Transformações *pointwise – point-free*

José Proença
ze@correio.ci.uminho.pt

Techn. Report DI-PURe-05:02:01
2005, Fevereiro

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-05:02:01
Transformações pointwise – point-free by José Proença

Abstract

Na programação funcional existem vários estilos de programação, não havendo consenso quanto a qual será o melhor. Dois estilos opostos são, por exemplo, *point-free* e *pointwise*, que se distinguem principalmente por no primeiro serem utilizadas variáveis e no segundo não.

Neste trabalho foi introduzida uma possível descrição de expressões em *point-free* e em *pointwise*, de tal forma que estas fossem o mais simples possível. Depois foi introduzido um processo que permite converter expressões entre estes dois tipos. Foram então consideradas outras linguagens *pointwise* (nomeadamente, PCF (capítulo 7) e BNL (capítulo 8)), que podem ser convertidas para o tipo *pointwise* originalmente criado (o qual se passou a designar por Core).

O principal objectivo deste trabalho foi tornar possível a transição de código *pointwise* para *point-free* utilizando bases teóricas e com a garantia de que não sejam introduzidos erros no processo de conversão.

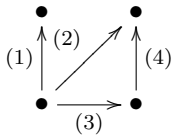
Conteúdo

1 Introdução

Neste projecto optou-se por utilizar o paradigma funcional.

Neste paradigma destacam-se dois estilos distintos: *pointwise* e *point-free*, que se distinguem principalmente pela utilização de variáveis no primeiro, e uma ausência destas no segundo.

Também se podem distinguir os casos em que é usada recursividade explícita e implícita. Na recursividade explícita é normal encontrar-se na definição de funções o nome da própria função que está a ser definida, enquanto na recursividade implícita são usadas funções que descrevem o tipo de recursividade (como os paramorfismos, hilomorfismos, *etc*). No quadro abaixo podem ver-se as várias alternativas de código segundo as duas características descritas acima.

	PW PF
rec. implíc.	
rec. explíc.	

Em geral, existe uma tendência maior para escrever código em *point-wise* com recursividade explícita.

Existem no entanto vantagens e desvantagens em cada estilo de programação. Seria bom então poder usufruir das vantagens de todos os estilos caso fosse possível comutar entre várias formas de programar.

Este trabalho trata principalmente a forma como se poderia fazer a conversão (3) marcada no quadro acima. A conversão no sentido inverso também é abordada mas torna-se menos interessante pois é relativamente fácil. É ainda tratada a conversão (2) para alguns paramorfismos.

Esta é a conversão que mais interesse parece ter, directa ou indirectamente.

Para a conversão (1) já existem ferramentas que permitem a transformação de recursividade explícita para implícita, como o *DrHylo*[1]. A conversão no sentido inverso a (1) é, mais uma vez, de pouco interesse.

[1] desenvolvido por Alcino Cunha

A conversão (2) parece ser a mais vantajosa. Para a maior parte dos casos, ela apenas pode ser conseguida indirectamente, como por (3) e por (4). Esta última já vai para além dos objectivos propostos para este trabalho.

Mas porquê *point-free*, se uma grande maioria dos programadores funcionais está habituada a programar em *pointwise*?

Não se pode afirmar que um dos estilos é melhor ou pior que o outro: cada um tem as suas vantagens e desvantagens, como referido acima. A principal desvantagem do *point-free* é que muito facilmente se torna difícil de compreender, como se exemplificará mais tarde. No entanto, torna possível um raciocínio elaborado, mais facilitado, sobre os programas, uma vez que são herdados conhecimentos matemáticos (leis algébricas e equacionais) que permitem calcular e transformar programas de forma rigorosa.

Estrutura do Relatório

Neste trabalho começa-se por descrever no capítulo 2 o que consiste em programar em *point-free*, introduzindo uma notação teórica e descrevendo alguns padrões de recursividade. É então mostrado algum trabalho feito anteriormente em *Haskell* que implementa estas mesmas ideias no capítulo 3.

Seguem-se no capítulo 4 aspectos mais práticos, como a definição de tipos de dados em *Haskell* que representem termos *point-free*. Da mesma forma é introduzida, no capítulo 5, uma possível definição de uma linguagem *pointwise* simplificada (aqui designada por Core), e são definidos novos tipos de dados que representem estes termos.

No capítulo seguinte (6) é definida uma forma de efectuar conversões entre estes tipos de dados. Finalmente são definidas em *Haskell* duas linguagens *pointwise*: PCF e BNL (nos capítulos 7 e 8, respectivamente), que apenas tratam um determinado conjunto de tipos primitivos, e que podem ser convertidos para a linguagem Core definida anteriormente.

As principais diferenças entre as três linguagens definidas em *pointwise* são:

- **Core** – Linguagem mais genérica, onde a recursividade é definida somente com o operador de ponto-fixo;
- **PCF** – Linguagem definida apenas para os tipos booleano, natural e lista, onde a recursividade pode ser definida através do operador de ponto-fixo ou através de um letrec.

– **BNL** – Linguagem definida para os mesmos tipos da linguagem PCF, mas a recursividade é definida apenas por paramorfismos sobre listas e naturais.

Note-se que na linguagem *point-free* definida a recursividade pode ser expressa através do operador ponto-fixo ou de paramorfismos, sendo ainda possível a introdução de mais padrões recursivos através de *macros*.

2 Programação *point-free*

O estilo de programação algébrico *point-free* foi introduzido por John Backus em 1977, numa palestra ACM Turing Award [Bac78]. A maioria dos combinadores apresentados neste capítulo foram introduzidos por este autor.

O principal interesse deste paradigma é obter, além do seu poder de programação, o poder das leis algébricas associadas a este. A ideia principal é então desenvolver um cálculo de programas que possa ser utilizado na sua transformação.

Este capítulo encontra-se dividido em três secções:

– **Combinadores** – onde são apresentados os combinadores *point-free* mais importantes que permitem a escrita de programas neste estilo;
– **Tipos Recursivos** – onde estes tipos são descritos e relacionados com a noção de functor;
– **Padrões Recursivos** – onde são introduzidos os seguintes padrões de recursividade: catamorfismos, anamorfismos, hilomorfismos e paramorfismos.

2.1 Combinadores

O operador composição pode ser considerado um combinador *point-free*. Este será denotado pelo símbolo $\circ$, e atribui a um par de funções $g : A \to B$ e $f : B \to C$ a sua composição funcional $f \circ g : A \to C$. Esta será tal que $(f \circ g)\ x = f(g(x))$.

O diagrama seguinte ilustra o comportamento da composição

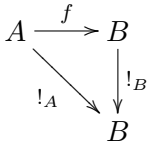
$$A \xrightarrow{g} B \xrightarrow{f} C \qquad A \xrightarrow{f \circ g} C$$

O *objecto terminal* aqui considerado é $1 = \{\bot_1\}$. Isto significa que para qualquer outro objecto A existe uma única função que vai deste

para 1, nomeadamente a função que devolve sempre $\perp_1$. Esta função será denotada por $\cdot!_A$. A sua unicidade é capturada pela lei

$$!_B \circ f = !_A$$

que é válida para qualquer $f : A \to B$, como se pode verificar no seguinte diagrama:



Serão agora introduzidos construtores de tipos (como produtos e somas), cada qual munido dos seus próprios combinadores e leis.

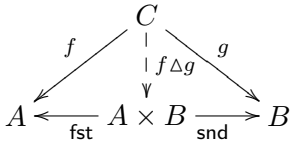
O **produto** de dois tipos é definido como sendo o produto cartesiano:

$$A \times B = \{(x, y) | x \in A, y \in B\}$$

É possível agora definir as funções de projecção e o combinador *split* (que será denotado por $\cdot \vartriangle \cdot$).

$$\begin{array}{l} \mathsf{fst}\ (x, y) = x \\ \mathsf{snd}\ (x, y) = y \end{array} \qquad\qquad (f \vartriangle g)\ x = (f\ x, g\ x)$$

A lei universal do produto pode ser obtida com base no diagrama abaixo. Neste diagrama o produto de dois objectos A e B é um objecto $A \times B$, e as projecções $\mathsf{fst} : A \times B \to A$ e $\mathsf{snd} : A \times B \to B$ são tais que para cada objecto C, $f : C \to A$ e $g : C \to B$ existe apenas uma função de C para $A \times B$ que faz o diagrama comutar (denotada por $f \vartriangle g$, apresentada a tracejado).



A lei universal do produto é então:

$$k = (f \vartriangle g) \Leftrightarrow \begin{cases} \mathsf{fst} \circ k = g \\ \mathsf{snd} \circ k = f \end{cases}$$

Um outro combinador muito útil é o produto de funções. Será denotado por $\cdot \times \cdot$, e é definido da seguinte forma (é dada maior precedência ao operador de composição que a qualquer outro para evitar o uso excessivo de parêntesis):

$$f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}$$

Outra função muito útil que pode ser definida com o operador *split* é a função swap, que troca os elementos de um par.

$$\begin{aligned} &\mathsf{swap} \ : \ A \times B \to B \times A \\ &\mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst} \end{aligned}$$

Existem mais um conjunto de leis sobre produtos que não serão abordadas neste trabalho.

A **soma** será aqui vista, tal como na maioria das linguagens funcionais, como uma união etiquetada de dois conjuntos ao qual é adicionado um novo elemento *bottom*.

$$A + B = (\{0\} \times A) \cup (\{1\} \times B) \cup \{\bot_{A+B}\}$$

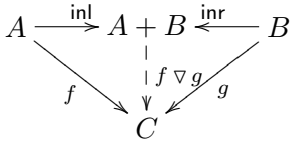
De forma análoga aos produtos, existem as funções injecção e o combinador *either* (que será denotado por $\cdot \triangledown \cdot$). Estes são definidos da seguinte forma:

$$\begin{aligned} \mathsf{inl}\ x &= (0, x) \\ \mathsf{inr}\ x &= (1, x) \end{aligned} \qquad \begin{aligned} (f \triangledown g)\ \bot &= \bot \\ (f \triangledown g)\ (0, x) &= f\ x \\ (f \triangledown g)\ (1, x) &= g\ x \end{aligned}$$

A lei universal da soma pode ser definida como se apresenta abaixo.

$$k = f \triangledown g \Leftrightarrow \begin{cases} k \circ \mathsf{inl} = g \\ k \circ \mathsf{inr} = f \end{cases}$$

Esta lei pode-se verificar por existir uma única função de $A + B$ para C que faz o seguinte diagrama comutar:



É possível agora, como foi feito com o produto, tornar a soma num bifunctor através da introdução do combinador soma $(\cdot + \cdot)$ definido abaixo.

$$f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g$$

A **exponenciação** do tipo B para o tipo A é definido como o conjunto de todas as funções com domínio A e contra-domínio B:

$$B^A = \{f \mid f : A \to B\}$$

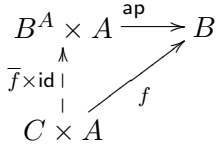
Às exponenciações podem ser associados os combinadores *aplica* e *curry* (denotado por $\overline{\cdot}$).

$$\begin{aligned} \mathsf{ap}\ (f, x) &= f\ x \\ \overline{f}\ x\ y &= f\ (x, y) \end{aligned}$$

Como feito anteriormente com o produto e a soma, é possível definir a lei universal da exponenciação da seguinte forma:

$$k = \overline{f} \Leftrightarrow \mathsf{ap} \circ (f \times \mathsf{id})$$

Esta pode ser observada no diagrama abaixo, onde se verifica que para um objecto C qualquer e $f : C \times A \to B$, $\overline{f}$ é a única função de C para B^A que faz o diagrama comutar.



O combinador exponenciação permite então tornar esta operação num functor:

$$f^A = \overline{f \circ \mathsf{ap}}$$

É possível agora definir funções que distribuem o produto sobre a soma num estilo *point-free*, e que se encontrarão nos capítulos posteriores.

$$
\begin{aligned}
&\mathsf{distl} \;:\; (A+B)\times C \to (A\times C)+(B\times C)\\
&\mathsf{distl} = \mathsf{ap}\circ((\overline{\mathsf{inl}}\,\triangledown\,\overline{\mathsf{inr}})\times \mathsf{id})\\
&\mathsf{distr} \;:\; A\times(B+C) \to (A\times B)+(A\times C)\\
&\mathsf{distr} = (\mathsf{swap}+\mathsf{swap})\circ\mathsf{distl}\circ\mathsf{swap}
\end{aligned}
$$

Os elementos de um objecto A podem ser representados por funções do tipo $1 \to A$, às quais se chamam **pontos**. Dado um elemento $x \in A$, o seu ponto será denotado por $\underline{x}$.

Em particular, as funções também possuem um ponto. Dada uma função $f : A \to B$, o seu ponto será $\underline{f} : 1 \to B^A$ (da mesma forma é possível definir uma função através do seu ponto). Utilizando os combinadores já definidos é possível construir as funções que efectuam estas conversões da seguinte forma:

$$
\begin{aligned}
&\underline{f} = \overline{f\circ\mathsf{snd}}\\
&f = \mathsf{ap}\circ(\underline{f}\circ!\;\triangle\;\mathsf{id})
\end{aligned}
$$

Para concluir esta secção introduz-se agora o conceito de **guarda**.

Os booleanos podem ser vistos como $\mathsf{Bool} = 1 + 1$, onde

$$
\begin{array}{ll}
\mathsf{true} \;:\; 1 \to \mathsf{Bool} & \mathsf{false} \;:\; 1 \to \mathsf{Bool}\\
\mathsf{true} = \mathsf{inl} & \mathsf{false} = \mathsf{inr}
\end{array}
$$

A negação pode então ser implementada pela função coswap, onde:

$$
\begin{aligned}
&\mathsf{coswap} \;:\; A+B \to B+A\\
&\mathsf{coswap} = \mathsf{inr}\,\triangledown\,\mathsf{inl}
\end{aligned}
$$

Para facilitar então o tratamento em *point-free* de expressões condicionais, é útil definir-se o combinador *guarda*, para um dado um predicado $p : A \to Bool$.

$$
\begin{aligned}
&p? \;:\; A \to A+A\\
&p? = (\mathsf{fst}+\mathsf{fst})\circ\mathsf{distr}\circ(\mathsf{id}\;\triangle\;p)
\end{aligned}
$$

2.2 Tipos recursivos

Os tipos de dados recursivos caracterizam-se por serem utilizados na sua própria definição. Alguns exemplos de tipos recursivos definidos em *Haskell* são:

```
data Nat = Zero | Succ Nat
data List a = Nil | Cons a (List a)
```

O primeiro é um tipo monomórfico que representa números naturais, e o segundo é polimórfico e representa listas de um tipo arbitrário.

Os construtores com mais que um argumento serão aqui tratados na sua versão *uncurried*, onde os argumentos serão agrupados em pares aninhados à direita. Obtemos então as seguintes declarações.

$$\begin{array}{ll} \mathsf{zero} : 1 \to \mathsf{Nat} & \mathsf{nil} : 1 \to \mathsf{List}A \\ \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} & \mathsf{cons} : A \times \mathsf{List}A \to \mathsf{List}A \end{array}$$

É necessário agora introduzir o conceito de **functor**. Um *functor* F é um mapeamento que associa tipos a tipos e funções a funções, tal que

$$\begin{array}{l} F\ f : F\ A \to F\ B \quad \Leftarrow \quad f : A \to B \\ F\ (f \circ g) = F\ f \circ F\ g \\ F\ \mathsf{id}_A = \mathsf{id}_{F\ A} \end{array}$$

Entende-se por um *bifunctor* $\star$ uma correspondência de pares de categorias para uma categoria. Neste caso um bifunctor mapeia pares de tipos para tipos, e pares de funções para funções, de tal forma que as condições seguintes se verifiquem:

$$\begin{array}{l} f \star g : A \star B \to C \star D \quad \Leftarrow \quad f : A \to C \wedge g : B \to D \\ (f \circ g) \star (h \circ k) = (f \star h) \circ (g \star k) \\ \mathsf{id}_A \star \mathsf{id}_B = \mathsf{id}_{A \star B} \end{array}$$

Dados dois monofunctores F e G, e um bifunctor $\star$, pode ser definido um novo monofunctor $F \hat{\star} G$ através do *lift* de $\star$, definido da seguinte forma:

$$\begin{array}{l} (F \hat{\star} G)\ A = (F\ A) \star (G\ A) \\ (F \hat{\star} G)\ f = (F\ f) \star (G\ f) \end{array}$$

Voltando aos tipos recursivos, verifica-se que todos os construtores partilham o mesmo codomínio. Por este motivo é possível usar o combinador *either* para agrupar todos os combinadores numa única declaração. Para o caso dos naturais, por exemplo, ter-se-ia:

$$\mathsf{zero} \triangledown \mathsf{succ} : 1 + Nat \to Nat$$

Como o domínio é uma expressão que envolve também o tipo de destino, pode recorrer-se ao conceito de functor para factorizar este tipo. A representação agrupada dos construtores de um tipo de dados T será denotada por in_T, e o *functor de base* que contém a assinatura por F_T. Desta forma o tipo de in_T é sempre $\mathsf{F}_T \to T$. Nos tipos de dados polimórficos será omitida o tipo das variáveis, apenas para tornar a leitura mais fácil.

$$\begin{array}{ll} \mathsf{F}_{\mathsf{Nat}} = \underline{1} \hat{+} \mathsf{Id} & \mathsf{F}_{\mathsf{List}} = \underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id} \\ \mathsf{in}_{\mathsf{Nat}} = \mathsf{zero} \triangledown \mathsf{succ} & \mathsf{in}_{\mathsf{List}} = \mathsf{nil} \triangledown \mathsf{cons} \end{array}$$

Em rigor, um tipo de dados recursivo é então definido pelo ponto fixo do seu *functor de base*. Dado um functor de base F, existe um único tipo de dados μF e duas únicas funções $\mathsf{in}_{\mu F} : F(\mu F) \to \mu F$ e $\mathsf{out}_{\mu F} : \mu F \to F(\mu F)$ que são a inversa uma da outra.

$$\mathsf{in}_{\mu F} \circ \mathsf{out}_{\mu F} = \mathsf{out}_{\mu F} \circ \mathsf{in}_{\mu F} = \mathsf{id}_{\mu F}$$

Os naturais e as listas podem assim definir-se da seguinte forma:

$$\mathsf{Nat} = \mu(\mathsf{F}_{\mathsf{Nat}}) \qquad\qquad \mathsf{List}\ A = \mu(\mathsf{F}_{\mathsf{List}})$$

Desta forma, as funções in e out testemunham o seguinte isomorfismo:

$$\mu F \underset{\mathsf{in}_{\mu F}}{\overset{\mathsf{out}_{\mu F}}{\underset{\cong}{\rightleftarrows}}} F(\mu F)$$

Resta agora ver como definir os destrutores – as funções out. Dado o predicado $\mathsf{iszero} : \mathsf{Nat} \to \mathsf{Bool}$ que testa se um natural é zero, e uma função $\mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat}$ que calcula o predecessor, temos:

$$\begin{array}{l} \mathsf{out}_{\mathsf{Nat}} \ : \ \mathsf{Nat} \to 1 + \mathsf{Nat} \\ \mathsf{out}_{\mathsf{Nat}} = (! + \mathsf{pred}) \circ \mathsf{iszero?} \end{array}$$

De forma análoga para listas, dado o predicado $\mathsf{isnil} : \mathsf{List}A \to \mathsf{Bool}$ e os destrutores $\mathsf{head} : \mathsf{List}A \to A$ e $\mathsf{tail} : \mathsf{List}A \to \mathsf{List}A$ temos então:

$$\begin{array}{l} \mathsf{out}_{\mathsf{List}} \ : \ \mathsf{List} \to 1 + A \times \mathsf{List} \\ \mathsf{out}_{\mathsf{List}} = (! + (\mathsf{head} \vartriangle \mathsf{tail})) \circ \mathsf{isnil?} \end{array}$$

2.3 Padrões recursivos

Exprimir funções utilizando padrões de recursividade é vantajoso por tornar possível a aplicação de várias leis já conhecidas de cálculo e transformação de programas. Seguir-se-ão alguns padrões de recursividade importantes para este trabalho.

Catamorfismos Um catamorfismo consiste numa iteração, onde tipos de dados são "*consumidos*"através da substituição dos seus construtores por funções arbitrárias. Muitas vezes é também referido como *fold*. Em *Haskell*, por exemplo, já se encontra definido o padrão do catamorfismo de listas através da função `foldr`.

Dada uma função $g : F\ A \to A$, é possível definir o catamorfismo de g ($(\!|g|\!)$, usando a notação com parêntesis, segundo a tradição Holandesa), que itera sobre um tipo de dados μF, da seguinte forma:

$$
\begin{aligned}
&(\!|g|\!)_{\mu F}\ :\ \mu F \to A \\
&(\!|g|\!)_{\mu F} = g \circ F\ (\!|g|\!)_{\mu F} \circ \mathsf{out}_{\mu F}
\end{aligned}
$$

de acordo com o diagrama que se segue.

$$
\begin{array}{ccc}
\mu F & \xrightarrow{\mathsf{out}_{\mu F}} & F\ (\mu F) \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle F\ (\!|g|\!)} \\
A & \xleftarrow[g]{} & F\ A
\end{array}
$$

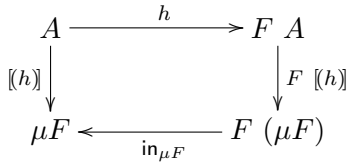
A função g é aqui designada por *gene* do catamorfismo.

Anamorfismos O padrão de recursividade dual à iteração é uma maneira de produzir valores de um determinado tipo de dados recursivo. É também referido como *unfold*.

Dada uma função $h : A \to F\ A$, é possível definir um anamorfismo para qualquer tipo de dados recursivo da seguinte forma (continuará a ser usada a notação com parêntesis).

$$
\begin{aligned}
&[\!(h)\!]_{\mu F}\ :\ A \to \mu F \\
&[\!(h)\!]_{\mu F} = \mathsf{in}_{\mu F} \circ F\ [\!(h)\!] \circ h
\end{aligned}
$$

de acordo com o diagrama que se segue.

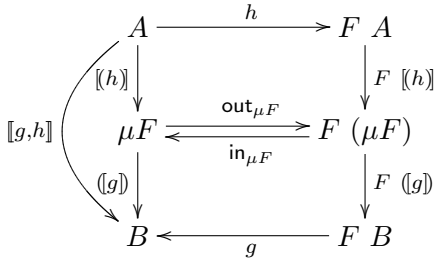


Neste caso a função h é designada por *gene* do anamorfismo.

Hilomorfismos Os hilomorfismos foram originalmente introduzidos em [FM91]. Dado um functor F e funções $g : F\ B \to B$ e $h : A \to F\ A$, é possível definir um hilomorfismo da seguinte forma (usando a notação com parêntesis).

$$\begin{aligned}&[\![g, h]\!]_{\mu F}\ :\ A \to B\\&[\![g, h]\!]_{\mu F} = (\!|g|\!)_{\mu F} \circ [\!(h)\!]_{\mu F}\end{aligned}$$

De acordo com o seguinte diagrama.



Note-se que um hilomorfismo também pode ser definido por um ponto fixo da seguinte forma:

$$[\![g, h]\!]_{\mu F} = \mu(\lambda f\ .\ g \circ F f \circ h)$$

É também possível definir um catamorfismo e um anamorfismo como casos particulares de hilomorfismos:

$$\begin{aligned}(\!|g|\!)_{\mu F} &= [\![g, \mathsf{out}_{\mu F}]\!]_{\mu F}\\[\!(h)\!]_{\mu F} &= [\![\mathsf{in}_{\mu F}, h]\!]_{\mu F}\end{aligned}$$

Paramorfismos Trata-se de uma variação dos catamorfismos. Enquanto estes codificam a iteração, os paramorfismos codificam a noção de recursão primitiva [Mee92]. Isto significa que, ao contrário do catamorfismo, é também passado como argumento ao gene o objecto ao qual é aplicado a recursividade, e não apenas o seu resultado.

Dada uma função $g : F\ (A \times (\mu F)) \to A$, um paramorfismo pode ser definido genericamente da seguinte forma.

$$\begin{aligned} &(\!|g|\!)_{\mu F}\ :\ \mu F \to A \\ &(\!|g|\!)_{\mu F} = g \circ F\ ((\!|g|\!)_{\mu F}\ \vartriangle\ \mathsf{id}) \circ \mathsf{out}_{\mu F} \end{aligned}$$

de acordo com o seguinte diagrama.

$$\begin{array}{ccc} \mu F & \xrightarrow{\mathsf{out}_{\mu F}} & F\ (\mu F) \\ {\scriptstyle (\!|g|\!)_{\mu F}}\downarrow & & \downarrow{\scriptstyle F\ ((\!|g|\!)_{\mu F} \vartriangle \mathsf{id})} \\ A & \xleftarrow[g]{} & F\ (A \times (\mu F)) \end{array}$$

Neste caso o gene já é um pouco mais elaborado que no caso dos catamorfismos. Note-se que também os paramorfismos podem ser escritos como hilomorfismos, da seguinte forma.

$$(\!|g|\!)_{\mu F} = [\![g, F\ (\mathsf{id}\ \vartriangle\ \mathsf{id}) \circ \mathsf{out}_{\mu F}]\!]_{\mu(F \circ (\mathsf{Id} \hat{\times} \underline{\mu F}))}$$

3 Pointless

A biblioteca Pointless [Cun04] enquadra-se no projecto PURe, e encontra-se actualmente numa fase terminal de desenvolvimento. Esta pode ser encontrada nas *UMinho Haskell Libraries*[2].

Trata-se de uma biblioteca *Haskell* que torna possível a programação ***point-free*** com padrões de recursividade. Apesar de utilizar algumas extensões menos comuns da linguagem, permite utilizar uma sintaxe quase idêntica à apresentada no capítulo 2.

A biblioteca em questão é já bastante completa. No entanto, aqui apenas serão desenvolvidos os aspectos mais relevantes para este trabalho, como se verá em seguida.

[2] http://wiki.di.uminho.pt/twiki/bin/view/PURe/PUReSoftware#The_UMinho_Haskell_Libraries

3.1 Implementação de conceitos básicos

Muitas das funções primitivas, combinadores básicos e tipos apresentados no capítulo 2 fazem já parte do prelúdio do Haskell 98, como é o caso da composição de funções e da identidade (definidas como `(.)` e `id`, respectivamente).

Existe já definido um *bottom* polimórfico (designado `undefined`), também acessível com o nome `_L`, por ser muito utilizado. Foi então definido um tipo de dados sem construtores – `One` –, cujo único habitante é `_L`. Para converter elementos para "points" foram definidas as funções `bang :: a -> One` e `pnt :: a -> One -> a` (`bang` foi também definida como `(!)`).

Para o tratamento de produtos encontram-se já definidos os destrutores `fst` e `snd`, que extraem o primeiro e segundo elemento de um par, respectivamente. Foram então definidos os combinadores infixos *split* e *produto* (respectivamente, `(/\)` e `(><)`).

Relativamente às somas, foi utilizado o tipo de dados `Either` já definido em *Haskell*. Foram então definidos *atalhos* para os construtores: `inl` para representar o construtor `Left` e `inr` para o construtor `Right`. O combinador `either` encontra-se também no prelúdio do Haskell 98, mas foi definido o atalho infixo `(/\)` para este operador. Foi também introduzida a soma infixa – `(-|-)`.

As exponenciações foram modeladas em *Haskell* pelo tipo funcional `->`. Já existe a função `curry`, que transforma uma função que recebe dois argumentos noutra equivalente que recebe os dois argumentos agrupados num par. O combinador *aplicação*, que aplica o primeiro elemento de um par ao segundo, foi definida com o nome `app`.

Finalmente, foi definido um combinador de guarda que opera sobre os booleanos do *Haskell*. A sua definição é a seguinte:

```
(?) :: (a -> Bool) -> a -> Either a a
p ? x = if p x then inl x else inr x
```

É agora possível, utilizando estes combinadores, definir alguns dos isomorfismos mais usados exactamente como foi feito no capítulo 2.

```
swap :: (a,b) -> (b,a)
swap = snd /\ fst

distl :: (Either a b, c) -> Either (a,c) (b,c)
distl = app . ((curry inl \/ curry inr) >< id)
```

3.2 Functores Explícitos

Uma forma bem conhecida de implementar em *Haskell* versões genéricas de padrões de recursividade é definindo tipos de dados explicitamente como pontos fixos de functores. O ponto fixo de functores é implementado da seguinte forma:

```
newtype Functor f => Mu f = Mu {unMu :: f (Mu f)}
```

É então introduzida uma noção de equivalência estrutural entre tipos em *Haskell*. Isto é feito através da classe:

```
class (Functor f) => FunctorOf f d | d -> f
   where inn' :: f d -> d
         out' :: d -> f d
```

Esta classe permite estabelecer a equivalência estrutural entre um tipo de dados `d` e um functor `f`. Por questões práticas, foi definida uma forma de descrever functores usando um conjunto fixo de combinadores, em vez de tipos de dados arbitrários, e para estes foram definidas as instâncias da classe `Functor` apropriadas.

Os combinadores que permitem descrever functores definidos na biblioteca Pointless são:

- `Id` – identidade;
- `Const t` – constante `t`;
- `g:+:h` – `g` ou `h`;
- `g:*:h` – par (`g`,`h`);
- `g:@:h` – `g` após `h`.

No caso das somas os construtores são `Inl` e `Inr`, que representam a injecção à esquerda e à direita, respectivamente, e no caso da composição o construtor é `Comp`.

É possível então definir o tipo `Int` (interpretado como o tipo dos naturais) como instância da classe `FunctorOf`, para o functor $\mathsf{F}_{Nat} = \underline{1}\hat{+}\mathsf{Id}$ (descrito na secção 2.2) da seguinte forma.

```
instance FunctorOf (Const One :+: Id) Int
   where inn' (Inl (Const _)) = 0
         inn' (Inr (Id n))    = n+1
         out' 0     = Inl (Const _L)
         out' (n+1) = Inr (Id n)
```

Foi então implementada coerção implícita entre elementos definidos com os combinadores de functores e os tipos esperados pré-definidos em *Haskell* através da seguinte classe:

```
class Rep a b | a -> b
   where to :: a -> b
         from :: b -> a
```

Foram definidas instâncias desta classe para os combinadores que permitem descrever functores descritos acima.

Feito isto, as funções `inn` e `out` foram definidas da seguinte forma:

```
out :: (FunctorOf f d, Rep (f d) fd) => d -> fd
out = to . out'

inn :: (FunctorOf f d, Rep (f d) fd) => fd -> d
inn = inn' . from
```

onde `fd` é um tipo de um combinador que permite descrever functores, como descrito anteriormente, e as funções `inn'` e `out'` são as definidas na classe `FunctorOf`.

Para os naturais definidos acima como instância de `FunctorOf`, por exemplo, obtêm-se os seguintes resultados:

```
> out (0::Int)
Left _L
> inn (Right 3) :: Int
4
```

Finalmente foram definidas um conjunto de ferramentas politípicas que permitem operar sobre tipos de dados cuja instância de `FunctorOf` esteja definida.

- `pmap` – map politípico
- `hylo` – hilomorfismo
- `cata` – catamorfismo
- `ana` – anamorfismo
- `para` – paramorfismo

3.3 Ligação a este projecto

Nos capítulos seguintes serão definidos tipos de dados que representam expressões de linguagens descritas posteriormente em *point-free* e *point-wise*. Para estes tipos de dados são definidas funções que convertem para expressões em *Haskell* e vice-versa. No entanto, o *Haskell* considerado já inclui as bibliotecas Pointless.

Em vez de serem utilizadas as funções `inn` e `out`, são utilizadas outras semelhantes da biblioteca Pointless, `inN` e `ouT`, que recebem um argumento que define o tipo que o operador trata. Escreve-se:

`inN (_L :: tipo ) x`	em vez de	`inn x :: tipo`
`ouT (_L :: tipo ) x`	em vez de	`out x :: tipo`

Outra função que se encontra na biblioteca Pointless e que não foi referida acima é `fix`, que permite calcular o ponto fixo de uma função.

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

4 *Point-free*

A primeira linguagem *point-free* definida neste trabalho admite tipos algébricos arbitrários construídos como pontos fixos de functores, e recursividade primitiva sobre estes tipos na forma de paramorfismos.

4.1 Definição de um termo

Linguagem *point-free* Utilizando a notação introduzida no capítulo 2, os termos *point-free* encontram-se definidos abaixo, sobre tipos de dados quaisquer, denotados por uma string (que tenham um functor associado).

tipos:

$$A, B ::= string \mid A \to B \mid A \times B \mid 1$$

termos:

$$\begin{aligned} M, N ::= {} & \mathsf{bang} \mid \mathsf{id} \mid \mathsf{ap} \mid \overline{M} \mid M \circ N \mid M \triangledown N \mid M \vartriangle N \\ & \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{in}_{string} \mid \mathsf{out}_{string} \\ & \mid (\!|M|\!)_{string} \mid \mathsf{fix} \mid macro \end{aligned}$$

Note-se que há uma diferença relativamente à notação introduzida no capítulo 2: as funções in e out, assim como o paramorfismo, são aqui indexadas pelo tipo de dados e não pelo ponto fixo do functor de base associado a este. É assumido que tal functor existe.

Macros Analisando as funções que definem a linguagem *point-free* verifica-se que foi introduzido o conceito de *macro*. Uma macro é apenas um nome que representa um termo já conhecido.

A notação *point-free* facilmente se torna *ilegível*, devido ao grande número de funções que se encadeiam. Além disso, no tratamento de expressões *point-free* há certas noções/funções que já sabemos escrever em *point-free* mas que têm de ser reconhecidas com alguma frequência.

Um dos exemplos de uma função que pode aparecer algumas vezes é a seguinte:

$$\mathsf{distr} :: (C, A + B) \to (C, A) + (C, B)$$

que se pode escrever em *point-free* como:

$$\mathsf{distr} = \mathsf{ap} \circ ((\overline{\mathsf{inl} \circ \mathsf{swap}} \triangledown \overline{\mathsf{inr} \circ \mathsf{swap}}) \times \mathsf{id}) \circ \mathsf{swap}$$

onde swap é ainda outra macro que troca os elementos de um par, descrita no capítulo 2.

Deste modo, para evitar o aparecimento de blocos de funções cujo significado é conhecido na expressão final, e principalmente para tornar a escrita de código mais *legível* e prática, optou-se por adicionar a possibilidade de guardar o nome de eventuais macros conhecidas que possam aparecer.

Na altura de converter um termo *point-free* (como se verá posteriormente) para uma expressão em *Haskell*, a macro será apenas reconhecida como uma variável com o seu nome, como se verá na secção 4.3. No entanto seria possível assumir a existência de um conjunto predefinido de macros que seriam substituídas pela sua definição, caso a legibilidade da expressão final não fosse considerada prioritária.

Definição em *Haskell* Um termo *point-free* é definido em *Haskell* pelo seguinte tipo de dados:

```
-- *Definition of the data type

data Term = BANG                -- ^Constant function that returns Unit
          | ID                  -- ^Identity
          | APP                 -- ^Apply
          | Curry Term          -- ^Curry
          | Term :.: Term       -- ^Composition
          | Term :\/: Term      -- ^Either
          | Term :/\: Term      -- ^Split
          | FST                 -- ^Point-free first
          | SND                 -- ^Point-free second
          | INL                 -- ^Point-free left injection
          | INR                 -- ^Point-free right injection
          | IN String           -- ^Injection on a specified type
          | OUT String          -- ^Gets the functor
          | PARA String Term    -- ^Paramorphism
          | FIX                 -- ^Fixed point function
          | Macro String        -- ^Macros of frequently used pointfree terms
            deriving Show
```

4.2 As bibliotecas em *Language.Haskell*

Nas versões actuais do GHC e do Hugs é possível encontrar os seguintes módulos:

- `Language.Haskell.Syntax`
- `Language.Haskell.Parser`
- `Language.Haskell.Pretty`

Onde a linguagem *Haskell* é representada numa árvore sintáctica abstracta, e se encontram definidas funções para fazer o *parsing* e para imprimir o código correspondente a uma árvore.

Para este trabalho o tipo de dados (sub-árvore) mais relevante é o `HsExp`, que representa uma expressão em *Haskell*.

Posteriormente, quando for referido que um determinado tipo de dados é convertido para expressões em *Haskell*, é porque é convertido para o tipo de dados *HsExp*, e vice-versa.

4.3 Conversões entre um termo *point-free* e um *HsExp*

A conversão de termos da linguagem *point-free* acima definida para programas *Haskell* é imediata, envolvendo pouca ou nenhuma manipulação da expressão em causa. No sentido inverso a função de conversão é igualmente simples, mas é naturalmente parcial. Em termos práticos, estas funções de conversão têm o tipo $Term \rightarrow HsExp$ e $HsExp \rightarrow Term$ respectivamente, sendo *HsExp* o tipo utilizado para a representação de expressões da linguagem *Haskell* (como referido na secção 4.2).

A relação entre termos *point-free* escritos utilizando diferentes notações encontra-se resumida na tabela 1.

Notação teórica	**Pointfree.Term**	***Haskell***
bang	`BANG`	`bang`
id	`ID`	`id`
ap	`APP`	`app`
$\overline{term}$	`Curry` $[term]$	`curry` $[term]$
$term_1 \circ term_2$	$[term_1]$ `:.:` $[term_2]$	$[term_1]$ `.` $[term_2]$
$term_1 \triangledown term_2$	$[term_1]$ `:\/:` $[term_2]$	$[term_1]$ `\/` $[term_2]$
$term_1 \triangle term_2$	$[term_1]$ `:/\:` $[term_2]$	$[term_1]$ `/\` $[term_2]$
fst	`FST`	`fst`
snd	`SND`	`snd`
inl	`INL`	`Left`

Continua na página seguinte

Tabela 1 – continuação da página anterior

Notação teórica	Pointfree.Term	*Haskell*
inr	`INR`	`Right`
in_{string}	`IN "`*string*`"`	`inN (_L::`*string*`)`
out_{string}	`OUT "`*string*`"`	`ouT (_L::`*string*`)`
$\langle\!\vert term \vert\!\rangle_{string}$	`PARA "`*string*`"` *term*	`para (_L::`*string*`)` [*term*]
fix	`FIX`	`fix`
macro	`Macro "`*macro*`"`	*macro*

Tabela 1: Associação entre várias notações de termos da linguagem *point-free*: notação teórica, tipo de dados definido, expressão em *Haskell*

Ao definir as conversões de e para *HsExp*, define-se a instância de *SubHsExp*, onde se irão definir as funções `toHsExp` e `fromHsExp` (nas secções 4.3 e 4.3).

```
instance SubHsExp Term where
  -- toHsExp :: Term -> HsExp
  -- fromHsExp :: HsExp -> Maybe Term
```

Detalhes de implementação Ao exportar e importar *HsExp*'s foram tomadas algumas decisões:

- :/\: é associado à função infixa `/\` e não à função `split`;
- :\/: é associado à função infixa `\/` e não à função `either`;
- A função `ap` é associada à função `app` em *Haskell*, e não à função escrita com abstracções lambda correspondente.

Existem várias formas possíveis de representar recursividade. Inicialmente foi considerada a possibilidade de utilizar hilomorfismos para representar qualquer tipo de recursividade. Contudo optou-se pela utilização de pontos-fixos, através da função fix definida na biblioteca Pointless (página 15).

No entanto, um ponto fixo também pode ser visto como um hilomorfismo sobre um tipo de dados X, tal que $FX = (A \to A) \times X$, onde $\mathsf{fix} = [\![\mathsf{ap},\ \mathsf{id} \vartriangle \mathsf{id}]\!]_{\mu(A\to A)\times \mathsf{Id}}$.

Mais tarde, foram introduzidos os paramorfismos, que embora não introduzissem nenhum conceito novo, uma vez que podem ser descritos como pontos fixos ou outros hilomorfismos, aumentam a interpretação da expressão em *point-free*.

Conversão para *HsExp* Nesta secção descreve-se a função que converte um termo *point-free* numa expressão *Haskell* (que utiliza funções da biblioteca Pointless).

```
-- Implementation of a pointfree term on a HsExp

toHsExp BANG          = mkVar "bang"
toHsExp ID            = mkVar "id"
toHsExp APP           = mkVar "app"
toHsExp (Curry t1)    = HsApp (mkVar "curry") (mbParen$toHsExp t1)
toHsExp (t1 :.: t2)   = HsInfixApp  (mbParen$toHsExp t1)
                            (mkOp ".") (mbParen$toHsExp t2)
toHsExp (t1 :\/: t2) = HsInfixApp (mbParen$toHsExp t1) (mkOp "\\/")
                              (mbParen$toHsExp t2)
toHsExp (t1 :/\: t2) = HsInfixApp (mbParen$toHsExp t1) (mkOp "/\\")
                              (mbParen$toHsExp t2)

toHsExp FST           = mkVar "fst"
toHsExp SND           = mkVar "snd"
toHsExp INL           = mkCon "Left"
toHsExp INR           = mkCon "Right"


toHsExp (IN str)      = -- inN (_L :: str)
    HsApp (mkVar "inN") (HsParen $ HsExpTypeSig mkLoc
      (mkVar "_L") (HsQualType [] $ HsTyCon$UnQual$HsIdent str))
toHsExp (OUT str)     = -- ouT (_L :: str)
    HsApp (mkVar "ouT") (HsParen $ HsExpTypeSig mkLoc
      (mkVar "_L") (HsQualType [] $ HsTyCon$UnQual$HsIdent str))

toHsExp (PARA str t) = -- para (_L :: str) t
    HsApp (HsApp (mkVar "para") (HsParen $ HsExpTypeSig mkLoc
      (mkVar "_L") (HsQualType [] $ HsTyCon$UnQual$HsIdent str)))
      (mbParen$toHsExp t)
toHsExp FIX         = mkVar "fix"

toHsExp (Macro str)  = mkVar str
```

Conversão simples de *HsExp* para um termo *point-free* Nesta secção descreve-se um *parser* muito simples que consegue reconhecer expressões em *Haskell* que tenham tradução directa para o tipo *point-free*. Esta implementação é ainda incompleta, e pode ser melhorada. Por exemplo, a leitura de macros não é permitida, uma vez que se optou por não restringir a linguagem a um conjunto limitado de macros.

```
-- Simple transformation from HsExp to pointfree

{- It creates a PF term based on a HsExp, but it is a very neive
approach. It is far from being complete and it is not fully correct.
No macros are recognized.
-}
fromHsExp (HsParen e) = fromHsExp e
fromHsExp (HsVar(UnQual(HsIdent "fst")))    = return $ FST
```

```
fromHsExp (HsVar(UnQual(HsIdent "snd")))    = return $ SND
fromHsExp (HsCon(UnQual(HsIdent "Left")))   = return $ INL
fromHsExp (HsCon(UnQual(HsIdent "Right"))) = return $ INR
fromHsExp (HsVar(UnQual(HsIdent "app")))    = return $ APP
fromHsExp (HsVar(UnQual(HsIdent "id")))     = return $ ID
fromHsExp (HsVar(UnQual(HsIdent "bang")))   = return $ BANG
fromHsExp (HsVar(UnQual(HsIdent "fix")))   = return $ FIX

fromHsExp (HsApp (HsVar(UnQual(HsIdent "inN")))
    (HsParen (HsExpTypeSig _ (HsVar(UnQual(HsIdent "_L")))
    (HsQualType [] (HsTyCon (UnQual (HsIdent str)))))))) =
           return $ IN str
fromHsExp (HsApp (HsVar(UnQual(HsIdent "ouT")))
    (HsParen (HsExpTypeSig _ (HsVar(UnQual(HsIdent "_L")))
    (HsQualType [] (HsTyCon (UnQual (HsIdent str)))))))) =
           return $ OUT str
fromHsExp (HsApp (HsApp (HsVar(UnQual(HsIdent "para")))
    (HsParen (HsExpTypeSig _ (HsVar(UnQual(HsIdent "_L")))
    (HsQualType [] (HsTyCon (UnQual (HsIdent str)))))))) exp) =
   do t <- fromHsExp exp
      return $ PARA str t

fromHsExp (HsInfixApp e1 (HsQVarOp(UnQual(HsSymbol "."))) e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :.: t2
fromHsExp (HsInfixApp e1 (HsQVarOp (UnQual (HsSymbol "\\/"))) e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :\/: t2
fromHsExp (HsInfixApp e1 (HsQVarOp (UnQual (HsSymbol "/\\"))) e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :/\: t2
fromHsExp (HsApp (HsVar(UnQual(HsIdent "curry"))) e1) =
   do t1 <- fromHsExp e1
      return $ Curry t1

fromHsExp _ = fail "not a Pointfree term"
```

4.4 Exemplos

Em seguida encontram-se exemplos de expressões escritas utilizando a notação teórica, o tipo de dados `Pointfree.Term` e a sua representação em *Haskell* associada.

1. – $\overline{\mathsf{ap} \circ ((succ \circ \mathsf{fst}) \vartriangle \mathsf{snd})}$
   – `Curry (APP :.: ((Macro "succ":.: FST) :/\: SND))`
   – `curry (app . ((succ . fst) /\ snd))`
2. – $\mathsf{fix} \circ (\overline{(\mathsf{in}_{Int} \circ (\mathsf{inl} \circ \mathsf{bang}))} \vartriangle \mathsf{snd})$
   – `FIX :.: (Curry (((IN "Int") :.: (INL :.: BANG)) :/\: SND))`

- `fix . (curry (((inN (_L :: Int)) . (Left . bang)) /\ snd))`

3. - $(\!|(\mathsf{in}_{[a]} \circ \mathsf{inl} \circ \mathsf{bang}) \triangledown (\mathsf{snd} \circ \mathsf{snd})|\!)_{[a]}$
   - `PARA "[a]"((IN "[a]":.: INL :.: BANG ) :\/: (SND :.: SND))`
   - `para (_L::[a]) (((inN (_L::[a])) . Left . bang) \/ (snd . snd)`

5 Core

A linguagem Core corresponde a um λ-calculus com produtos, co-produtos, recursividade por um operador de um ponto-fixo, e tipos de dados como pontos fixos de functores.

5.1 Definição de um termo

De forma semelhante ao que foi feito para termos *point-free* no capítulo anterior, foi agora definido um termo Core sobre um tipo de dados denotado por uma string (que seja um ponto fixo de functores).

$$A, B ::= \mathit{string} \mid A \to B \mid A \times B \mid 1$$

$$\begin{aligned} M, N, P ::= {} & * \mid x \mid M\ N \mid \lambda x.\ M \mid \langle M, N \rangle \mid \pi_1\ M \mid \pi_2\ M \mid \mathsf{fix}\ M \\ & \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \mathsf{of}\ x \to N; y \to P \\ & \mid \mathsf{in}_{\mathit{string}}\ M \mid \mathsf{out}_{\mathit{string}}\ M \end{aligned}$$

O significado destes termos é:

$*$ – Constante *unit*;
x – Variável;
$M\ N$ – Aplicação de um termo Core a um outro;
$\lambda x.\ M$ – Abstracção: função que dado o nome de uma variável devolve um outro termo Core onde a variável possa ocorrer;
$\langle M, N \rangle$ – Par de termos Core;
$\pi_1\ M$ – Primeiro elemento de um par;
$\pi_2\ M$ – Segundo elemento de um par;
$\mathsf{fix}\ M$ – Ponto fixo de um termo;
$\mathsf{inl}\ M$ – Injecção à esquerda de um elemento;
$\mathsf{inr}\ M$ – Injecção à direita de um elemento;
$\mathsf{case}\ M\ \mathsf{of}\ x \to N; y \to P$ – Uma função que dado um termo verifica se é uma injecção à esquerda ou à direita, e devolve um outro termo onde o conteúdo da injecção possa ocorrer ("*case of*");

$\mathsf{in}_{string}\ M$ – Injecção de um dado termo num tipo de dados com functor associado;
$\mathsf{out}_{string}\ M$ – Extracção do functor de um termo de um tipo de dados;

Note-se que mais uma vez as funções in e out são indexadas pelo tipo de dados e não pelo ponto fixo do functor de base associado a este. É assumido que tal functor existe.

Recursividade Das várias formas de representar a recursividade, optou-se pelo ponto-fixo de um termo, definido da mesma forma que nos termos *point-free*

Definição em *Haskell* Um termo Core é definido em *Haskell* da seguinte forma:

```
-- *Definition of the data type

data Term = Unit                    -- ^Unit
          | Var String              -- ^Variable
          | Term:@:Term             -- ^Aplication
          | Abstr String Term       -- ^Abstraction
          | Term:><:Term            -- ^Pair
          | Fst Term                -- ^Point-wise first
          | Snd Term                -- ^Point-wise second
          | Inl Term                -- ^Point-wise left injection
          | Inr Term                -- ^Point-wise right injection
          | Case Term (String,Term) (String,Term)
                                    -- ^Case of
          | In  String Term         -- ^Injection on a specified type
          | Out String Term         -- ^Extraction of the functor of a specified type
          | Fix Term
                                    -- ^Fixed point
            deriving Show
```

Note-se que o tipo de dados que representa expressões *point-free* definida no capítulo 4 também tem o nome `Term`, o que não traz inconvenientes porque as definições dos termos *point-free* e Core encontram-se em módulos diferentes.

5.2 Conversões entre um termo Core e um *HsExp*

A conversão de termos da linguagem Core acima definida para programas *Haskell* é imediata, envolvendo pouca ou nenhuma manipulação da expressão em causa. No sentido inverso a função de conversão é igualmente simples, mas é naturalmente parcial. Em termos práticos, estas

funções de conversão têm o tipo $Term \rightarrow HsExp$ e $HsExp \rightarrow Term$ respectivamente, sendo *HsExp* o tipo utilizado para a representação de expressões da linguagem *Haskell* (como referido na secção 4.2).

A relação entre termos Core escrito utilizando diferentes notações encontra-se resumida na tabela 3.

Notação Teórica	**Core.Term**	***Haskell***
$*$	Unit	`undefined`; `_L`
x	`Var "`x`"`	x
$\langle term_1, term_2 \rangle$	$[term_1]$ `:><:` $[term_2]$	`(`$[term_1]$`,`$[term_2]$`)`
$\lambda x.\ term$	`Abstr "`x`"` $[term]$	$\backslash x$ `->` $[term]$
$term_1\ term_2$	$[term_1]$ `:@:` $[term_2]$	$[term_1]$ $[term_2]$

case $term_1$ of
$x \rightarrow term_2$;
$y \rightarrow term_3$
	`Case` $[term_1]$
	`("`x`",` $[term_2]$`)`
	`("`y`",` $[term_3]$`)`
		`case` $[term_1]$ `of`
		`Left` x `->` $[term_2]$
		`Right` y `->` $[term_3]$

fst $term$	`Fst` $[term]$	`fst` $[term]$
snd $term$	`Snd` $[term]$	`Snd` $[term]$
inl $term$	`Inl` $[term]$	`Left` $[term]$
inr $term$	`Inr` $[term]$	`Right` $[term]$
$\text{in}_{string}\ term$	`In` $string$ $[term]$	`inN (_L::` $string$ `)` $[term]$
$\text{out}_{string} term$	`Out` $string$ $[term]$	`ouT (_L::` $string$ `)` $[term]$
fix $term$	`Fix` $[term]$	`fix` $[term]$

Tabela 3: Associação entre várias notações de termos da linguagem Core: notação teórica, tipo de dados definido, expressão em *Haskell*

Ao definir as conversões de e para *HsExp*, define-se a instância de *SubHsExp*, onde se irão definir as funções `toHsExp` e `fromHsExp` (nas secções 5.2 e 5.2, respectivamente).

```
instance SubHsExp Term where
  -- toHsExp :: Term -> HsExp
  -- fromHsExp :: HsExp -> Maybe Term
```

Conversão para *HsExp* Nesta secção descreve-se a função que converte um termo Core numa expressão *Haskell* (que utiliza funções da biblioteca Pointless).

```
-- Implementation of a Core term on a HsExp

toHsExp Unit           = mkVar "undefined"
toHsExp (Var str)      = mkVar str
toHsExp (t1 :><: t2)   = HsTuple [toHsExp t1,toHsExp t2]
toHsExp (Fst t)        = HsApp (mkVar "fst") (mbParen$toHsExp t)
toHsExp (Snd t)        = HsApp (mkVar "snd") (mbParen$toHsExp t)
toHsExp (Abstr str t) = HsLambda mkLoc [mkPVar str] (mbParen (toHsExp t))
toHsExp (t1 :@: t2)    = HsApp (mbParen$toHsExp t1) (mbParen$toHsExp t2)
toHsExp (Case t1 (str2,t2) (str3,t3)) =
            HsCase (toHsExp t1)
                [HsAlt mkLoc (HsPApp (UnQual$HsIdent "Left") [mkPVar str2])
                 (HsUnGuardedAlt$mbParen$toHsExp t2) [],
                 HsAlt mkLoc (HsPApp (UnQual$HsIdent "Right") [mkPVar str3])
                 (HsUnGuardedAlt$mbParen$toHsExp t3) []]
toHsExp (Inl t)        = HsApp (mkCon "Left") (mbParen$toHsExp t)
toHsExp (Inr t)        = HsApp (mkCon "Right") (mbParen$toHsExp t)

toHsExp (In typ term) =
      HsApp (HsApp (mkVar "inN") (HsParen (HsExpTypeSig mkLoc
      (mkVar "_L") (HsQualType [] (HsTyCon (UnQual (HsIdent typ)))))))
      (mbParen$toHsExp term)

toHsExp (Out typ term) =
      HsApp (HsApp (mkVar "ouT") (HsParen (HsExpTypeSig mkLoc
      (mkVar "_L") (HsQualType [] (HsTyCon (UnQual (HsIdent typ)))))))
      (mbParen$toHsExp term)

toHsExp (Fix term) = HsApp (mkVar "fix")  (mbParen$toHsExp term)
```

Conversão de *HsExp* para um termo Core Nesta secção descreve-se um *parser* muito simples que consegue reconhecer expressões em *Haskell* que tenham tradução directa para o tipo Core.

```
-- Simple transformation from HsExp to Core

fromHsExp (HsParen e) = fromHsExp e

-- unit -> "undefined" or "_L"
fromHsExp (HsVar(UnQual(HsIdent "undifined"))) = return $ Unit
fromHsExp (HsVar(UnQual(HsIdent "_L")))        = return $ Unit

-- Constants
fromHsExp (HsApp (HsApp (HsVar (UnQual (HsIdent "inN"))) (HsParen
     (HsExpTypeSig _ (HsVar (UnQual (HsIdent "_L"))) (HsQualType []
     typ)))) exp) =
  do fromHsExp exp >>= return . In (prettyPrint typ)
fromHsExp (HsApp (HsApp (HsVar (UnQual (HsIdent "ouT"))) (HsParen
     (HsExpTypeSig _ (HsVar (UnQual (HsIdent "_L"))) (HsQualType []
     typ)))) exp) =
  do fromHsExp exp >>= return . Out (prettyPrint typ)

-- Recursion
fromHsExp (HsApp (HsVar (UnQual (HsIdent "fix"))) exp) =
```

```
  do term <- fromHsExp exp
     return $ Fix term

fromHsExp (HsVar(UnQual(HsIdent str)))     = return $ Var str

fromHsExp (HsTuple [e1,e2]) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :><: t2
fromHsExp (HsApp (HsVar (UnQual (HsIdent "fst"))) e) =
   do t <- fromHsExp e
      return $ Fst t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "snd"))) e) =
   do t <- fromHsExp e
      return $ Snd t
fromHsExp (HsApp (HsCon (UnQual (HsIdent "Left"))) e) =
   do t <- fromHsExp e
      return $ Inl t
fromHsExp (HsApp (HsCon (UnQual (HsIdent "Right"))) e) =
   do t <- fromHsExp e
      return $ Inr t
fromHsExp (HsApp e1 e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :@: t2
fromHsExp (HsLambda _ [HsPVar(HsIdent str)] e) =
   do t <- fromHsExp e
      return $ Abstr str t
fromHsExp (HsCase e1
        [HsAlt _ (HsPApp (UnQual(HsIdent "Left"))
         [HsPVar(HsIdent str2)]) (HsUnGuardedAlt e2) _,
         HsAlt _ (HsPApp (UnQual(HsIdent "Right"))
         [HsPVar(HsIdent str3)]) (HsUnGuardedAlt e3) _]) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      t3 <- fromHsExp e3
      return $ Case t1 (str2,t2) (str3,t3)

fromHsExp _ = fail "not a Core term"
```

5.3 Exemplos

Em seguida encontram-se exemplos de expressões escritas utilizando a notação teórica, o tipo de dados `Core.Term` e a sua representação em *Haskell* associada.

1. – $\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ y \rightarrow \mathsf{in}_{Int}\ (\mathsf{inl}\ *);\ z \rightarrow \mathsf{in}_{Int}\ (\mathsf{inr}\ z)$
   – `Abstr "x"(Case (Var "x") ("y",In "Int"(Inl Unit)) ("z",In "Int"(Inr (Var "z"))))`
   – `\x -> case x of Left y -> inN (_L::"Int") (Left \_L); Right z -> inN (_L::"Int") (Right z))`
2. – $\mathsf{fix}(\lambda f.\ (\lambda x.\ \mathsf{in}_{Int}\ (\mathsf{inr}\ x)))$

   - `Fix ( Abstr (Var "f") (Abstr (Var "x") (In "Int"(Inr (Var "x")))))`
   - `fix (\f -> \x -> inN (_L::Int) (Right x))`
3. - $\lambda var.\ \mathsf{fst}\ \langle var, * \rangle$
   - `Abstr "var"(Fst ( (Var "var") :><: Unit))`
   - `\x -> fst (x,_L)`

6 Conversões Core ↔ Point-free

Este capítulo trata o principal problema que envolve este trabalho. Nos capítulos 4 e 5 foram definidas duas linguagens cujas expressões são representadas através de tipos *Haskell* apropriados. Resta agora definir uma possível tradução entre estas duas linguagens. Esta tradução foi definida de acordo com [Cun04].

Este problema foi já estudado em detalhe por Curien, que definiu as traduções entre o lambda calculus com pares (*pointwise*) e os combinadores categóricos que definem uma Categoria Cartesiana Fechada (CCC) [Cur93]. Ele provou ainda a correcção desta tradução, *i.e.*, qualquer igualdade provada num dos sistemas pode ser também provada no outro. Neste trabalho a tradução foi extendida para o conjunto bicartesiano, apesar da sua correcção ainda não ter sido provada.

6.1 *Point-free* para Core

Estas traduções são relativamente simples, como se pode ver na tabela 4. Na referida tabela foi utilizada a notação *Haskell* equivalente (de acordo com os capítulos anteriores) para facilitar a leitura.

$$
\begin{array}{ll}
\Psi(f \circ g) & = \lambda x.\ \Psi(f)\ (\Psi(g)\ x) \\
\Psi(\mathsf{id}) & = \lambda x.\ x \\
\Psi(\mathsf{bang}) & = \lambda x.\ * \\
\Psi(\mathsf{fst}) & = \lambda x.\ \mathsf{fst}\ x \\
\Psi(\mathsf{snd}) & = \lambda x.\ \mathsf{snd}\ x \\
\Psi(f \vartriangle g) & = \lambda x.\ \langle \Psi(f)\ x, \Psi(g)\ x \rangle \\
\Psi(\mathsf{inl}) & = \lambda x.\ \mathsf{inl}\ x \\
\Psi(\mathsf{inr}) & = \lambda x.\ \mathsf{inr}\ x \\
\Psi(f \triangledown g) & = \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ y \rightarrow \Psi(f)\ x;\ z \rightarrow \Psi(g)\ x \\
\Psi(\mathsf{ap}) & = \lambda x.\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x) \\
\Psi(\overline{f}) & = \lambda x.\ (\lambda y.\ \Psi(f)\ (x, y))
\end{array}
$$

$$
\begin{aligned}
\Psi(\mathsf{in}_{Tipo}) &= \lambda x.\ \mathsf{in}_{Tipo}\ x \\
\Psi(\mathsf{out}_{Tipo}) &= \lambda x.\ \mathsf{out}_{Tipo}\ x \\
\Psi(\mathsf{fix}) &= \lambda x.\ \mathsf{fix}\ x \\
\Psi(macro) &= macro
\end{aligned}
$$

Tabela 4: Tradução de *point-free* para Core

No caso da macro, esta passa a ser considerada uma variável na linguagem Core.

Apesar de simples, esta tradução é bastante verbosa, como se pode ver no seguinte exemplo:

$$\Psi(\mathsf{snd} \vartriangle \mathsf{fst}) = \lambda x.\ \langle(\lambda y.\ \mathsf{snd}\ y)\ x, (\lambda y.\ \mathsf{fst}\ y)\ x\rangle$$

Que após um passo de beta redução toma a seguinte forma:

$$\Psi(\mathsf{snd} \vartriangle \mathsf{fst}) = \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x\rangle$$

Point-free* para Core em *Haskell

Aplicando agora as regras definidas na tabela 4 é possível definir uma função em *Haskell* que converta um termo *point-free* num termo Core.

```
-- *Translation from point-free to point-wise

pf2core :: PwPf.Pointfree.Term -> PwPf.Core.Term
pf2core BANG          = Abstr "x" Unit
pf2core ID            = Abstr "x" (Var "x")
pf2core APP           = Abstr "x" ((Fst $ Var"x") :@:
                                   (Snd $ Var "x"))
pf2core (Curry t1)    = Abstr "x" (Abstr "y"
                        ((pf2core t1):@:((Var "y"):><:(Var "x"))) )
pf2core (t1 :.: t2)   = Abstr "x" ((pf2core t1):@:((pf2core t2):@:(Var "x")))
pf2core (t1 :\/: t2)  = Abstr "x" (Case (Var "x")
                        ("y" , (pf2core t1):@:(Var "y"))
                        ("z" , (pf2core t2):@:(Var "z")))
pf2core (t1 :/\: t2)  = Abstr "x" ((pf2core t1):><:(pf2core t2))
pf2core FST           = Abstr "x" (Fst $ Var "x")
pf2core SND           = Abstr "x" (Snd $ Var "x")
pf2core INL           = Abstr "x" (Inl $ Var "x")
pf2core INR           = Abstr "x" (Inr $ Var "x")
pf2core (IN  str)     = Abstr "x" (In  str  $ Var "x")
pf2core (OUT str)     = Abstr "x" (Out str $ Var "x")
pf2core FIX           = Abstr "x" (Fix $ Var "x")
pf2core (Macro x)     = Var x
```

6.2 Core para *point-free*

A tradução de Core para *point-free* é bastante mais sofisticada. Esta tradução foi inspirada na do lambda calculus para a notação de *de Bruijn* [dB72], onde as variáveis são representadas por inteiros que descrevem a *distância* à abstracção a que estão ligadas. Neste caso as variáveis serão denotadas por um par aninhado à esquerda, e uma variável será substituída pelo *caminho* até à posição nesse tuplo.

Numa expressão *point-free* apenas são representadas funções, enquanto que no lambda calculus é possível exprimir valores de qualquer tipo. Desta forma, para que a tradução em causa seja uma função total, Curien definiu-a de tal forma que receba um valor do tipo A e devolva um do tipo $1 \to A$.

Contexto

É necessário começar por introduzir a noção de um contexto, que será representada por um termo Core com um conjunto de pares de variáveis aninhados à esquerda. As variáveis serão aquelas que estão ligadas a cada altura na computação (devido a abstracções *lambda*), e que serão transformadas em sucessivas aplicações das projecções *fst* e *snd* (com o "*apontador*"até à variável em causa).

Assim, em *Haskell* temos:

```
-- | A context will be a left-nested pair, where the most to the
--   right value will be a never used constant
type Context = PwPf.Core.Term
```

Caminho de uma variável Após a definição de um contexto, é necessário agora definir a tradução de uma variável dado um contexto. Isso é feito pela aplicação sucessiva das funções *fst* e *snd* enquanto se percorre o contexto, até ser encontrada a variável desejada, segundo a definição:

$$\mathsf{path}(\langle t, y\rangle, x) = \begin{cases} \mathsf{snd} & se\ x = y \\ \mathsf{path}(t, x) \circ \mathsf{fst} & caso\ contrário \end{cases}$$

Na implementação em *Haskell* optou-se por permitir o aparecimento de variáveis que não estivessem no contexto. Quando isto acontece é devolvida a macro com o nome da variável livre, após o caminho até o início do contexto. Note-se então que esta variável livre também será do tipo $1 \to A$, para algum A.

```
-- | Calculates a "path" to a variable in a context by composing the
--   "fst" and "snd" functions
path :: Context -> PwPf.Core.Term -> PwPf.Pointfree.Term
path (t:><:(Var y)) (Var x)
     | x == y     = SND
     | otherwise = (path t (Var x)) :.: FST
path _ (Var x) = Macro x
```

Conversão A conversão foi feita de acordo com a tabela 5. À semelhança do que foi feito acima, foi utilizada a notação *Haskell* equivalente para facilitar a leitura.

$$
\begin{array}{ll}
\Phi(\Gamma, x) & = \mathsf{path}(\Gamma, x) \\
\Phi(\Gamma, *) & = \mathsf{bang} \\
\Phi(\Gamma, \langle t, u\rangle) & = \Phi(\Gamma, t) \vartriangle \Phi(\Gamma, u) \\
\Phi(\Gamma, \mathsf{fst}\ t) & = \mathsf{fst} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \mathsf{snd}\ t) & = \mathsf{snd} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \lambda x.\ t) & = \overline{\Phi(\langle \Gamma, x\rangle, t)} \\
\Phi(\Gamma, t\ u) & = \mathsf{ap} \circ (\Phi(\Gamma, t) \vartriangle \Phi(\Gamma, u)) \\
\Phi(\Gamma, \mathsf{inl}\ t) & = \mathsf{inl} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \mathsf{inr}\ t) & = \mathsf{inr} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \mathsf{case}\ t\ \mathsf{of}\ x \to u;\ y \to v) = & \\
\quad \mathsf{ap} \circ (\overline{(\Phi(\langle \Gamma, x\rangle, u) \triangledown \Phi(\langle \Gamma, y\rangle, v)) \circ distr} \vartriangle \Phi(\Gamma, t)) & \\
\Phi(\Gamma, \mathsf{in}_{Tipo}\ t) & = \mathsf{in}_{Tipo} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \mathsf{out}_{Tipo}\ t) & = \mathsf{out}_{Tipo} \circ \Phi(\Gamma, t) \\
\Phi(\Gamma, \mathsf{fix}\ t) & = \mathsf{fix} \circ \Phi(\Gamma, t)
\end{array}
$$

Tabela 5: Tradução de Core para *point-free*

Na definição de $\Phi(\Gamma, \mathsf{case}\ t\ \mathsf{of\ Left}\ x\ \texttt{->}\ u;\ \mathsf{Right}\ y\ \texttt{->}\ v)$ encontra-se a macro *distr*, que será do tipo $1 \times (A + B) \to 1 \times A + 1 \times B$. Esta macro é utilizada apenas para o resultado final se tornar de mais fácil compreensão, e encontra-se já definida na secção 4.1.

Conversão em *Haskell* de casos conhecidos Antes da transformação de cada um dos construtores de um termo Core para um termo *point-free*, é verificado se o termo tem algum padrão já conhecido (*e.g.*,

paramorfismo). Desta forma torna-se possível reconhecer algumas estruturas no código *point-free* gerado.

Isto torna-se prático porque existem bibliotecas que geram termos Core relativamente complexos, que se tornam difíceis de ler. No entanto, estes poderiam ser interpretados de forma mais *"inteligente"* do que a tradução directa de cada um dos construtores mais externos.

Para já apenas está a ser efectuado o reconhecimento de paramorfismos de naturais e de listas (possivelmente gerados pela biblioteca BNL, que é introduzida no capítulo 8). Estes paramorfismos não podem ter variáveis livres na função iterativa nem no caso de paragem.

```
-- | Given a Context, it calculates the translation from Core
--   to Point-free
core2pf :: Context -> PwPf.Core.Term -> PwPf.Pointfree.Term

-- paramorphisms for Int
core2pf cont ((((Fix (Abstr r1 (Abstr n1 (Abstr f1 (Abstr z1
    (Case (Out typ (Var n2))
          (_,Var z2)
          (y1,(Var f2 :@: Var y2) :@: (((Var r2 :@: Var y3) :@: Var f3)
           :@: Var z3))))))))) :@: n) :@: f) :@: z)
  | r1==r2 && n1==n2 && f1==f2 && f2==f3 &&
    z1==z2 && z2==z3 && y1==y2 && y2==y3 &&
    isClosed f && isClosed z =
       (PARA typ (unpoint $ g(f,z))) :.: (core2pf cont n)
 where
    unpoint f = APP :.: ((f :.: BANG) :/\: ID)
    g (f,z) = let y = getFV f; x = getFV (z:><:Var y)
              in core2pf Unit $
     Abstr x $
       Case (Var x) ("_",z)
                    (y,f :@: (Snd$Var y) :@: (Fst$Var y))
------------------------------ pred ^ - recursive result ^

-- paramorphisms for [a]
core2pf cont ((((Fix (Abstr r1 (Abstr l1 (Abstr f1 (Abstr z1
    (Case (Out typ (Var l2))
          (_,Var z2)
          (y1,Var f2 :@: (Fst (Var y2)) :@: (Snd (Var y3)) :@:
             (Var r2 :@: (Snd (Var y4)) :@: Var f3 :@: Var z3))))))))
          :@: n) :@: f) :@: z)
  | r1==r2 && l1==l2 && f1==f2 && f2==f3 &&
    z1==z2 && z2==z3 && y1==y2 && y2==y3 && y3==y4 &&
    isClosed f && isClosed z =
       (PARA typ (unpoint $ g(f,z))) :.: (core2pf cont n)
 where
    unpoint f = APP :.: ((f :.: BANG) :/\: ID)
    g (f,z) = let y = getFV f; x = getFV (z:><:Var y)
              in core2pf Unit $
     Abstr x $
       Case (Var x) ("_",z)
              (y,f :@: (Fst$Var y) :@: (Snd$Snd$Var y) :@: (Fst$Snd$Var y))
------------------------------ head ^ ---------- tail ^ --- recursive result ^
```

Conversão em *Haskell* de um termo Core

Aplicando as regras definidas na tabela 5 é possível definir em *Haskell* uma função que converte um termo Core num termo *point-free*, assumindo que a macro *distr* se encontra definida externamente.

```
core2pf _    Unit          = BANG
core2pf cont var@(Var x) = path cont var
core2pf cont (t1:><:t2)   = (core2pf cont t1):/\:(core2pf cont t2)
core2pf cont (Fst t)      = FST :.: (core2pf cont t)
core2pf cont (Snd t)      = SND :.: (core2pf cont t)
core2pf cont (Abstr x t) = Curry (core2pf (cont:><:(Var x)) t)
core2pf cont (t1:@:t2)    = APP :.: ((core2pf cont t1):/\:(core2pf cont t2))
core2pf cont (Inl t)      = INL :.: (core2pf cont t)
core2pf cont (Inr t)      = INR :.: (core2pf cont t)
core2pf cont (Case t (x,u) (y,v)) =
    APP :.: ((Curry (((core2pf (cont:><:(Var x)) u) :\/:
            (core2pf (cont:><:(Var y)) v)) :.: (Macro "distr")))
            :/\: (core2pf cont t))
core2pf cont (In  str t) = IN  str :.: (core2pf cont t)
core2pf cont (Out str t) = OUT str :.: (core2pf cont t)
core2pf cont (Fix t)     = FIX :.: (core2pf cont t)
```

Variáveis livres Há casos acima onde é necessário obter uma variável fresca relativamente às variáveis que ocorrem num termo, e outros onde é necessário verificar se um termo é fechado ou não. Para isso são definidos dois catamorfismos sobre o tipo de dados *point-free*.

```
getFV :: PwPf.Core.Term -> String
getFV = (\x->x++"_").maximum.getVars

getVars :: PwPf.Core.Term -> [String]
getVars (Var str)       = [str]
getVars Unit            = []
getVars (t1:><:t2)      = (getVars t1) ++ (getVars t2)
getVars (Abstr str t)   = str:(getVars t)
getVars (t1:@:t2)       = (getVars t1) ++ (getVars t2)
getVars (Case t1 (str1,t2) (str2,t3)) =
        str1:str2:(getVars t1++getVars t2++getVars t3)
getVars (Fst t)    = getVars t
getVars (Snd t)    = getVars t
getVars (Inl t)    = getVars t
getVars (Inr t)    = getVars t
getVars (In _ t)   = getVars t
getVars (Out _ t) = getVars t
getVars (Fix t)    = getVars t
```

```
isClosed :: PwPf.Core.Term -> Bool
isClosed t = closed t []
 where
  closed Unit _           = True
  closed (Var str) ac     = str `elem` ac
  closed (t1:><:t2) ac    = (closed t1 ac) && (closed t2 ac)
```

```
closed (Abstr str t) ac = closed t (str:ac)
closed (t1:@:t2) ac     = (closed t1 ac) && (closed t2 ac)
closed (Case t1 (str1,t2) (str2,t3)) ac =
   (closed t1 ac) && (closed t2 (str1:ac)) && (closed t3 (str2:ac))
closed (Fst t) ac       = closed t ac
closed (Snd t) ac       = closed t ac
closed (Inl t) ac       = closed t ac
closed (Inr t) ac       = closed t ac
closed (In _ t) ac      = closed t ac
closed (Out _ t) ac     = closed t ac
closed (Fix t) ac       = closed t ac
```

6.3 Exemplos

Recursividade sobre naturais Neste exemplo é convertida para *point-free* a expressão:

```
func = fix (\ f -> \ x ->
           case (ouT (_L::Int) x) of
               Left  z -> inN (_L::[Int]) (Left _L)
               Right y -> inN (_L::[Int]) (Right (inN (_L::Int) (Left _L), f y)))
```

que recebe um natural "n"e devolve uma lista onde o número 0 é replicado "n"vezes.

O código *point-free* gerado é o seguinte:

```
*PwPf.Tester> tCore2pf "func = fix (\\ f -> \\ x -> case (ouT (_L::Int) x) of
     Left z -> inN (_L::[Int]) (Left _L);
     Right y -> inN (_L::[Int]) (Right (inN (_L::Int) (Left _L),f y)))"
module Main (main) where
transf_func
  = fix .
      (curry
         (curry
            (app .
               ((curry
                   ((((inN (_L :: [Int])) . (Left . bang)) \/
                       ((inN (_L :: [Int])) .
                          (Right .
                             (((inN (_L :: Int)) . (Left . bang)) /\
                                (app . (((snd . fst) . fst) /\ snd))))))
                      . distr))
                  /\ ((ouT (_L :: Int)) . snd)))))
```

Aplicando por sua vez a função *point-free* aos argumentos `_L` e `3` obtém-se o seguinte resultado, que corresponde ao esperado:

```
*PwPf.Tester> (fix . (....)) _L 3
[0,0,0]
```

Utilizando a notação teórica, é convertida a expressão:

$$\begin{array}{l}\mathsf{fix}(\lambda f.\ \lambda x.\\ \quad \mathsf{case}(\mathsf{out}_{Int}\ x)\ \mathsf{of}\\ \quad\quad (z, \mathsf{in}_{[Int]}\ (\mathsf{inl}\ *))\\ \quad\quad (y, \mathsf{in}_{[Int]}\ (\mathsf{inr}\ \langle \mathsf{in}_{Int}\ (\mathsf{inl}\ *), f\ y\rangle))\end{array}$$

para:

$$\mathsf{fix} \circ (((\mathsf{ap} \circ ((((((\mathsf{in}_{[Int]} \circ \mathsf{inl} \circ \mathsf{bang}) \triangledown (\mathsf{in}_{Int} \circ (\mathsf{inr} \circ ((\mathsf{in}_{Int} \circ \mathsf{inl} \circ \mathsf{bang}) \vartriangle (\mathsf{ap} \circ ((\mathsf{snd} \circ \mathsf{fst} \circ \mathsf{fst}) \vartriangle \mathsf{snd})))))) \circ distr)) \vartriangle (\mathsf{out}_{Int} \circ \mathsf{snd}))))$$

Nas secções 7.4 e 8.4 é explorado este exemplo, mas utilizando representações de código *pointwise* mais intuitivas.

Recursividade sobre listas Neste exemplo é calculada a dimensão de uma lista. Para isso é convertida a seguinte expressão:

```
func = fix (\ f -> \ x ->
           case (ouT (_L::[a]) x) of
               Left  _ -> inN (_L::Int) (Left _L)
               Right y -> inN (_L::Int) (Right (f (snd y))))
```

que recebe uma lista e devolve um inteiro com o seu comprimento.

O código gerado é o seguinte:

```
*PwPf.Tester> tCore2pf "func = fix (\\ f -> \\ x -> case (out (_L::[a]) x) of
      Left  z -> inN (_L::Int) (Left _L);
      Right y -> inN (_L::Int) (Right (f (snd y))))"
module Main (main) where
transf_func
  = fix .
      (curry
         (curry
            (app .
               ((curry
                   ((((inN (_L :: Int)) . (Left . bang)) \/
                       ((inN (_L :: Int)) .
                          (Right . (app . (((snd . fst) . fst) /\ (snd . snd))))))
                      . distr))
                 /\ ((ouT (_L :: [a])) . snd)))))
```

Aplicando por sua vez a função *point-free* aos argumentos `_L` e `[3,5,1]` obtém-se o seguinte resultado, que corresponde ao resultado esperado:

```
*PwPf.Tester> (fix . (....)) _L [3,5,1]
3
```

Nas secções 7.4 e 8.4 é explorado este exemplo, mas utilizando representações de código *pointwise* mais intuitivas.

7 PCF

Tendo definido a linguagem Core, pretende-se agora extender o âmbito do trabalho para a linguagem PCF, sendo possível definir termos que representem outras expressões *pointwise*, sem introduzir nenhum conceito que não possa ser expresso por um termo Core. Pretende-se que a definição de termos PCF se aproxime muito mais do estilo de programação funcional a que estamos habituados.

É de notar que a linguagem PCF apenas define um conjunto particular de tipos, ao contrário da linguagem Core que permitia a utilização de qualquer tipo ao qual se pudesse associar um functor.

7.1 Definição de um termo

A linguagem PCF PCF significa "Programming with computable functions"(programar com funções computáveis). A linguagem PCF é uma extensão do lambda-calculus com booleanos, naturais e recursividade, introduzida por Dana Scott, em 1969 [Sco93] (ele apenas introduziu o LFC – lógica de funções computáveis –, que inspirou uma nova geração de investigadores, sendo o PCF publicado por Gordon Ploktin em 1977 [Plo77]).

Os tipos podem ser descritos da seguinte forma:

$$A, B ::= \mathit{bool} \mid \mathit{nat} \mid A \to B \mid A \times B \mid 1$$

Os termos são definidos com se segue:

$$\begin{aligned} M, N, P ::= {} & * \mid x \mid M\ N \mid \lambda x.\ M \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \\ & \mid \mathsf{T} \mid \mathsf{F} \mid \mathsf{zero} \mid \mathsf{succ}\ M \mid \mathsf{pred}\ M \\ & \mid \mathsf{iszero}\ M \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \mid \mathsf{Y}\ M \end{aligned}$$

onde as duas primeiras linhas são outros nomes para noções existentes nos termos Core, e $\mathsf{Y}\ M$ é o ponto fixo de M. Os restantes termos permitem efectuar algumas operações com booleanos e naturais, que se inferem com facilidade pelos seus nomes.

Extensões à linguagem PCF A linguagem implementada neste capítulo foi enriquecida com construtores, que permitem a utilização de listas e formas de explicitar recursividade diferentes:

– nil – Lista vazia;

– cons M N – Lista com cabeça M e cauda N;
– head – Primeiro elemento de uma lista;
– tail – Elementos após o primeiro de uma lista;
– isnil – Predicado que verifica se uma lista é vazia;
– letrec $x \leftarrow M,\ N$ – Seja x uma variável que tome o valor de M (onde pode ocorrer recursivamente), é devolvido N (onde x pode ocorrer).

Os tipos são então enriquecidos com listas, e passam a ser descritos da seguinte forma:

$$A, B ::= bool \mid nat \mid list\ A \mid A \to B \mid A \times B \mid 1$$

Funções Parciais Há algumas funções que são parciais:

– pred – calcula o predecessor de um número;
– head – calcula o primeiro elemento de uma lista;
– tail – calcula a cauda de uma lista;

Para o primeiro e terceiro caso foi considerado, respectivamente, que o predecessor de zero é zero, e que a cauda de uma lista vazia é também vazia. A cabeça de uma lista vazia é indefinida, que em *Haskell* corresponderá a devolver a variável "`undefined`", como se verá mais tarde, para evitar erros de tipos.

Definição em *Haskell* Um termo PCF é definido em *Haskell* da seguinte forma:

```
-- *Definition of the data type

data Term = Star                                  -- ^Unit
          | V String                              -- ^Variable
          | Lam String PwPf.PCF.Term              -- ^Lambda abstraction
          | PwPf.PCF.Term :-: PwPf.PCF.Term       -- ^Aplication
          | PwPf.PCF.Term :&: PwPf.PCF.Term       -- ^Pair
          | Pi1 PwPf.PCF.Term                     -- ^Projection of the first element
          | Pi2 PwPf.PCF.Term                     -- ^Projection of the second element
          | T                                     -- ^Constant True
          | F                                     -- ^Constant False
          | Z                                     -- ^Constant Zero
          | Suc PwPf.PCF.Term                     -- ^Successor
          | Pred PwPf.PCF.Term                    -- ^Predecessor
          | IsZ PwPf.PCF.Term                     -- ^is zero?
          | Ite PwPf.PCF.Term PwPf.PCF.Term PwPf.PCF.Term -- ^if then else
          | Y PwPf.PCF.Term                       -- ^Fix point

          | N                                       -- ^Empty list
          | PwPf.PCF.Term ::: PwPf.PCF.Term         -- ^Constructor for lists
```

```
  | Hd PwPf.PCF.Term                          -- ^Head of a list
  | Tl PwPf.PCF.Term                          -- ^Tail of a list
  | IsN  PwPf.PCF.Term                        -- ^is the list empty?
  | Letrec String PwPf.PCF.Term PwPf.PCF.Term
                                              -- ^recursive let
  deriving Show
```

7.2 Conversões entre um termo PCF e um *HsExp*

A conversão de termos da linguagem PCF acima definida para programas *Haskell* é imediata, envolvendo pouca ou nenhuma manipulação da expressão em causa. No sentido inverso a função de conversão é igualmente simples, mas é naturalmente parcial. Em termos práticos, estas funções de conversão têm o tipo $Term \rightarrow HsExp$ e $HsExp \rightarrow Term$ respectivamente, sendo *HsExp* o tipo utilizado para a representação de expressões da linguagem *Haskell* (como referido na secção 4.2).

A relação entre termos *point-free* escrito utilizando diferentes notações encontra-se resumida na tabela 6.

Notação teórica	**PCF.Term**	***Haskell***
$*$	`Star`	`undefined`; `_L`
x	`V "`x`"`	x
$\lambda x.\ term$	`Lam "`x`"` $[term]$	$\backslash x$ `->` $[term]$
$term_1\ term_2$	$[term_1]$ `:-:` $[term_2]$	$[term_1]$ $[term_2]$
$\langle term_1, term_2 \rangle$	$[term_1]$ `:&:` $[term_2]$	`(`$[term_1]$`,`$[term_2]$`)`
fst $term$	`Pi1` $[term]$	`fst` $[term]$
snd $term$	`Pi2` $[term]$	`snd` $[term]$
T	`T`	`True`
F	`F`	`False`
Z	`Z`	`0`
succ $term$	`Suc` $[term]$	`succ` $[term]$; *natural_number*
pred $term$	`Pred` $[term]$	`pred` $[term]$
iszero $term$	`IsZ` $[term]$	`(==0)` $[term]$
if $term_1$ then tem_2 else $term_3$	`Ite` $[term_1]$ $[term_2]$ $[term_3]$	`if` $[term_1]$ `then` $[term_2]$ `else` $[term_3]$
Y $term$	`Y` $[term]$	`fix` $[term]$
nil	`N`	`[]`
cons $term_1\ term_2$	$term_1$ `:::` $term_2$	$[term_1]$ `:` $[term_2]$;

Continua na página seguinte

Tabela 6 – continuação da página anterior

Notação teórica	PCF.Term	*Haskell*
		[[$term_1$], *elements of the list(*[$term_2$]*)*]
head $term$	`Hd` [$term$]	`head` [$term$]
tail $term$	`Tl` [$term$]	`tail` [$term$]
isnil $term$	`IsN` [$term$]	`null` [$term$]
letrec $x \leftarrow term_1,\ term_2$	`Letrec "`x`"` [$term_1$] [$term_2$]	`let` *string* `=` [$term_1$] `in` [$term_2$]

Tabela 6: Associação entre várias notações de termos da linguagem PCF: notação teórica, tipo de dados definido, expressão em *Haskell*

Ao definir as conversões de e para *HsExp*, define-se a instância de *SubHsExp*, onde se irão definir as funções `toHsExp` e `fromHsExp` (nas secções 7.2 e 7.2, respectivamente).

```
instance SubHsExp PwPf.PCF.Term where
    -- toHsExp :: Term -> HsExp
    -- fromHsExp :: HsExp -> Maybe Term
```

Conversão para *HsExp* Nesta secção descreve-se a função que converte um termo PCF numa expressão *Haskell* (que utiliza funções da biblioteca Pointless).

```
-- Implementation of a PCF term on a HsExp

toHsExp Star           = mkVar "undefined"
toHsExp (V str)        = mkVar str
toHsExp (t1 :&: t2)    = HsTuple [toHsExp t1,toHsExp t2]
toHsExp (Pi1 t)        = HsApp (mkVar "fst") (mbParen$toHsExp t)
toHsExp (Pi2 t)        = HsApp (mkVar "snd") (mbParen$toHsExp t)
toHsExp (Lam str t)    = HsLambda mkLoc [mkPVar str] (mbParen (toHsExp t))
toHsExp (t1 :-: t2)    = HsApp (mbParen$toHsExp t1) (mbParen$toHsExp t2)
toHsExp T              = mkCon "True"
toHsExp F              = mkCon "False"
toHsExp Z              = HsLit$HsInt 0
toHsExp (Suc t)        = HsApp (mkVar "succ") (mbParen$toHsExp t)
toHsExp (Pred t)       = HsApp (mkVar "pred") (mbParen$toHsExp t)
toHsExp (IsZ t)        = HsApp (HsRightSection (mkOp "==") (HsLit$HsInt 0))
                               (mbParen$toHsExp t)
toHsExp (Ite t1 t2 t3) = HsIf (mbParen$toHsExp t1) (mbParen$toHsExp t2)
                              (mbParen$toHsExp t3)
toHsExp (Y t)          = HsApp (mkVar "fix") (mbParen$toHsExp t)
```

```
toHsExp N = HsList []
toHsExp (t1 ::: t2)     = case (toHsExp t2) of
                            HsList l -> HsList $ (mbParen$toHsExp t1):l
                            x -> HsInfixApp (mbParen$toHsExp t1)
                             (HsQConOp (Special HsCons)) (mbParen x)
toHsExp (Hd t)          = HsApp (mkVar "head") (mbParen$toHsExp t)
toHsExp (Tl t)          = HsApp (mkVar "tail") (mbParen$toHsExp t)
toHsExp (IsN t)         = HsApp (mkVar "null") (mbParen$toHsExp t)
toHsExp (Letrec str t1 t2) =
       HsLet [HsPatBind mkLoc (HsPVar$HsIdent str)
       (HsUnGuardedRhs (toHsExp t1)) []] (toHsExp t2)
```

Conversão de *HsExp* para um termo PCF Nesta secção descreve-se um *parser* muito simples que consegue reconhecer expressões em *Haskell* que tenham tradução directa para o tipo PCF.

```
-- Simple transformation from HsExp to PCF

fromHsExp (HsParen e) = fromHsExp e

-- unit -> "undefined" or "_L"
fromHsExp (HsVar(UnQual(HsIdent "undifined"))) = return Star
fromHsExp (HsVar(UnQual(HsIdent "_L")))        = return Star

-- Bool
fromHsExp (HsCon(UnQual(HsIdent "True")))  = return T
fromHsExp (HsCon(UnQual(HsIdent "False"))) = return F
-- Nat
fromHsExp (HsLit(HsInt 0))                 = return Z
fromHsExp (HsLit(HsInt n))|n>0 =
  do predN <- fromHsExp (HsLit$HsInt$n-1)
     return $ Suc predN
-- List
fromHsExp (HsList [])                      = return N
fromHsExp (HsList (e1:e2)) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp (HsList e2)
      return $ t1 ::: t2
fromHsExp (HsInfixApp e1 (HsQConOp (Special HsCons)) e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 ::: t2

-- Recursion
fromHsExp (HsApp (HsVar (UnQual (HsIdent "fix"))) exp) =
  do term <- fromHsExp exp
     return $ Y term
fromHsExp (HsLet [HsPatBind  _ (HsPVar (HsIdent str))
                             (HsUnGuardedRhs e1) []] e2) =
  do t1 <- fromHsExp e1
     t2 <- fromHsExp e2
     return $ Letrec str t1 t2

fromHsExp (HsApp (HsVar (UnQual (HsIdent "succ"))) e) =
   do t <- fromHsExp e
```

```
      return $ Suc t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "pred"))) e) =
   do t <- fromHsExp e
      return $ Pred t
fromHsExp (HsApp (HsRightSection (HsQVarOp (UnQual (HsSymbol "=="))))
                                                    (HsLit (HsInt 0))) e) =
   do t <- fromHsExp e
      return $ IsZ t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "head"))) e) =
   do t <- fromHsExp e
      return $ Hd t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "tail"))) e) =
   do t <- fromHsExp e
      return $ Tl t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "null"))) e) =
   do t <- fromHsExp e
      return $ IsN t

fromHsExp (HsIf e1 e2 e3) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      t3 <- fromHsExp e3
      return $ Ite t1 t2 t3


fromHsExp (HsVar(UnQual(HsIdent str)))     = return $ V str

fromHsExp (HsTuple [e1,e2]) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :&: t2
fromHsExp (HsApp (HsVar (UnQual (HsIdent "fst"))) e) =
   do t <- fromHsExp e
      return $ Pi1 t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "snd"))) e) =
   do t <- fromHsExp e
      return $ Pi2 t
fromHsExp (HsApp e1 e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :-: t2
fromHsExp (HsLambda _ [HsPVar(HsIdent str)] e) =
   do t <- fromHsExp e
      return $ Lam str t

fromHsExp _ = fail "not a PCF term"
```

7.3 Conversão de PCF para Core

Como referido anteriormente, na linguagem PCF não foi introduzido nenhum conceito que não possa ser expresso por um termo Core. Faz então sentido definir uma função que permita converter uma expressão da linguagem PCF para a linguagem Core (tabela 7).

Torna-se então possível converter uma expressão na linguagem PCF para a linguagem *point-free*, passando pela linguagem Core.

$$
\begin{array}{ll}
\Theta(*) & = * \\
\Theta(x) & = x \\
\Theta(\lambda x.\ t) & = \lambda x.\ t \\
\Theta(t\ u) & = t\ u \\
\Theta(\langle t, u\rangle) & = \langle t, u\rangle \\
\Theta(\mathsf{fst}\ x) & = \mathsf{fst}\ x \\
\Theta(\mathsf{snd}\ x) & = \mathsf{snd}\ x \\
\Theta(\mathsf{T}) & = \mathsf{in}_{Bool}\ (\mathsf{inl}\ *) \\
\Theta(\mathsf{F}) & = \mathsf{in}_{Bool}\ (\mathsf{inr}\ *) \\
\Theta(\mathsf{Z}) & = \mathsf{in}_{Int}\ (\mathsf{inl}\ *) \\
\Theta(\mathsf{succ}\ t) & = \mathsf{in}_{Int}\ (\mathsf{inr}\ \Theta(t)) \\
\Theta(\mathsf{pred}\ t) & = \mathsf{case}\ (\mathsf{out}_{Int}\ \Theta(t))\ \mathsf{of}\ x \to \mathsf{in}_{Int}\ (\mathsf{inl}\ *);\ y \to y \\
\Theta(\mathsf{IsZ}\ t) & = \mathsf{case}\ (\mathsf{out}_{Int}\ \Theta(t))\ \mathsf{of}\ x \to \mathsf{in}_{Bool}\ (\mathsf{inl}\ *);\ y \to \mathsf{in}_{Bool}\ (\mathsf{inr}\ *) \\
\Theta(\mathsf{Ite}\ t\ u\ v) & = \mathsf{case}\ (\mathsf{out}_{Bool}\ \Theta(t))\ \mathsf{of}\ x \to \Theta(u);\ y \to \Theta(v) \\
\Theta(\mathsf{Y}\ x) & = \mathsf{fix}\ x \\
 & \\
\Theta(\mathsf{nil}) & = \mathsf{in}_{[a]}\ (\mathsf{inl}\ *) \\
\Theta(\mathsf{cons}\ t\ u) & = \mathsf{in}_{[a]}\ (\mathsf{inr}\ \langle\Theta(t), \Theta(u)\rangle) \\
\Theta(\mathsf{isnil}\ t) & = \mathsf{case}\ (\mathsf{out}_{[a]}\ \Theta(t))\ \mathsf{of}\ x \to \mathsf{in}_{Bool}\ (\mathsf{inl}\ *);\ y \to \mathsf{in}_{Bool}\ (\mathsf{inr}\ *); \\
\Theta(\mathsf{head}\ t) & = \mathsf{case}\ (\mathsf{out}_{[a]}\ \Theta(t))\ \mathsf{of}\ x \to \mathit{undefined};\ y \to \mathsf{fst}\ y \\
\Theta(\mathsf{tail}\ t) & = \mathsf{case}\ (\mathsf{out}_{[a]}\ \Theta(t))\ \mathsf{of}\ x \to \mathsf{in}_{[a]}\ (\mathsf{inl}\ *);\ y \to \mathsf{snd}\ y \\
\Theta(\mathsf{letrec}\ x \leftarrow t,\ u) & = (\lambda x.\ \Theta(u))\ (\mathsf{fix}\ (\lambda x.\ \Theta(t)))
\end{array}
$$

Tabela 7: Tradução de PCF para Core

É de notar que numa expressão em *Haskell* é possível fazer referências ao tipo de sub-expressões (*e.g.*, a definição `1::Int + 2::Int` está correcta), e que o nome das variáveis presentes no tipo pode ser repetido em sub-expressões com tipos diferentes. Por exemplo, em Core a expressão em haskell que representa o par `([[]],[1]) :: ([[Int]],[Int])` é representada da seguinte forma:

```
(inN (_L::[a]) (Right (([]::[Int]), inN (_L::[a]) (Left _L)  )),
 inN (_L::[a]) (Right ((1::Int), inN (_L::[a]) (Left _L)  )))
```

que é calculada correctamente, apesar de a variável a representar listas de inteiros e inteiros em diferentes partes da mesma expressão.

As listas da linguagem PCF são então representadas em *Haskell* como elementos do tipo "`[a]`", sem que haja problemas na repetição da letra a para definir listas de tipos diferentes.

A função da tabela 7 pode ser definida em *Haskell* da seguinte forma:

```
pcf2core :: PwPf.PCF.Term -> PwPf.Core.Term
pcf2core Star = Unit
pcf2core (V str) = Var str
pcf2core (Lam str t) = Abstr str (pcf2core t)
pcf2core (t1 :-: t2) = (pcf2core t1) :@: (pcf2core t2)
pcf2core (t1 :&: t2) = (pcf2core t1) :><: (pcf2core t2)
pcf2core (Pi1 t)     = Fst (pcf2core t)
pcf2core (Pi2 t)     = Snd (pcf2core t)
pcf2core T           = In "Bool" (Inl Unit)
pcf2core F           = In "Bool" (Inr Unit)
pcf2core Z           = In "Int" (Inl Unit)
pcf2core (Suc t)     = In "Int" (Inr $ pcf2core t)
pcf2core (Pred t)    =
        Case (Out "Int"$pcf2core t) ("_",In "Int" (Inl Unit))
                          ("x", (Var "x"))
pcf2core (IsZ t) =
        Case (Out "Int"$pcf2core t) ("_", In "Bool" (Inl Unit))
                          ("_", In "Bool" (Inr Unit))
pcf2core (Ite t1 t2 t3) =
        Case (Out "Bool"$pcf2core t1) ("_", pcf2core t2)
                           ("_", pcf2core t3)
pcf2core (Y t) = Fix (pcf2core t)

pcf2core N           = In "[a]" (Inl Unit)
pcf2core (t1 ::: t2) = In "[a]" (Inr ((pcf2core t1):><:(pcf2core t2)))
pcf2core (IsN t) =
        Case (Out "[a]"$pcf2core t) ("_", In "Bool" (Inl Unit))
                          ("_", In "Bool" (Inr Unit))
pcf2core (Hd t) =
        Case (Out "[a]"$pcf2core t) ("_", Var "undefined")
                          ("x", Fst $ Var "x")
pcf2core (Tl t) =
        Case (Out "[a]"$pcf2core t) ("_", In "[a]" (Inl Unit))
                          ("x", Snd $ Var "x")
pcf2core (Letrec str t1 t2) =
        let func = (Fix (Abstr str (pcf2core t1)))
        in  (Abstr str (pcf2core t2)) :@: func
```

7.4 Exemplos

Termos PCF

Em seguida encontram-se exemplos de expressões escritas utilizando a notação teórica, o tipo de dados `PCF.Term` e a sua representação em *Haskell* associada.

Em seguida encontram-se exemplos de expressões escritas em *Haskell* e os termos PCF associados.

1. – $\lambda x.$ if (iszero x) then $\langle 0, * \rangle$ else $\langle 1, x \rangle$
   – `Lam "x" (Ite (IsZ (V "x")) (0:&:Star) (1:&:(V "x")))`
   – `\x -> if ((==0) x) then (0,_L) else (1,x)`
2. – letrec $len \leftarrow (\lambda x.$ if (nil x) then zero else succ $(len$ (tail $x)))$, len (cons zero (cons zero nill))
   – `Letrec "len" (Lam "x" (Ite (IsN "x") Z (Suc:@:((V "len"):-:(Tl:-:(V "x"))))))`
   (V "len" `:-:` (0`:::`0`:::`[]))
   – `let len = \x -> if (null x) then 0 else succ (len (tail x))`
   `in len [0,0]`

Recursividade sobre naturais A mesma função do exemplo 6.3 pode ser escrita da seguinte forma:

```
fix (\ f -> \ x ->
             if ((==0) x) then []
                          else 0:(f (pred x))
```

que recebe um natural "n"e devolve uma lista onde o número 0 é replicado "n"vezes.

O código *point-free* gerado é o seguinte:

```
*PwPf.Tester> tPcf2pf "func = fix (\\ f -> \\ x -> if ((==0) x) then [] else 0:(f (pred x)))"
module Main (main) where
transf_func
  = fix .
      (curry
         (curry
            (app .
               ((curry
                   ((((inN (_L :: [a])) . (Left . bang)) \/
                       ((inN (_L :: [a])) .
                          (Right .
                             (((inN (_L :: Int)) . (Left . bang)) /\
                                (app .
                                   (((snd . fst) . fst) /\
                                      (app .
                                         ((curry
                                             ((((inN (_L :: Int)) . (Left . bang)) \/ snd) . distr))
                                           /\ ((ouT (_L :: Int)) . (snd . fst)))))))))
                      . distr))
                 /\
                 ((ouT (_L :: Bool)) .
                    (app .
                       ((curry
                           ((((inN (_L :: Bool)) . (Left . bang)) \/
                               ((inN (_L :: Bool)) . (Right . bang)))
                             . distr))
                          /\ ((ouT (_L :: Int)) . snd)))))))))
```

Aplicando por sua vez a função *point-free* aos argumentos `_L` e `3` obtém-se o seguinte resultado:

```
*PwPf.Tester> (fix . (....)) _L 3
[0,0,0]
```

Que corresponde ao resultado esperado.

A função acima poderia ter sido definida utilizando o *Letrec*, que corresponde em *Haskell* ao seguinte código:

```
let repl = \x -> if ((==0) x) then []
                              else 0:(repl (pred x))
in \x -> repl x
```

que gera código *point-free* diferente mas igualmente correcto.

Recursividade sobre listas De forma idêntica ao exemplo em 6.3, é calculada a dimensão de uma lista. Para isso é convertida a seguinte expressão:

```
fix (\ f -> \ x ->
    if (null x) then 0
                else (succ (f (tail x))))
```

O código gerado é o seguinte:

```
*PwPf.Tester> tPcf2pf "func = fix (\\ f -> \\ x -> if (null x) then 0 else (succ (f (tail x))))"
module Main (main) where
transf_func
= fix . (curry
         (curry
            (app .
               ((curry
                    ((((inN (_L :: Int)) . (Left . bang)) \/
                        ((inN (_L :: Int)) .
                           (Right .
                              (app .
                                 (((snd . fst) . fst) /\
                                    (app .
                                       ((curry
                                           ((((inN (_L :: [a])) . (Left . bang)) \/ (snd . snd)) .
                                              distr))
                                         /\ ((ouT (_L :: [a])) . (snd . fst)))))))))
                       . distr))
                  /\
                  ((ouT (_L :: Bool)) .
                     (app .
                        ((curry
                            ((((inN (_L :: Bool)) . (Left . bang)) \/
                                ((inN (_L :: Bool)) . (Right . bang)))
                               . distr))
                           /\ ((ouT (_L :: [a])) . snd)))))))
```

Que à semelhança do exemplo em 6.3 produz o resultado esperado.

A função também poderia ter sido escrita com letrec, utilizando o seguinte código *Haskell*:

```
let length = \x -> if (null x) then 0
                                else (succ (length (tail x)))
in \x -> length x
```

que gera código *point-free* diferente mas igualmente correcto.

8 BNL

De forma idêntica à linguagem PCF, a linguagem BNL também é outra possível definição que se aproxima mais do estilo de programação funcional a que estamos habituados.

A palavra BNL vem de "booleanos, naturais e listas". Um termo BNL é então uma extensão de um termo Core onde constantes destes tipos são introduzidos, assim como paramorfismos.

8.1 Definição de um termo

Trabalho anterior No início deste projecto a definição da linguagem Core era diferente. Era possível definir uma extensão num ficheiro à parte que permitisse reaproveitar todas as suas definições, e as conversões com expressões em *Haskell*. Foi neste contexto que a linguagem BNL foi introduzida, como extensão de termos Core.

No entanto ainda não era possível definir recursividade, e as constantes na linguagem Core eram representadas por expressões em *Haskell* (*HsExp*). Desta forma, ao converter uma constante em BNL para Core, era simplesmente convertido para a sua notação em *Haskell* e aplicado o construtor `Const`.

Para cada um dos tipos, naturais e listas, existia um case of expecífico, que era transformado num case genérico. Por exemplo, no caso dos naturais, o termo BNL:

$$\mathsf{natCase}\ term1\ term2\ \langle var, term3\rangle$$

era convertido num termo equivalente ao seguinte na linguagem Core:

$$\mathsf{case}\ (\mathsf{out}_{Int}\ term1)\ \mathsf{of}\ x \to term2; var \to term3$$

A linguagem BNL

Os tipos da linguagem BNL são os mesmos da linguagem PCF extendida:

$$A, B ::= bool \mid nat \mid list \mid A \to B \mid A \times B \mid 1$$

Um termo BNL é definido da seguinte forma:

$$\begin{aligned} M, N, P ::= {} & * \mid x \mid M\ N \mid \lambda x.\ M \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \\ & \mid \mathsf{T} \mid \mathsf{F} \mid \mathsf{zero} \mid \mathsf{succ}\ M \mid \mathsf{nil} \mid \mathsf{cons}\ M\ N \\ & \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \mid \mathsf{recNat}\ M\ N\ P \mid \mathsf{recList}\ M\ N\ P \end{aligned}$$

Como se pode verificar, a linguagem BNL é bastante parecida com a linguagem PCF extendida, a menos da forma de representar recursividade e do facto de ter menos destructores (mas que podem ser definidos com as funções recNat e recList).

Recursividade – paramorfismos vs. pontos fixos Na definição de termos BNL, ao contrário de termos Core e PCF, não são utilizados pontos fixos para definir padrões recursivos. No caso da linguagem PCF é ainda utilizado o letrec, que consiste apenas em algum açúcar sintático para a função ponto-fixo. Em vez disso são utilizados padrões de recursividade primitiva (recNat e recList), que em *point-free* são também conhecidos como paramorfismos (definidos na secção 2.3).

As funções recNat e recList podem definir-se em *Haskell* da seguinte forma:

```
recNat :: Int -> (Int -> a -> a) -> a -> a
recNat 0 f z = z
recNat (n+1) f z = f n (recNat n f z)

recList :: [a] -> (a -> [a] -> b -> b) -> b -> b
recList [] f z = z
recList (h:t) f z = f h t (recList t f z)
```

O principal objectivo é criar um termo *point-free* equivalente a um termo BNL, utilizando os termos Core como passo intermédio. É desejável então que um recNat ou um recList seja reconhecido como um paramorfismo em *point-free*. O construtor que representa paramorfismos já se encontra definido na linguagem *point-free* (secção 4).

Ao converter um recNat ou um recList num termo Core, estes são passados para pontos fixos. No entanto, estes pontos fixos obedecem a um determinado padrão que é reconhecido como um paramorfismo na conversão de termos Core para *point-free*, conforme a secção 6.2.

Os paramorfismos apenas são reconhecidos nos casos em que as funções que determinam o passo recursivo e o de paragem (no recNat e no recList) não possuem variáveis livres, *i.e.*, não são feitas referências a variáveis declaradas fora do respectivo rec.

Na tabela 8 é possível comparar o resultado da conversão de uma mesma função para *point-free*, onde num dos casos é convertido para um paramorfismo e no outro é utilizado o ponto-fixo.

A função testada na tabela 8 é a seguinte:

$$\lambda x.\ \mathsf{recNat}\ x\ (\lambda n.\ \lambda rec.\ \mathsf{succ}\ (\mathsf{succ}\ rec))\ 0$$

que calcula o dobro de um natural.

Recorrendo a um paramorfismo	**Sem recorrer a um paramorfismo**

`curry`
`((para (_L :: Int)`
`(app .`
`(((curry (app . ((curry`
`((((inN (_L :: Int)) . (Left . bang))`
`\/ (app . ((app . ((curry (curry`
`((inN (_L :: Int)) .`
`(Right . ((inN (_L :: Int))`
`. (Right . snd))))))`
`/\ (snd . snd)))`
`/\ (fst . snd))))`
`. distr)) /\ snd))) . bang)`
`/\ id)))`
`. snd)`
	`curry`
	`(app . ((app . ((app .`
	`((fix .`
	`(curry (curry (curry`
	`(curry (app . ((curry (((snd . fst) \/`
	`(app . ((app . (((snd . fst) . fst)`
	`/\ snd)) /\`
	`(app . ((app . ((app .`
	`(((((snd . fst) . fst) . fst) . fst)`
	`/\ snd))`
	`/\ ((snd . fst) . fst)))`
	`(snd . fst))))))`
	`. distr))`
	`/\ ((ouT (_L :: Int)) . ((snd . fst)`
	`. fst)))))))))`
	`/\ snd))`
	`/\`
	`(curry`
	`(curry`
	`((inN (_L :: Int)) .`
	`(Right . ((inN (_L :: Int))`
	`. (Right . snd))))))))`
	`/\ ((inN (_L :: Int)) . (Left . bang))))`

Tabela 8. Comparação de um termo em *point-free* com e sem recurso a paramorfismos

Definição em *Haskell* Um termo BNL é definido em *Haskell* da seguinte forma:

```
-- *Definition of the data type

data Term = Star                              -- ^Unit
          | V String                          -- ^Variable
          | Lam String PwPf.BNL.Term          -- ^Lambda abstraction
          | PwPf.BNL.Term :-: PwPf.BNL.Term   -- ^Aplication
          | PwPf.BNL.Term :&: PwPf.BNL.Term   -- ^Pair
          | Pi1 PwPf.BNL.Term                 -- ^Projection of the first element
          | Pi2 PwPf.BNL.Term                 -- ^Projection of the second element
          | T                   -- ^Constant True
          | F                   -- ^Constant False
          | Z                   -- ^Constant Zero
```

```
  | N                                  -- ^Constant Nil (empty list)
  | Suc PwPf.BNL.Term                  -- ^Successor
  | PwPf.BNL.Term ::: PwPf.BNL.Term    -- ^List constructor
  | Ite PwPf.BNL.Term PwPf.BNL.Term PwPf.BNL.Term
                                       -- ^if then else
  | RecNat  PwPf.BNL.Term PwPf.BNL.Term PwPf.BNL.Term
                                       -- ^Primitive recursion on Nat's
  | RecList PwPf.BNL.Term PwPf.BNL.Term PwPf.BNL.Term
                                       -- ^Primitive recursion on List's
    deriving Show
```

8.2 Conversões entre um termo BNL e um *HsExp*

A conversão de termos da linguagem BNL acima definida para programas *Haskell* é imediata, envolvendo pouca ou nenhuma manipulação da expressão em causa. No sentido inverso a função de conversão é igualmente simples, mas é naturalmente parcial. Em termos práticos, estas funções de conversão têm o tipo $Term \rightarrow HsExp$ e $HsExp \rightarrow Term$ respectivamente, sendo *HsExp* o tipo utilizado para a representação de expressões da linguagem *Haskell* (como referido na secção 4.2).

A relação entre termos Core escritos utilizando diferentes notações encontra-se resumida na tabela 9.

Notação Teórica	**BNL.Term**	***Haskell***
$*$	`Star`	`undefined`; `_L`
x	`V "`x`"`	x
$\lambda x.\ term$	`Lam "`x`"` $[term]$	$\backslash x$ `->` $[term]$
$term_1\ term_2$	$[term_1]$ `:-:` $[term_2]$	$[term_1]$ $[term_2]$
$\langle term_1, term_2 \rangle$	$[term_1]$ `:&:` $[term_2]$	`(`$[term_1]$`,`$[term_2]$`)`
fst $term$	`Pi1` $[term]$	`fst` $[term]$
snd $term$	`Pi2` $[term]$	`snd` $[term]$
T	`T`	`True`
F	`F`	`False`
zero	`Z`	`0`
nil	`N`	`[]`
succ $term$	`Suc` $[term]$	`succ` $[term]$; $natural\_number$

Continua na página seguinte

Tabela 9 – continuação da página anterior

BNL	Read *Haskell*	Written *Haskell*
cons $term_1$ $term_2$	$term_1$ `:::` $term_2$	$[term_1]$ `:` $[term_2]$; $[[term_1]$, *elements of the list(*$[term_2]$*)*$]$
if $term_1$ then tem_2 else $term_3$	`Ite` $[term_1]$ $[term_2]$ $[term_3]$	`if` $[term_1]$ `then` $[term_2]$ `else` $[term_3]$
recNat $term_1$ $term_2$ $term_3$	`RecNat` $[term_1]$ $[term_2]$ $[term_3]$	`recNat` $[term_1]$ $[term_2]$ $[term_3]$
recNat $term_1$ $term_2$ $term_3$	`RecNat` $[term_1]$ $[term_2]$ $[term_3]$	`recNat` $[term_1]$ $[term_2]$ $[term_3]$

Tabela 9: Associação entre várias notações de termos da linguagem BNL: notação teórica, tipo de dados definido, expressão em *Haskell*

Ao definir as conversões de e para *HsExp*, define-se a instância de *SubHsExp*, onde se irão definir as funções `toHsExp` e `fromHsExp` (nas secções 8.2 e 8.2, respectivamente).

```
instance SubHsExp PwPf.BNL.Term where
    -- toHsExp :: Term -> HsExp
    -- fromHsExp :: HsExp -> Maybe Term
```

Conversão para *HsExp* Nesta secção descreve-se a função que converte um termo BNL numa expressão *Haskell* (que utiliza funções da biblioteca Pointless).

```
-- Implementation of a BNL term on a HsExp

toHsExp Star           = mkVar "undefined"
toHsExp (V str)        = mkVar str
toHsExp (t1 :&: t2)    = HsTuple [toHsExp t1,toHsExp t2]
toHsExp (Pi1 t)        = HsApp (mkVar "fst") (mbParen$toHsExp t)
toHsExp (Pi2 t)        = HsApp (mkVar "snd") (mbParen$toHsExp t)
toHsExp (Lam str t)    = HsLambda mkLoc [mkPVar str] (mbParen (toHsExp t))
toHsExp (t1 :-: t2)    = HsApp (mbParen$toHsExp t1) (mbParen$toHsExp t2)
toHsExp T              = mkCon "True"
toHsExp F              = mkCon "False"
toHsExp Z              = HsLit$HsInt 0
toHsExp N              = HsList []
toHsExp (Suc t)        = HsApp (mkVar "succ") (mbParen$toHsExp t)
toHsExp (t1 ::: t2)    = case (toHsExp t2) of
                           HsList l -> HsList $ (mbParen$toHsExp t1):l
                           x -> HsInfixApp (mbParen$toHsExp t1)
```

```
                                (HsQConOp (Special HsCons)) (mbParen x)
toHsExp (Ite t1 t2 t3) = HsIf (mbParen$toHsExp t1) (mbParen$toHsExp t2)
                               (mbParen$toHsExp t3)
toHsExp (RecNat t1 t2 t3) = HsApp (HsApp (HsApp (mkVar "recNat")
                                (mbParen$toHsExp t1)) (mbParen$toHsExp t2))
                                 (mbParen$toHsExp t3)
toHsExp (RecList t1 t2 t3) = HsApp (HsApp (HsApp (mkVar "recList")
                                (mbParen$toHsExp t1)) (mbParen$toHsExp t2))
                                 (mbParen$toHsExp t3)
```

Conversão de *HsExp* para um termo BNL Nesta secção descreve-se um *parser* muito simples que consegue reconhecer expressões em *Haskell* que tenham tradução directa para o tipo *BNL*.

```
-- Simple transformation from HsExp to BNL

fromHsExp (HsParen e) = fromHsExp e

-- unit -> "()" or "undefined" or "_L"
fromHsExp (HsCon(Special HsUnitCon))            = return Star
fromHsExp (HsVar(UnQual(HsIdent "undifined"))) = return Star
fromHsExp (HsVar(UnQual(HsIdent "_L")))         = return Star

-- Bool
fromHsExp (HsCon(UnQual(HsIdent "True")))  = return T
fromHsExp (HsCon(UnQual(HsIdent "False"))) = return F
-- Nat
fromHsExp (HsLit(HsInt 0))                 = return Z
fromHsExp (HsLit(HsInt n))|n>0 =
  do predN <- fromHsExp (HsLit$HsInt$n-1)
     return $ Suc predN
-- List
fromHsExp (HsList [])                      = return N
fromHsExp (HsList (e1:e2)) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp (HsList e2)
      return $ t1 ::: t2

-- Recursion
fromHsExp (HsApp (HsApp (HsApp (HsVar (UnQual (HsIdent "recNat")))
                                                exp1) exp2) exp3) =
  do term1 <- fromHsExp exp1
     term2 <- fromHsExp exp2
     term3 <- fromHsExp exp3
     return $ RecNat term1 term2 term3
fromHsExp (HsApp (HsApp (HsApp (HsVar (UnQual (HsIdent "recList")))
                                                exp1) exp2) exp3) =
  do term1 <- fromHsExp exp1
     term2 <- fromHsExp exp2
     term3 <- fromHsExp exp3
     return $ RecList term1 term2 term3


fromHsExp (HsApp (HsVar (UnQual (HsIdent "succ"))) e) =
   do t <- fromHsExp e
```

```
      return $ Suc t
fromHsExp (HsInfixApp e1 (HsQConOp (Special HsCons)) e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 ::: t2
fromHsExp (HsIf e1 e2 e3) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      t3 <- fromHsExp e3
      return $ Ite t1 t2 t3

fromHsExp (HsVar(UnQual(HsIdent str)))     = return $ V str

fromHsExp (HsTuple [e1,e2]) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :&: t2
fromHsExp (HsApp (HsVar (UnQual (HsIdent "fst"))) e) =
   do t <- fromHsExp e
      return $ Pi1 t
fromHsExp (HsApp (HsVar (UnQual (HsIdent "snd"))) e) =
   do t <- fromHsExp e
      return $ Pi2 t
fromHsExp (HsApp e1 e2) =
   do t1 <- fromHsExp e1
      t2 <- fromHsExp e2
      return $ t1 :-: t2
fromHsExp (HsLambda _ [HsPVar(HsIdent str)] e) =
   do t <- fromHsExp e
      return $ Lam str t

fromHsExp _ = fail "not a BNL term"
```

8.3 Conversão de BNL para Core

De forma análoga ao que foi feito com a linguagem PCF, é possível definir uma função que converte uma expressão BNL numa expressão Core (para posteriormente se poder converter numa expressão *point-free*).

$$
\begin{array}{ll}
\Omega(*) & = * \\
\Omega(x) & = x \\
\Omega(\lambda x.\ t) & = \lambda x.\ t \\
\Omega(t\ u) & = t\ u \\
\Omega(\langle t, u\rangle) & = \langle t, u\rangle \\
\Omega(\mathsf{fst}\ x) & = \mathsf{fst}\ x \\
\Omega(\mathsf{snd}\ x) & = \mathsf{snd}\ x \\
\Omega(\mathsf{T}) & = \mathsf{in}_{Bool}\ (\mathsf{inl}\ *) \\
\Omega(\mathsf{F}) & = \mathsf{in}_{Bool}\ (\mathsf{inr}\ *)
\end{array}
$$

$$
\begin{array}{ll}
\Omega(\mathsf{zero}) & = \mathsf{in}_{Int}\ (\mathsf{inl}\ *) \\
\Omega(\mathsf{nil}) & = \mathsf{in}_{[a]}\ (\mathsf{inl}\ *) \\
\Omega(\mathsf{succ}\ t) & = \mathsf{in}_{Int}\ (\mathsf{inr}\ \Omega(t)) \\
\Omega(\mathsf{cons}\ t\ u) & = \mathsf{in}_{[a]}\ (\mathsf{inr}\ \langle \Omega(t), \Omega(u) \rangle) \\
\Omega(\mathsf{Ite}\ tuv) & = \mathsf{case}\ (\mathsf{out}_{Bool}\ \Omega(t))\ \mathsf{of}\ x \to \Omega(u);\ y \to \Omega(v) \\
\Omega(\mathsf{Y}\ x) & = \mathsf{fix}\ x
\end{array}
$$

$$
\begin{array}{l}
\Omega(\mathsf{recNat}\ t\ u\ v) = (\mathsf{fix}\ (\lambda r.\ \lambda n.\ \lambda f.\ \lambda z. \\
\qquad\qquad \mathsf{case}\ (\mathsf{out}_{Int}\ n)\ \mathsf{of} \\
\qquad\qquad\quad x \to z; \\
\qquad\qquad\quad y \to f\ y\ (r\ y\ f\ z))) \\
\qquad\qquad \Omega(t)\ \Omega(u)\ \Omega(v) \\
\Omega(\mathsf{recList}\ t\ u\ v) = (\mathsf{fix}\ (\lambda r.\ \lambda l.\ \lambda f.\ \lambda z. \\
\qquad\qquad \mathsf{case}\ (\mathsf{out}_{Int}\ l)\ \mathsf{of} \\
\qquad\qquad\quad x \to z; \\
\qquad\qquad\quad y \to f\ (\mathsf{fst}\ y)\ (\mathsf{snd}\ y)\ (r\ (\mathsf{snd}\ y)\ f\ z))) \\
\qquad\qquad \Omega(t)\ \Omega(u)\ \Omega(v)
\end{array}
$$

Tabela 10: Tradução de BNL para Core

De forma idêntica às listas da linguagem PCF, as listas da linguagem BNL são então representadas em *Haskell* como elementos do tipo "`[a]`", sem que haja problemas na repetição da letra a para definir listas de tipos diferentes.

Em *Haskell*, a conversão de termos da linguagem BNL para a linguagem Core pode ser definido da seguinte forma:

```
bnl2core :: PwPf.BNL.Term -> PwPf.Core.Term
bnl2core Star = Unit
bnl2core (V str) = Var str
bnl2core (Lam str t) = Abstr str (bnl2core t)
bnl2core (t1 :-: t2) = (bnl2core t1) :@: (bnl2core t2)
bnl2core (t1 :&: t2) = (bnl2core t1) :><: (bnl2core t2)
bnl2core (Pi1 t)     = Fst (bnl2core t)
bnl2core (Pi2 t)     = Snd (bnl2core t)
bnl2core T           = In "Bool" (Inl Unit)
bnl2core F           = In "Bool" (Inr Unit)
bnl2core Z           = In "Int" (Inl Unit)
bnl2core N           = In "[a]" (Inl Unit)
bnl2core (Suc t)     = In "Int" (Inr $ bnl2core t)
bnl2core (t1 ::: t2) = In "[a]" (Inr ((bnl2core t1):><:(bnl2core t2)))
bnl2core (Ite t1 t2 t3) =
```

```
        Case (Out "Bool"$bnl2core t1) ("_", bnl2core t2)
                              ("_", bnl2core t3)
bnl2core (RecNat t1 t2 t3) =
        (Fix $ Abstr "r" $ Abstr "n" $ Abstr "f" $ Abstr "z" $
            Case (Out "Int" (Var "n"))
                 ("x", Var "z")
                 ("y", (Var "f") :@: (Var "y") :@:
                     ((Var "r"):@:(Var "y"):@:(Var "f"):@:(Var "z"))))
        :@: (bnl2core t1) :@: (bnl2core t2) :@: (bnl2core t3)
{-   (fix (
       \ r n f z ->
        case (out n) of Left x -> z
                        Right y -> f y (r y f z)
     )
    ) t1 t2 t3
-}
bnl2core (RecList t1 t2 t3) =
        (Fix $ Abstr "r" $ Abstr "l" $ Abstr "f" $ Abstr "z" $
            Case (Out "[a]" (Var "l"))
                 ("x", Var "z")
                 ("y", (Var "f") :@: (Fst$Var "y") :@: (Snd$Var "y") :@:
                   ((Var "r"):@:(Snd$Var "y"):@:(Var "f"):@:(Var "z"))))
        :@: (bnl2core t1) :@: (bnl2core t2) :@: (bnl2core t3)
```

8.4 Exemplos

Termos BNL Em seguida encontram-se exemplos de expressões escritas utilizando a notação teórica, o tipo de dados `Pointfree.BNL` e a sua representação em *Haskell* associada.

1. – $\lambda x.\ \mathsf{recNat}\ x\ (\lambda y.\ \lambda z.\ \mathsf{F})\ \mathsf{T}$
   – `Lam "x"(RecNat (V "x") (Lam "y"(Lam "z" F)) T)`
   – `\x -> recNat x (\y -> \z -> False) True`
2. – $\mathsf{fst}\ \langle \mathsf{succ}\ \mathsf{zero}, \mathsf{cons}\ \mathsf{T}(\mathsf{cons}\ \mathsf{F}\ \mathsf{N})\rangle$
   – `Pi1 (Suc Z :&: (T:::F:::N))`
   – `fst (1,[True,False])`

Recursividade sobre naturais A mesma função do exemplo 6.3 pode ser escrita da seguinte forma:

```
\ x -> recNat x (\ y -> \ rec -> 0:rec) []
```

que recebe um natural "n"e devolve uma lista onde o número 0 é replicado "n"vezes.

O código *point-free* gerado é o seguinte:

```
*PwPf.Tester> tBnl2pf "func = \\ x -> recNat x (\\ y -> \\ rec -> 0:rec ) []"
module Main (main) where
transf_func
  = curry
      ((para (_L :: Int)
          (app .
             (((curry
                  (app .
                     ((curry
                         ((((inN (_L :: [a])) . (Left . bang)) \/
                             (app .
                                ((app .
                                    ((curry
                                        (curry
                                           ((inN (_L :: [a])) .
                                              (Right .
                                                 (((inN (_L :: Int)) . (Left . bang)) /\ snd)))))
                                       /\ (snd . snd)))
                                   /\ (fst . snd))))
                            . distr))
                        /\ snd)))
                . bang)
               /\ id)))
         . snd)
```

Que à semelhança do exemplo em 6.3 produz o resultado esperado.

É de notar que no código *point-free* gerado foi produzido um paramorfismo e não um ponto fixo (embora tal também estivesse correcto). Na seccção 8.1 esta ideia é explorada em mais profundidade.

Recursividade sobre listas De forma idêntica ao exemplo em 6.3, é calculada a dimensão de uma lista. Para isso é convertida a seguinte expressão:

```
\ x -> recList x (\ hd -> \ tl -> \ rec -> succ rec) 0
```

O código gerado é o seguinte:

```
*PwPf.Tester> tBnl2pf "func = \\ x -> recList x (\\ hd -> \\ tl -> \\ rec -> succ rec) 0"
module Main (main) where
transf_func
  = curry
      ((para (_L :: [a])
          (app .
             (((curry
                  (app .
                     ((curry
                         ((((inN (_L :: Int)) . (Left . bang)) \/
                             (app .
                                ((app .
                                    ((app .
                                        ((curry (curry (curry ((inN (_L :: Int)) . (Right . snd)))))
                                           /\ (fst . snd)))
```

```
                                    /\ (snd . (snd . snd))))
                                /\ (fst . (snd . snd)))))
                        . distr))
                    /\ snd)))
            . bang)
           /\ id)))
    . snd)
```

que à semelhança do exemplo em 6.3 produz o resultado esperado.

Neste exemplo também é gerado um paramorfismo e não um ponto fixo, que é explicado em mais detalhe na secção 8.1.

9 Conclusão

Neste trabalho foram implementadas formas de converter um sub-conjunto da linguagem *Haskell* em *pointwise* com recursividade explícita para expressões em *point-free* com recursividade explícita e, para alguns casos particulares, para *point-free* com recursividade implícita, que seria o principal objectivo. Também foi exemplificada uma forma de extender este sub-conjunto através da definição das linguagens PCF e BNL.

A recursividade explícita descrita neste trabalho apenas reconhece pontos fixos e alguns `let`'s (sem argumentos). Existe no entanto bastante açúcar sintáctico permitido na linguagem *Haskell* que não é contemplado neste trabalho.

É possível também verificar que neste trabalho não são implementados mecanismos de inferências de tipos que permitam saber o tipo dos termos em *point-free* obtidos. Apesar de ser sempre possível inferir um tipo genérico para cada termo, o conhecimento do tipo concreto poderia facilitar muito a transformação e simplificação destes termos que, como se pode verificar, são geralmente grandes e difíceis de ler. Esta tarefa poderia ser simplificada caso os actuais compiladores/interpretadores de *Haskell* incluíssem ferramentas que permitissem a inferência de tipos de expressões.

Como trabalho futuro destacam-se as seguintes tarefas:

- Desenvolvimento de formas de extrair açúcar sintáctico para aumentar o sub-conjunto da linguagem *Haskell* abrangida por este trabalho;
- Desenvolvimento de ferramentas que manipulem expressões *point-free* com vista a simplificar termos resultantes das traduções e em reconhecer padrões conhecidos.

A segunda sugestão, como dito anteriormente, seria mais simples e produziria provavelmente termos mais simplificados caso os termos em *point-free* produzidos fossem tipados.

Referências

Bac78. John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

Cun04. Alcino Cunha. *Métodos Algébricos para o Cálculo e Transformação de Programas*. PhD thesis, University of Minho, 2004. Not yet published.

Cur93. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhäuser, 2nd edition, 1993.

dB72. Nicolaas de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

FM91. Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

Mee92. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

Plo77. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

Sco93. Dana S. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theor. Comput. Sci.*, 121(1-2):411–440, 1993.

A Common

Um conjunto de funções e instâncias que faziam sentido serem importados por todas as bibliotecas desenvolvidas foram colocados num único ficheiro, que se apresenta em seguida:

```
module PwPf.Common where

import Language.Haskell.Syntax

-- Debug functions
import Debug.Trace
ffail = error
traceT s = trace ("##<## "++s++" ##>##")
trace2 s = traceT (show s) s


-- | places parentisis in an expression only if it is necessary
mbParen :: HsExp -> HsExp
mbParen e@(HsApp _ _) = HsParen e
mbParen e@(HsInfixApp _ _ _) = HsParen e
mbParen e@(HsCase _ _) = HsParen e
mbParen e@(HsLambda _ _ _) = HsParen e
mbParen x = x

-- Auxiliary functions
```

```
-- | creates a null location
mkLoc = SrcLoc "" 0 0
-- | creates a null name
mkName = HsIdent ""
mkCon  = HsCon . UnQual . HsIdent
mkOp   = HsQVarOp . UnQual . HsSymbol
mkVar  = HsVar . UnQual . HsIdent
mkPVar = HsPVar . HsIdent



{-
 | SubHsExp represents a data type that can be written has an haskell
expression. It requires the definition of the functions to convert
form an to an Haskell expression, according to the library
Language.Haskell.Syntax.

-}
class SubHsExp a where
    toHsExp :: a -> HsExp
    fromHsExp :: HsExp -> Maybe a
```