma_1 \to \sigma_2} = [D_{\sigma_1} \longrightarrow D_{\sigma_2}]$.

Definição 4.8. Uma **interpretação standard** de PCF é modelo standard $\{D_\sigma\}$ juntamente com as seguintes interpretações: para $n \in D_{\textbf{nat}}$ e $b \in D_{\textbf{bool}}$

$[\![true]\!] = true$,

$$[\![false]\!] = false,$$

$$[\![suc]\!]n = \begin{cases} n+1 & \text{se } n \geq 0 \\ \bot & \text{se } n = \bot \end{cases},$$

$$[\![pred]\!]n = \begin{cases} n-1 & \text{se } n \geq 1 \\ 0 & \text{se } n = 0, \\ \bot & \text{se } n = \bot \end{cases}$$

$$[\![zero]\!]n = \begin{cases} true & \text{se } n = 0 \\ false & \text{se } n \geq 1 \\ \bot & \text{se } n = \bot \end{cases},$$

$$[\![ifz]\!]n = \begin{cases} true & \text{se } n = 0 \\ false & \text{se } n > 0, \\ \bot & \text{se } n = \bot \end{cases}$$

$$[\![if_\sigma]\!]bde = \begin{cases} d & (\text{se } b = true) \\ e & (\text{se } b = false) \\ \bot & (\text{se } b = \bot) \end{cases},$$

$$[\![\boldsymbol{Y}_\sigma]\!](f) = \bigsqcup_{n \geq 0} f^n(\bot) \quad (f \in D_{\sigma \to \sigma}).$$

A denotação de um termo que possui variáveis livres depende do significado atribuído a essas variáveis. Por isso, a função semântica é parametrizada por outra função, denominada **ambiente**. O ambiente associa a cada variável um elemento do domínio. O ambiente indefinido ⊥, associa todas as variáveis a ⊥. A denotação de um termo M num ambiente ρ, representa-se por $[\![M]\!](\rho)$.

Definição 4.9. Dizemos que ρ é um **Γ–ambiente** no caso de, para todo $x{:}\,\sigma$ em Γ, $\rho(x) \in D_\sigma$. O conjunto de todos os Γ −ambientes representa-se por Env(Γ)

Definição 4.10. Dado um Γ −ambiente ρ, a semântica denotacional

$$[\![\Gamma \vdash M{:}\,\sigma]\!](\rho) \in D_\sigma$$

define-se por recursão sobre M do seguinte modo:

1. $[\![\Gamma \vdash x{:}\,\sigma]\!](\rho) = \rho(x)$, ou seja, uma variável denota o que o ambiente lhe atribuir;
2. $[\![\Gamma \vdash c{:}\,\sigma]\!](\rho) = [\![c]\!]$, ou seja, uma constante denota o que a interpretação lhe atribuir;

3. $[\![\Gamma \vdash MN:\tau]\!](\rho) = [\![\Gamma \vdash M:\sigma \longrightarrow \tau]\!](\rho)\big([\![\Gamma \vdash N:\sigma]\!](\rho)\big)$, ou seja, se um termo M denota a função $f:D \longrightarrow E$ e o termo N denota o valor $x \in D$, então a combinação MN denota o valor $f(x) \in E$.
4. $[\![\Gamma \vdash \lambda x:\sigma.M:\sigma \longrightarrow \tau]\!](\rho) = \lambda d \in D_\sigma.[\![\Gamma, x:\sigma \vdash M:\tau]\!](\rho[d/x])$, ou seja, se um termo M denota o valor $y_d \in E$ num ambiente que atribui o valor $d \in D$ à variável x, então a abstracção $\lambda x.M$ denota a função $f:D \longrightarrow E$ definida por $f(d) = y_d$.

Se M é fechado, então a sua denotação não depende do ambiente, no sentido em que $[\![\Gamma \vdash M:\sigma]\!](\rho) = [\![\Gamma \vdash M:\sigma]\!](\rho')$ para todo ρ e ρ'.

Observações:

(1) Se $[\![x]\!](\rho) = [\![x]\!](\rho')$ para todo x pertencente a $FV(M)$, então

$$[\![\Gamma \vdash M:\sigma]\!](\rho) = [\![\Gamma \vdash M:\sigma]\!](\rho');$$

(2) Para M e N termos, e para todo ρ,

$$[\![\Gamma \vdash M[N/x]:\sigma]\!](\rho) = [\![\Gamma, x:\tau \vdash M:\sigma]\!](\rho[[\![\Gamma \vdash N:\tau]\!](\rho)/x]);$$

(3) Se $[\![\Gamma \vdash M:\sigma]\!](\rho) = [\![\Gamma \vdash N:\sigma]\!](\rho)$ para todo ρ, então para todo o contexto $C[X]$ tal que $\Gamma, \Delta \vdash C[M]:\tau$ e $\Gamma, \Delta \vdash C[N]:\tau$, então para todo ρ (Γ, Δ) −ambiente,

$$[\![\Gamma, \Delta \vdash C[M]:\tau]\!](\rho) = [\![\Gamma, \Delta \vdash C[N]:\tau]\!](\rho).$$

Para simplificação de escrita, irá por vezes escrever-se $[\![M]\!](\rho)$ em vez da denotação $[\![\Gamma \vdash M:\sigma]\!](\rho)$, assumindo que as varáveis livres de M estão implicitamente tipadas de acordo com um Γ fixo.

4.5. Adequação Computacional

Como foi visto anteriormente, dois termos são observacionalmente equivalentes se puderem ser trocados, um pelo outro num programa, sem afectar o output deste. De acordo com a semântica denotacional, o significado de um programa (ou termo) é aquele que a função semântica denotar. Portanto, do ponto de vista denotacional, dois programas são iguais se tiverem a mesma denotação no modelo.

Definição 4.11. Uma semântica denotacional diz-se **adequada** para PCF, se a igualdade denotacional implica a equivalência observacional, ou seja, se $\Gamma \vdash M:\sigma$ e $\Gamma \vdash N:\sigma$ então

$$\forall \rho \in Env(\Gamma) \quad [\![M]\!](\rho) = [\![N]\!](\rho) \quad \Rightarrow \quad M \approx N$$

Teorema 4.1. (Adequabilidade Computacional) *Para todo o termo fechado M de tipo* **nat** *se $[\![M]\!] = n$ então $M \Downarrow \underline{n}$.*

A prova da adequação computacional vai ser feita do seguinte modo: define-se uma relação R_σ para todo o tipo σ; prova-se que para todo o termo fechado M de tipo σ tem-se que $[\![M]\!]R_\sigma M$. Esta relação vai ser definida de tal forma que $[\![M]\!]R_{\mathbf{nat}}M$ é equivalente a $\forall n \in N \quad [\![M]\!] = n \quad \Rightarrow \quad M \Downarrow \underline{n}$, o que completa a prova do teorema.

Definição 4.12. Defina-se uma relação $R_\sigma \subseteq D_\sigma \times Prg_\sigma$, para cada tipo σ, do seguinte modo:

$dR_{\mathbf{nat}}M$ sse $\forall n \in \mathbb{N} \; d = n \Rightarrow M \Downarrow \underline{n}$

$fR_{\sigma\to\tau}M$ sse $\forall d \in D_\sigma \; \forall N \in Prg_\sigma \quad dR_\sigma N \Rightarrow f(d)R_\tau M(N)$

À família $R = (R_\sigma : \sigma \in \mathbf{Tipo})$ chama-se **relação lógica** entre a semântica e a sintaxe de PCF.

Seguem-se algumas propriedades da relação lógica R que mostram que $[\![M]\!]R_\sigma M$ se verifica para todo $M \in Prg_\sigma$.

Lema 4.1. *Para todo o tipo σ, tem-se que:*

(1) *Se $d' \sqsubseteq d$ e $dR_\sigma M$, então $d'R_\sigma M$;*

(2) *Para todo $M \in Prg_\sigma$, o conjunto $R_\sigma M := \{d \in D_\sigma : dR_\sigma M\}$ é fechado sobre supremos direccionados e contém $\perp$;*

(3) *Se $dR_\sigma M$ e $M \cong_\sigma M'$, então $dR_\sigma M'$.*

Demonstração. Seja $\sigma = \sigma_1 \to \cdots \to \sigma_k \to$ **nat**.

(1) Suponhamos que $g \sqsubseteq f$ e $fR_\sigma M$. Para provar que $gR_\sigma M$ suponhamos que $d_i R_{\sigma_i} N_i$ para $i = 1, \ldots, k$ e $g(d_1) \ldots (d_k) = n$. Então, $f(d_1) \ldots (d_k) = n$ uma vez que $g \sqsubseteq f$. Portanto, como $fR_\sigma M$ tem-se que $M(N_1) \ldots (N_k) \Downarrow \underline{n}$, logo $gR_\sigma M$.

(2) Suponhamos que $F \subseteq R_\sigma M$ é direccionado e $fR_\sigma M$ para todo o $f \in F$. Para provar que $\bigsqcup F R_\sigma M$, suponhamos que $d_i R_{\sigma_i} N_i$ para $i = 1, \ldots, k$ e $(\bigsqcup F)(d_1) \ldots (d_k) =$

n. Então, existe um $f \in F$ com $f(d_1) \dots (d_k) = n$, donde resulta que $M(N_1) \dots (N_k) \Downarrow \underline{n}$ porque $f R_\sigma M$.

Como $\bot\ (d_1) \dots (d_k) = \bot$ para todo d_i, então $\bot\ R_\sigma M$.

(3) Suponhamos que $f R_\sigma M$ e $M \cong_\sigma M'$. Para provar que $f R_\sigma M'$ suponhamos que $d_i R_{\sigma_i} N_i$ para $i = 1, \dots, k$ e $f(d_1) \dots (d_k) = n$. Então, de $f R_\sigma M$ segue que $M(N_1) \dots (N_k) \Downarrow \underline{n}$. Portanto, como $M \cong_\sigma M'$, tem-se que $M'(N_1) \dots (N_k) \Downarrow \underline{n}$, como pretendíamos.

■

Lema 4.2. *Se $f R_{\sigma\to\sigma} M$, então $Y(f) R_\sigma Y_\sigma(M)$.*

Demonstração. Suponhamos que $f R_{\sigma\to\sigma} M$. Pelo Lema 4.1 (2), para provar que $Y(f) R_\sigma Y_\sigma(M)$, basta provar que $f^n(\bot) R_\sigma Y_\sigma(M)$ para todo $n \in \mathbb{N}$. A demosntração faz-se por indução sobre n. O caso base $\bot\ R_\sigma Y_\sigma(M)$ verifica-se pelo lema anterior. Suponhamos que $f^n(\bot) R_\sigma Y_\sigma(M)$. Então, como $f R_\sigma M$ segue que $f^{n+1}(\bot)\ R_\sigma M(Y_\sigma(M))$. Basta provar que $M(Y_\sigma(M)) \cong Y_\sigma(M)$, uma vez que, pelo Lema 4.1 (3), juntamente com $f^{n+1}(\bot) R_\sigma M(Y_\sigma(M))$, tem-se que $f^{n+1}(\bot) R_\sigma Y_\sigma(M)$.

Para provar que $M(Y_\sigma(M)) \cong Y_\sigma(M)$ suponhamos que $M(Y_\sigma(M))(N_1) \dots (N_k) \Downarrow \underline{n}$. Então existem valores $V_1, \dots, V_k$ tais que $M(Y_\sigma(M)) \Downarrow V_1$, $V_i(N_i) \Downarrow V_{i+1}$ para $i > k$ e $V_k(N_k) \Downarrow \underline{n}$. Mas, de $M(Y_\sigma(M)) \Downarrow V_1$ segue que $Y_\sigma(M) \Downarrow V_1$ e, portanto, também $Y_\sigma(M)(N_1) \dots (N_k) \Downarrow \underline{n}$.

■

Lema 4.3. *Se $x_1 \colon \sigma_1, \dots, x_k \colon \sigma_k \vdash M \colon \tau$ e $d_i R_{\sigma_i} N_i$ para $i = 1, \dots, k$, então*

$$[\![x_1 \colon \sigma_1, \dots, x_k \colon \sigma_k \vdash M]\!]\big(\vec{d}\big) R_\tau M[\vec{N}/\vec{x}].$$

Nota: $\vec{d}$ é o k-uplo $\langle d_1 \dots d_k \rangle$.

Demonstração. A demonstração será feita por indução sobre os termos. Para facilitar a notação, escrever-se-á $\vec{d} R \vec{N}$ em vez de $d_i R_{\sigma_i} N_i$ para $i = 1, \dots, k$.

(Variáveis) Para $x_1 \colon \sigma_1, \dots, x_k \colon \sigma_k \vdash x_i \colon \sigma_i$ e $\vec{d} R \vec{N}$, tem-se que

$$[\![x_1 \colon \sigma_1, \dots, x_k \colon \sigma_k \vdash x_i]\!] = d_i\ R_{\sigma_i}\ N_i \equiv x_i[\vec{N}/\vec{x}]$$

(λ-Abstracção) Suponhamos que o teorema se verifica para $\Gamma, x \colon \sigma \vdash M \colon \tau$ e que $\vec{d} R \vec{N}$. Pretende-se provar que

$$[\![\Gamma \vdash \lambda x{:}\sigma.M]\!](\vec{d})\ R_{\sigma\to\tau}\ (\lambda x{:}\sigma.M)[\vec{N}/\vec{x}]$$

onde $\vec{x}$ é a lista das variáveis declaradas em Γ. Para tal, assuma-se que $dR_{\sigma}N$. Da hipótese de indução segue que

$$[\![\Gamma, x{:}\sigma \vdash M]\!](\vec{d}, d)\ R_{\tau}\ M[\vec{N}, N/\vec{x}, x]$$

e, pelo Lema 4.1, tem-se que

$$[\![\Gamma \vdash \lambda x{:}\sigma.M]\!](\vec{d})(d)\ R_{\tau}\ (\lambda x{:}\sigma.M)[\vec{N}/\vec{x}](N)$$

uma vez que

$$[\![\Gamma \vdash \lambda x{:}\sigma.M]\!](\vec{d})(d) = [\![\Gamma, x{:}\sigma \vdash M]\!](\vec{d}, d)$$

e

$$M[\vec{N}, N/\vec{x}, x] \equiv M[\vec{N}/\vec{x}][N/x] \cong_{\tau} (\lambda x{:}\sigma.M[\vec{N}/\vec{x}])(N) \equiv (\lambda x{:}\sigma.M)[\vec{N}/\vec{x}](N).$$

(Aplicações) Suponhamos que, por hipótese de indução, que o teorema se verifica para $\Gamma \vdash M_1{:}\sigma \longrightarrow \tau$ e $\Gamma \vdash M_2{:}\sigma$. Se $\vec{d}R\vec{N}$, então

$$[\![\Gamma \vdash M_i]\!](\vec{d})\ R\ M_i[\vec{N}/\vec{x}]$$

para $i = 1,2$, donde segue que

$$[\![\Gamma \vdash M_1]\!](\vec{d})\left([\![\Gamma \vdash M_2]\!](\vec{d})\right) R\ M_1[\vec{N}/\vec{x}](M_2[\vec{N}/\vec{x}]).$$

Como

$$[\![\Gamma \vdash M_1(M_2)]\!](\vec{d}) = [\![\Gamma \vdash M_1]\!](\vec{d})([\![\Gamma \vdash M_2]\!](\vec{d}))$$

e

$$M_1[\vec{N}/\vec{x}](M_2[\vec{N}/\vec{x}]) \equiv M_1(M_2)[\vec{N}/\vec{x}]$$

então, tem-se que

$$[\![\Gamma \vdash M_1(M_2)]\!](\vec{d})\ R\ M_1(M_2)[\vec{N}/\vec{x}]$$

(Recursão) Suponhamos, por hipótese de indução, que $\Gamma \vdash M{:}\sigma \longrightarrow \sigma$ satisfaz o teorema. Se $\vec{d}R\vec{N}$, então, pela hipótese de indução, tem-se que

$$[\![\Gamma \vdash M]\!](\vec{d})\ R\ M[\vec{N}/\vec{x}]$$

donde, pelo Lema 4.2, segue que

$$[\![Y_{\sigma}(M)]\!](\vec{d}) = Y\left([\![\Gamma \vdash M]\!](\vec{d})\right)\ R\ Y_{\sigma}(M[\vec{N}/\vec{x}]) \equiv Y_{\sigma}(M)[\vec{N}/\vec{x}] \qquad \blacksquare$$

A demonstração do Teorema 4.1 é imediata pelo lema anterior, uma vez que se trata de um caso particular deste.

Estamos agora em condições de enunciar o Lema do Contexto de Milner:

Teorema 4.2. (Teorema do Contexto de Milner) *Para todos os tipos* σ *e* $M, N \in Prg_\sigma$, *as seguintes condições são equivalentes:*

(a) $M \cong_\sigma N$

(b) $M \lesssim_\sigma N$

(c) $[\![M]\!]\, R_\sigma\, N$

Demonstração. (a) ⇒ (c): Suponhamos (a). Pelo Lema 4.3 sabe-se que $[\![M]\!]\, R\, M$. Portanto, como $M \cong_\sigma N$ por (a), pelo Lema 4.1 (3) segue que $MR_\sigma N$.

(c) ⇒ (b): Suponhamos que $[\![M]\!]R_\sigma N$. Seja $P \in Prg_{\sigma\to\mathbf{nat}}$. Pelo Lema 4.3, tem-se que $[\![P]\!]\, R_{\sigma\to\mathbf{nat}} P$. Portanto, daqui segue que $[\![P]\!]([\![M]\!])R_{\mathbf{nat}} N(P)$. Como $[\![P(M)]\!] = [\![P]\!]([\![M]\!])$, tem-se que $[\![P(M)]\!]\, R_{\mathbf{nat}}\, N(P)$. Se $P(M) \Downarrow \underline{n}$, então $[\![M(P)]\!] = n$ e, portanto, também $nR_{\mathbf{nat}} N(P)$, donde $N(P) \Downarrow \underline{n}$.

(b) ⇒ (a): porque contextos da forma $[]\vec{P}$ são apenas contextos particulares de tipo básico.

■

4.6. Computabilidade

Definição 4.13. Os predicados de **computabilidade**, são definidos por indução sobre os tipos por:

1. Se $M{:}\,\sigma$ é um programa então M é computável se e só se $[\![M]\!] = [\![V]\!]$ implica que $M \Downarrow V$.
2. Se $M{:}\,\sigma \to \tau$ é um termo fechado, este é computável se e só se MN é computável para todo $N{:}\,\sigma$ termo fechado computável.

3. Se $M{:}\,\sigma$ é um termo aberto com as variáveis livres $x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n$ então este é computável se, sempre que $N_1, \ldots, N_n$ forem termos fechados computáveis, então $M_\theta \equiv M[N_1/x_1] \ldots [N_n/x_n]$ é computável.

Portanto, um termo $M{:}\,(\sigma_1, \ldots, \sigma_n, \tau)$ é computável se e só se

- Qualquer substituição θ das variáveis livres de M por termos fechados computáveis resulta num termo M_θ computável;
- A aplicação de M a termos fechados computáveis resultar num termo computável.

Lema 4.4. *Todo o termo de PCF é computável.*

Demonstração Por indução estrutural sobre as regras de formação de termos:

(1) Toda a **variável** é computável porque, por definição, $x_\theta\vec{N}$ é computável, para toda a substituição fechada computável θ e para toda a sequência de termos fechados computáveis $N_i{:}\,\sigma_i$.

(2) Toda a **constante** diferente de Y_σ é computável. Tal é claro para constantes de tipo básico. Para suc, $pred$, $zero$, e if_σ considere-se $pred$ como exemplo. É suficiente mostrar que $pred(M)$ é computável quando M:**nat** é um termo fechado computável. Suponhamos que $[\![pred(M)]\!] = [\![c]\!]$. Então $c = k_m$ para algum m e portanto $[\![M]\!] = m + 1$. Uma vez que M é computável, $M \longrightarrow^* k_{m+1}$ e portanto $pred(M) \longrightarrow^* k_m = c$.

(3) Se $M{:}\,\tau$ é computável, então a **abstracção** $\lambda x.M$ também é. Suponhamos que $[\![(\lambda x{:}\,\sigma_1.M)_\theta\vec{N}]\!] = [\![V]\!]$, para qualquer substituição fechada computável θ de $\lambda x.M$ e para qualquer sequência de termos fechados computáveis $N_i{:}\,\sigma_i$. Note-se que:

(A) $[\![(\lambda x{:}\,\sigma_1.M)_\theta N_1]\!] = [\![(M_\theta[M_1/x{:}\,\sigma_1])]\!] = [\![(M[N_1/x{:}\,\sigma_1])_\theta]\!]$

(B) $(M[N_1/x_{\sigma_1}])_\theta \equiv M_\theta$, onde $\theta' \stackrel{\text{def}}{=} \theta[N_1/x_{\sigma_1}]$.

Juntando (A) e (B) observa-se que $[\![M_\theta, N_2 \ldots N_n]\!] = [\![(\lambda x{:}\,\sigma_1.M)_\theta N_1 \ldots N_n]\!] = [\![V]\!]$. Pela hipótese de indução, M é computável e, uma vez que a substituição θ' é fechada e computável para M, tem-se que $M_\theta, N_2 \ldots N_n \Downarrow V$, donde $(\lambda x{:}\,\sigma_1.M)\vec{N} \Downarrow V$.

(4) Cada $\boldsymbol{Y}_\sigma$ é computável. Para qualquer substituição θ de M fechada e computável e para qualquer sequência fechada computável de termos $N_i : \sigma_i$, suponhamos que $[\![Y_\sigma(M_\theta)\vec{N}]\!] = [\![V]\!]$. Tem-se que

$$[\![V]\!] = (\sqcup_{m\geq 0}[\![Y_\sigma^m(M_\theta)]\!])[\![N_1]\!] \dots [\![N_n]\!]$$
$$= \sqcup_{m\geq 0}[\![(Y_\sigma^m(M_\theta))N_1 \dots N_n]\!].$$

Então, para algum m, $[\![(Y_\sigma^m(M_\theta))N_1 \dots N_n]\!] = [\![V]\!]$. Uma vez que $Y_\sigma(M)$ é computável, tem-se que $(Y_\sigma^m(M_\theta))N_1 \dots N_n \Downarrow V$ donde $(Y_\sigma(M_\theta))\vec{N} \Downarrow V$.

(5) Se $M : \sigma \to \tau$ e $N : \sigma$ são computáveis, então a **aplicação** MN também é. Considerem-se as possíveis formas da aplicação:

- $xP\vec{Q}$
- $cP\vec{Q}$
- $(\lambda x.P)P\vec{Q}$
- $Y(R)P\vec{Q}$

Tome-se como exemplo a última. Suponhamos que $[\![(Y(R)P\vec{Q})_\theta\vec{N}]\!] = [\![V]\!]$, para substituição fechada computável θ de $Y(R)P\vec{Q}$ e para toda a sequência fechada de termos computáveis $N_i : \sigma_i$. Pela hipótese de indução, P_θ e $\vec{Q}_\theta$ são fechados e computáveis e, uma vez que, $Y(R)$ é computável, então $(Y(R)P\vec{Q})_\theta\vec{N} \Downarrow V$. ■

4.7. Abstracção Completa para PCF

Como foi visto anteriormente se $[\![M]\!] = [\![N]\!]$ então $M \approx N$, porque o modelo é computacionalmente adequado. No entanto, Plotkin demonstrou que, no modelo standard, o contrário não é verdade e, por isso, este não é completamente abstracto.

Definição 4.14. Um modelo de PCF diz-se **completamente abstracto** se e só se $M \approx N \Rightarrow [\![M]\!] = [\![N]\!]$, para todos os termos fechados M e N com o mesmo tipo.

Para alcançar a abstracção completa, temos duas hipóteses:

1. Manter o modelo considerado e expandir a linguagem de modo que esta extensão possa ser interpretada no modelo;

2. Manter a linguagem e alterar o modelo para um que seja completamente abstracto para a linguagem dada.

Neste trabalho, irei apenas mostrar uma extensão da linguagem de PCF por uma constante chamada "se paralelo", que torna o modelo apresentado completamente abstracto.

A segunda abordagem é feita através de domínios sequenciais. Esta solução é conhecida como "aproximação relacional" porque os domínios são dotados de uma estrutura relacional adicional, na qual as funções entre domínios sequenciais têm de satisfazer a usual continuidade e ainda os requisitos da Teoria de Domínios de Scott.

Curien mostrou que a linguagem PCF aumentada com uma constante "se paralelo", $\boldsymbol{pif}$**,** torna o modelo completamente abstracto.

O tipo deste termo é dado por

$$\frac{\Gamma \vdash P\text{:}\mathbf{bool} \quad \Gamma \vdash M_i\text{:}\mathbf{nat}\ (i = 1,2)}{pif(P, M_1, M_2)\text{:}\mathbf{nat}}$$

Dado um termo P de PCF com tipo **bool**, adicionamos as seguintes regras de redução imediata:

$$\frac{}{pif(true, M, N) \longrightarrow M} \qquad \frac{}{pif(false, M, N) \longrightarrow N}$$

$$\frac{P \to P'}{pif(P, M, N) \longrightarrow pif(P', M, N)} \qquad \frac{M \to M'}{pif(P, M, N) \longrightarrow pif(P, M', N)}$$

$$\frac{N \to N'}{pif(P, M, N) \longrightarrow pif(P, M, N')}$$

À relação de avaliação, adicionamos as seguintes regras:

$$\frac{P \Downarrow true \quad M \Downarrow \underline{n}}{pif(P, M, N) \Downarrow \underline{n}} \qquad \frac{P \Downarrow false \quad N \Downarrow \underline{n}}{pif(P, M, N) \Downarrow \underline{n}} \qquad \frac{M \Downarrow \underline{n} \quad N \Downarrow \underline{n}}{pif(P, M, N) \Downarrow \underline{n}}$$

A interpretação de pif é a esperada:

$$[\![pif]\!]PMN = \begin{cases} [\![M]\!] & \text{se } [\![P]\!] = true \\ [\![N]\!] & \text{se } [\![P]\!] = false \\ [\![M]\!] & \text{se } [\![P]\!] = \perp \text{ e } [\![M]\!] = [\![N]\!] \\ \perp & \text{se } [\![P]\!] = \perp \text{ e } [\![M]\!] \neq [\![N]\!] \end{cases}$$

Para provar que o modelo é completamente abstracto para PCF + pif recorremos ao teorema de definibilidade.

Teorema 4.3. (Definibilidade)

(i) Para cada tipo σ, D_σ é um domínio de Scott.

(ii) Um modelo de PCF é completamente abstracto se e só se todo o elemento compacto de D_σ é definível.

Lema 4.5. *(i) Se D e E forem domínios de Scott, então $[D \to E]$ também é.*

(ii) Os elementos compactos de $[D \to E]$ são os supremos de conjuntos finitos de elementos da forma $(d \searrow e)$, com d e e elementos compactos de D e E, respectivamente, onde, para $x \in D$

$$(d \searrow e)(x) = \begin{cases} e & \text{se } d \sqsubseteq x \\ \perp & \text{caso contrário} \end{cases}$$

Demonstração. Suponhamos que f e g pertencentes a $[D \longrightarrow E]$ têm um limite superior h, então para qualquer x, $h(x)$ é um limite superior de $f(x)$ e $g(x)$. Portanto, como E é um domínio de Scott, podemos definir uma função $k: D \longrightarrow E$ por:

$$k(x) = f(x) \sqcup g(x) \quad (x \in D).$$

Claramente k é o supremo de f e g em $[D \to E]$. Portanto, $[D \to E]$ é algébrico.

Agora suponhamos que $(d \searrow e) \sqsubseteq \bigsqcup F$ onde F é um subconjunto direccionado de $[D \to E]$. Então, $e = (d \searrow e)(d) \sqsubseteq (\bigsqcup F)(d) = \bigsqcup_{f \in F} f(d)$. Como e é compacto, $e \sqsubseteq f(d)$ para algum $f \in F$. Então, $(d \searrow e) \sqsubseteq f$. Portanto, $(d \searrow e)$ é compacto e segue que qualquer supremo de um conjunto finito de elementos da forma $(d \searrow e)$ também é compacto.

Tomemos $f \in [D \to E]$ e consideremos o conjunto:

$F = \{\bigsqcup F' : F'$ é um conjunto finito de elementos de $[D \to E]$ da forma $(d \searrow e)$ e $\sqsubseteq f\}$

Será provado que $f = \bigsqcup F$. Se $f' \in F$ então $f \sqsupseteq f'$, portanto $\bigsqcup F$ existe e $f \sqsupseteq (\bigsqcup F)$.

Para provar que $f \sqsubseteq (\bigsqcup F)$, considere-se $x \in D$ e um elemento compacto $e \sqsubseteq f(x)$.

Então,

$$e \sqsubseteq f(x) = f(\bigsqcup\{d \in D: d \text{ compacto e } \sqsubseteq x\})$$
$$= \bigsqcup\{f(d): d \text{ compacto e } \sqsubseteq x\}.$$

Portanto, para algum $d \sqsubseteq x$ compacto tem-se que $e \sqsubseteq f(d)$. Assim, $(d \searrow e) \sqsubseteq f$ e como $(d \searrow e) \in F$ então $(\bigsqcup F)(x) \sqsupseteq d$. Uma vez que d era arbitrário, $(\bigsqcup F)(x) \sqsupseteq f(x)$.

Se f é compacto, então, como F é direccionado, f é o supremo de um conjunto finito de elementos da forma $(d \searrow e)$, como era pretendido.

Portanto, se em vez de f ser um membro arbitrário de $[D \to E]$, F seria o conjunto de todos os membros compactos de $[D \to E]$ tais que $\sqsubseteq f$. Como $f = \bigsqcup F$, então $[D \to E]$ é algébrico, concluindo a demonstração do lema.

■

Uma vez que os domínios do modelo standard $D_{\textbf{nat}} = \mathbb{N}_\perp$ e $D_{\textbf{bool}} = \mathbb{B}_\perp$ são domínios de Scott, pelo lema anterior, para qualquer tipo σ, D_σ é um domínio de Scott.

Para cada domínio de Scott D, escrevemos $K(D)$ para representar a colecção de elementos compactos de D. Considere-se $\sigma = (\sigma_1, \dots, \sigma_n, \tau)$. Pelo lema anterior, qualquer elemento compacto de D_σ é $\bigsqcup F$, onde F representa os subconjuntos finitos de D_σ que satisfazem:

(1) Cada elemento de F tem a forma $(M_1 \searrow (M_2 \searrow \cdots (M_n \searrow V) \dots))$ onde $M_i \in K(D_{\sigma_i})$ para cada $1 \leq i \leq n$ e $V \in K(D_\sigma)$ com $V \neq \perp$,

(2) Para cada dois elementos $(M_1 \searrow (M_2 \searrow \cdots (M_n \searrow V) \dots))$ e $(M'_1 \searrow (M'_2 \searrow \cdots (M'_n \searrow V') \dots))$ de F, se M_i e M'_i são computáveis para todo $1 \leq i \leq n$, então $V = V'$.

Lema 4.6. *Para cada tipo σ, todos os elementos compactos de D_σ, no modelo standard, são definíveis em PCF + pif.*

Demonstração. A demonstração é feita por indução sobre os tipos e pretende-se mostrar que se e e f são elementos compactos em D_σ então e, $(e \searrow true)$ e, se existir, $(e \searrow true) \sqcup (f \searrow false)$ são definíveis em PCF + pif.

$\sigma =$ **bool**: $\perp, true$ e $false$ são definíveis por $\Omega_{\textbf{bool}}$, $true$ e $false$.

$(\perp \searrow true), (true \searrow true)$ e $(false \searrow true)$ são definíveis por $\lambda\, p.\, true$, $\lambda p.\, (if_{\textbf{bool}}(p, true, \Omega_{\textbf{bool}}))$ e $\lambda p.\, (if_{\textbf{bool}}(p, \Omega_{\textbf{bool}}, false))$, respectivamente.
$(true \searrow true) \sqcup (false \searrow false)$ e $(false \searrow true) \sqcup (true \searrow false)$ são definidos por $\lambda p.\, p$ e $\lambda p.\, (if_{\textbf{bool}}(p, false, true))$.

$\sigma = \textbf{nat}$: $\perp$ e n são definidos por $\Omega_{\textbf{nat}}$ e k_n.
$(\perp \searrow true)$ e $(n \searrow true)$ são definidos por $\lambda x.\, true$ e $\lambda x.\, (if_{\textbf{bool}}(zero(pred^n(x)), true, \Omega_{\textbf{bool}})$; $(k_m \searrow true) \sqcup (k_n \searrow false)$ e $(k_n \searrow true) \sqcup (k_m \searrow false)$, onde $m < n$, são definidos por $\lambda x.\, \left(if_{\textbf{bool}}(zero(pred^m(x)), true, if_{\textbf{bool}}(zero(pred^n(x), false, \Omega_{\textbf{bool}}))\right)$ e $\lambda x.\, \left(if_{\textbf{bool}}\left(zero(pred^m(x)), false, if_{\textbf{bool}}(zero(pred^n(x)), true, \Omega_{\textbf{bool}})\right)\right)$.

$\sigma = (\sigma_1, \dots, \sigma_n, \tau)$ com τ básico: Suponhamos que e e f são supremos de conjuntos finitos F e F'. Para provar que e e $(e \searrow true)$ são definíveis, irá usar-se a indução sobre o tamanho de F e depois mostrar-se-á que $(e \searrow true) \sqcup (f \searrow false)$ é definível, se existir.

Para e, se $F = \emptyset$, e é definido por $\Omega_{\textbf{bool}}$.

Considere-se $F = \emptyset$. Se existem $(e_1 \searrow \cdots \searrow e_n \searrow d)$ e $(e'_1 \searrow \cdots \searrow e'_n \searrow d')$ em F tais que, para algum i, $e_i \sqcup e'_i$ não existe então, $(e_i \searrow true) \sqcup (e'_i \searrow false)$ existe e é definível, digamos por M_1. Para além disso, $\bigsqcup(F\{e_1 \searrow \cdots \searrow d\})$ e $\bigsqcup(F\{e'_1 \searrow \cdots \searrow d'\})$ são definíveis por, digamos, F_1 e F_2. Então, $\bigsqcup F$ é ele próprio definível por:

$$\lambda x_1^{\sigma_1} \dots \lambda x_n^{\sigma_n} \left(pif_\tau\left(M_1 x_i^{\sigma_i}\right)\left(F_2 x_1^{\sigma_1} \dots x_n^{\sigma_n}\right)\left(F_1 x_1^{\sigma_1} \dots x_n^{\sigma_n}\right)\right).$$

Por outro lado se $(e_1 \searrow \cdots \searrow d)$ e $(e'_1 \searrow \cdots \searrow d')$ estão em F, então $d = d'$, e todos os $e_i \sqcup e'_i$'s existem. Tomemos $(e_1 \searrow \cdots \searrow e_n \searrow d)$ em F e considere-se que $E_1, \dots, E_n, D$, F_1 definem $(e_1 \searrow true), \dots, (e_n \searrow true)$, d e $\bigsqcup(F\{e_1 \searrow \cdots \searrow d\})$. Então $\bigsqcup F$ é definido por

$$\lambda x_1^{\sigma_1} \dots \lambda x_n^{\sigma_n} \left(\left(pif_\tau\left(E_1 x_1^{\sigma_1}\right) AND \dots AND \left(E_n x_n^{\sigma_n}\right)\right) D\left(F_1 x_1^{\sigma_1} \dots x_n^{\sigma_n}\right)\right).$$

Aqui AND é o termo $\lambda p \lambda q \left(if_{\textbf{bool}}\left(p, \left(if_{\textbf{bool}}(q, true, false)\right), false\right)\right)$ e é usado como infixo.

Para $(e \searrow true)$, se $F = \emptyset$, e é definido por $\lambda x^{\sigma}.true$.

Suponhamos que $F = \emptyset$ e tomemos $(e_1 \searrow \cdots \searrow e_n \searrow d)$ em F e considere-se que $E_1, \ldots, E_n$, D, F_1 definem $e_1, \ldots, e_n$, $(d \searrow true)$ e $((\bigsqcup F\{(e_1 \searrow \cdots \searrow d)\}) \searrow true)$, respectivamente. Então $(e \searrow true)$ é definido por:

$$\lambda x^{\sigma}\left(if_{\textbf{bool}}\left(\left(D(x^{\sigma}E_1, \ldots, E_n)\right), (F_1 x^{\sigma}), \Omega_{\textbf{bool}}\right)\right).$$

Se $(e \searrow true) \sqcup (f \searrow false)$ existe, $e \sqcup f$ não existe e existirem $(e_1 \searrow \cdots \searrow e_n \searrow d)$ em F e $(e'_1 \searrow \cdots \searrow e'_n \searrow d')$ em F' tais que $e_1 \sqcup e'_1, \ldots, e_n \sqcup e'_n$ existe mas $d \neq d'$. Considere-se que $E_1, \ldots, E_n$, F_1, F'_1, D definem $(e_1 \sqcup e'_1), \ldots, (e_n \sqcup e'_n)$, $(e \searrow true)$, $(f \searrow true)$ e $(d \searrow true) \sqcup (d' \searrow false)$, respectivamente.

Então $(e \searrow truet) \sqcup (f \searrow false)$ é definido por:

$$\lambda x^{\sigma}\left(if_{\textbf{bool}}\left(\left(D(x^{\sigma}E_1, \ldots, E_n)\right), (F_1 x^{\sigma}), NEG(F'_1 x^{\sigma})\right)\right)$$

onde NEG é o termo $\lambda p.\left(if_{\textbf{bool}}(p, false, true)\right)$.

O que conclui a prova do lema.

■

Teorema 4.4. (Abstracção completa)

O modelo standard é completamente abstracto para PCF + pif.

Demonstração. A demonstração segue directamente do Lema 4.6 e do Teorema 4.3.

■

Bibliografia

[1] Thompson, Simon. *Type Theory & Functional Programming*. Computing Laboratory, University of Kent, 1999.

[2] Milner, Robin. *Models of LCF*. Computer Science Department, Stanford University, 1973.

[3] Jung, Achim. *A short introduction to the Lambda Calculus*. School of Computer Science, The University of Birmingham, 2004.

[4] Mislove, Michael W.. *Topology, Domain Theory and Theoretical Computer Science*. Department of Mathematics, Tulane University. New Orleans, LA, 1996.

[5] Ong, C.-H. L.. *Correspondence between Operational and Denotational Semantics*. Oxford University Computing Laboratory.

[6] Streicher, Thomas. *Mathematical Foundations of Functional Programming*. 2002.

[7] Sorensen, Morten Heine B. and Urzyczyn, Pawel. *Lectures on the Curry-Howard Isomorphism*. University of Copenhagen and University of Warsaw.

[8] Rojas, Raúl. *A Tutorial Introduction to the Lambda Calculus*. FU Berlin, 1998.

[9] Hughes, John. *Why Functional Programming Matters*. Institutionem for Datavetenskap.

[10] Goldberg, Benjamin. *Functional Programming Languages*. New York University, 1996.

[11] Escardó, Martín Hotzel. *PCF extended with real numbers: a domain-theoretic approach to higher-order exact real number computation*. University of London, Imperial College of Science, Technology and Medicine, Department of Computing, 1996.

[12] Abramsky, Samson and Jung, Achim. *Domain Theory*. Computing Laboratory, University of Oxford and School of Computer Science, University of Birmingham.

[13] Plotkin, G. D.. *LCF considered as a Programming Language*. Department of Artificial Intelligence, University of Edinburgh, 1975.

[14] Selinger, Peter. *Lecture Notes on the Lambda Calculus*. Department of Mathematics and Statistics, University of Ottawa.

[15] Barendregt, Henk P.. *The lambda calculus – its syntax and semantics*, volume 103 de *Studies in Logic and Foundations of Mathematics.* North-Holland, 1984.

[16] Pfenning, Frank. *A proof of the Church-Rosser Theorem and its representation in a logical framework*. School of Computer Science, Carnegie Mellon University. Pittsburgh, 1992.

[17] Pimentel, Elaine Gouvêa. *Fundamentos da Matemática*. 2008

[18] Paolini, Luca. *Semantics of PCF and the full abstraction problem*. Università di Torino, Dipartimento di Informatica, 2006.