Conversão de Código *Pointwise* para Código *Point-free*

José Miguel Vilaça
jmvilaca@di.uminho.pt

Techn. Report DI-PURe-04.11.02
2004, Novembro

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-04.11.02
Conversão de Código Pointwise
para Código Point-free by José Miguel Vilaça

Abstract

Actualmente a certificação da qualidade do software é uma preocupação crescente da comunidade empresarial e instituicional. Esta necessidade induziu a comunidade científica a desenvolver enquadramentos teóricos para suportar tal certificação.

Situando-se apenas no paradigma de programação funcional este projecto pretende ser a "ponte" entre dois estilos de programação do referido paradigma: o *pointwise* e o *point-free*. O *pointwise* é o estilo mais usual e mais intuitivo mas menos adequado ao cálculo, à transformação e à verificação de propriedades. Em contraposição tem-se o *point-free* que se apresenta como mais favorável às operações de cálculo, transformação e verificação, contudo perdendo na comparação com as mais valias do *point-free*.

Mais concretamente, este projecto apresenta as regras de conversão de código Haskell *pointwise* para código Haskell *point-free*, assim como as regras de criação de padrões recursivos com base na definição do tipo de dados indutivo. Por forma a efectivar estas ideias, são também apresentadas bibliotecas e uma ferramenta que automatizam tais tarefas.

1 Introdução

Desde há longos anos que a comunidade científica procura modos de lidar, tratar e pensar os programas informáticos. Esta procura converge com as pretensões das empresas e instituições de terem garantias da qualidade desses programas e de os poderem melhorar, tornando-os mais eficientes em tempo e/ou espaço sem perda das garantias de qualidade já adquiridas.

Um paradigma em que tal esforço tem sido efectivo é o funcional. Este paradigma defende o princípio de programas estruturados como a chave para *software* de qualidade e capaz de garantir essa mesma qualidade; isto é, não se quer *software* cuja qualidade surge do acaso mas *software* de qualidade certificada.

Actualmente pensa-se que o cálculo e a transformação de programas funcionais poderão ser mais simples sobre programas *point-free*. Antes de mais convém salientar o que se entende por programas *point-free*; numa definição muito elementar, são programas que não contêm variáveis em oposição, obviamente, com os programas com variáveis, ditos *pointwise*.

Mas afinal o que significa "o cálculo e a transformação de programas são mais simples sobre programas *point-free*" ? Raciocinar, calcular e manipular, quer seja para transformar o programa ou para provar propriedades sobre o mesmo, são acções, em grande medida, simplificadas quando executadas sobre os programas *point-free*. Sobre isto vejam-se, por exemplo, os trabalhos *An Introduction to Pointfree Programming* [Oli99a] e *Point-free Program Transformation* [CP04].

Com esta aproximação *point-free*, ganha-se (gratuitamente) conhecimentos matemáticos tais como uma vasta quantidade de leis algébricas e equacionais e ainda métodos de cálculo já sobejamente estudados e cujo rigor está provado.

Isto, sem dúvida, representa um atalho (cujo rigor cientifico está garantido) na teoria a desenvolver e um enorme ganho no cálculo e na transformação de programas.

Mas (e há sempre um mas...) sabe-se que apenas um reduzido número de programadores escreve os seus programas no estilo *point-free*. Assim sendo, as técnicas e tudo o resto que está associado ao cálculo e à transformação de programas através do *point-free* estão, à partida, restritos a estes programadores.

Para ultrapassar esta restrição, torna-se pois necessário converter os programas *pointwise* nos seus equivalentes em *point-free*, podendo então ser aplicado o processo de cálculo e/ou transformação *point-free*.

Fazer este processo manualmente acarreta um esforço extra, comparável a escrever novamente os programas (agora em *point-free*), e aumenta a probabilidade de introdução de novos erros.

Surge assim a necessidade de desenvolver uma ferramenta capaz de, automaticamente, proceder à conversão de *pointwise* em *point-free*, por forma a que o facto de os programas serem escritos no estilo *pointwise* não penalize o seu tratamento com as técnicas *point-free*.

O objectivo deste projecto é então a criação desta ferramenta de conversão para os programas **Haskell**; esta ferramenta será aqui designada de "*pointfrezador*".

Institucionalmente este projecto enquadra-se no projecto de investigação **Program Understanding and Re-engineering: Calculi and Applications** (POSI/ICHS/44304/2002), abreviado para **PURe**, financiado pela Fundação para Ciência e Tecnologia e que decorre no Departamento de Informática da Universidade do Minho.

O projecto, alvo deste relatório, é contextualizado no PURe e especialmente nos trabalhos de Alcino Cunha e Jorge Sousa Pinto.

Destacam-se, desde já, os trabalhos *Point-free Program Transformation* [CP04] (investigação conjunta dos autores referidos anteriormente)e *Point-free Programming with Hylomorphisms* [Cun] da autoria de Alcino Cunha.

É então com ânimo que se inicia o projecto, ciente contudo da impossibilidade de criar a ferramenta total e perfeita.

Estrutura do Relatório

No capítulo 2 far-se-á uma introdução ao *point-free*; o que é, quais as suas vantagens e limitações. Nas secções far-se-á a apresentação e definição dos combinadores (2.1), uma introdução de uma sua implementação em **Haskell** - a biblioteca Pointless - (2.2), uma apresentação sucinta dos tipos indutivos e das suas propriedades (2.3) e uma curta explanação dos padrões recursivos que estão associados aos tipos indutivos (2.4), nomeadamente uma referência à implementação deste padrões na biblioteca Pointless (2.4).

No capítulo 3 aborda-se como se representam as expressões *point-free* neste trabalho, apresentando o tipo de dados que suporta a representação (3.1), as limitações da representação escolhida (3.2) e ainda o modo como se visualizam as expressões *point-free* (3.3).

O capítulo 4 concerne a representação do código **Haskell**, neste trabalho, afim de poder ser manipulado. As secções desta referem-se à representação das construções da linguagem **Haskell** que são relevantes no âmbito deste trabalho.

O capítulo 5 aborda a teoria subjacente à conversão de *point-free* para *pointwise* (o propósito deste trabalho). As três primeiras secções explicam as diferentes fases da conversão, enquanto a última (5.4) explica a geração automática das funções *in* e *out* de tipos de dados indutivos.

O capítulo 6 explica, a um nível bastante abstracto, como implementar as ideias do capítulo anterior e tem uma secção com alguns exemplos de utilização.

No capítulo 7 referem-se algumas ideias que não foram implementadas e situações para as quais não se conhece a conversão, embora algumas indicações se tenham encontrado.

O capítulo 8 apresenta as conclusões e o trabalho futuro.

Em apêndice surgem um manual de utilização da ferramenta de conversão e o código de implementação da representação de expressões *point-free*.

2 Point-free e sua Motivação

A introdução do estilo de programação *point-free* deve-se a John Backus, em 1977, na sua dissertação ACM Turing Award com o intuito de desenvolver um cálculo de programas que pudesse ser utilizado para a sua transformação.

Desde logo o estilo *point-free* se associou ao paradigma de programação funcional, não só pelo seu enorme poder de programação estrutural, mas sobretudo por todo o poder algébrico e equacional que já então lhe estava subjacente.

Este estilo caracteriza-se por programas que são expressos como combinações de funções mais simples. Dois factos se destacam de imediato; o primeiro é que os argumentos das funções não são referidos (é desta inexistência de variáveis que advém a designação *point-free*) e o segundo é que os combinadores funcionam como "ligadores" entre funções mais simples ou expressões construídas com os mesmos combinadores.

O objectivo inicial era o de criar um conjunto reduzido de combinadores que derivassem de formas categoriais *standard* estando, desde logo, estes combinadores equipados com um vasto conjunto de leis equacionais.

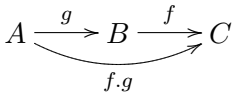
2.1 Combinadores

O primeiro combinador que surge, pois é já vulgarmente conhecido fora do contexto *point-free*, é a composição. Pense-se numa função f que é aplicada ao resultado de aplicar uma

outra função, seja g, a um argumento x. Simbolicamente escreve-se $f\ (g(x))$. Suponha-se que se quer uma função h tal que $f\ (g(x)) = h(x)$ para todo o x no domínio de g e tal que o contra-domínio de g esteja contido no domínio de f. Intuitivamente pensar-se-á que h é definida à custa de f e g. Define-se então o combinador composição (e representa-se pelo sinal .) que dadas duas funções $g\ :\ A \to B$ e $f\ :\ B \to C$ devolve uma nova função (h pretendida) $f\ .\ g\ :\ A \to C$ cuja definição *pointwise* é

$$(f\ .\ g)\ x\ =\ f(g(x))$$

Diagramaticamente



Usam-se aqui as definições *pointwise* para ambientar o leitor desconhecedor do estilo *point-free* com os combinadores usando algo que lhe é previamente conhecido. Além disso, existe um conjunto mínimo de combinadores que tem de ser expresso em *pointwise*.

Suponha-se agora o caso de duas funções que partilham o mesmo domínio, isto é, funções f e g tais que $f\ :\ A \to B$ e $g\ :\ A \to C$.

Como as combinar? Uma possibilidade é uma função $h\ :\ A \to B \times C$.

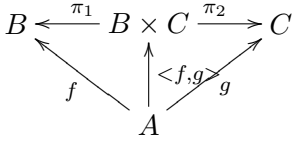
O combinador que permite expressar a função h à custa das funções f e g é denominado *split*. Existem duas representações usuais para o *split* das funções f e g

- $f\ /\backslash\ g$
- $<\ f,\ g\ >$

A definição *pointwise* é

$$<\ f,\ g\ >\ x\ =\ (f\ x,\ g\ x)$$

O respectivo diagrama de tipos é



em que π_1 e π_2 são respectivamente a primeira e a segunda projecções (também representadas pelas funções fst e snd, respectivamente e em **Haskell**).

As definições *pointwise* das projecções são

$$\begin{aligned} fst\ (a,b) &= a \\ snd\ (a,b) &= b \end{aligned}$$

Dualmente, pense-se em duas funções que partilham o mesmo co-domínio, sejam $f\ :\ B \to A$ e $g\ :\ C \to A$.

O dual do produto de tipos é o co-produto de tipos, pelo que se supõe $h\ :\ B + C \to A$.

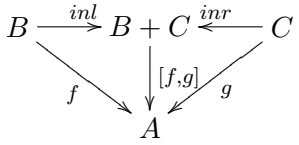
O combinador que constrói a função h a partir das funções f e g é o *either* sendo que também este tem duas representações usuais

- $f\ \backslash/\ g$
- $[\ f,\ g\ ]$

Denominando os injectores do co-produto por *inl* e *inr* com $inl \ : \ B \rightarrow B+C$ e $inr \ : \ C \rightarrow B+C$ tem-se a seguinte definição *pointwise*.

$$\begin{array}{l} [\ f,\ g\ ]\ (inl\ x) = \ f\ x \\ [\ f,\ g\ ]\ (inr\ x) = \ g\ x \end{array}$$

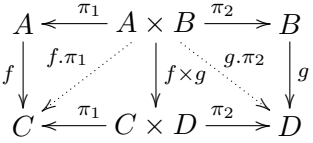
Diagramaticamente, é evidenciada a dualidade



Surge ainda o combinador **produto**, que se representa com o operador $\times$, e que dadas duas funções $f \ : \ A \rightarrow C$ e $g \ : \ B \rightarrow D$ dá a função $f \times g \ : \ A \times B \rightarrow C \times D$, sendo que esta última função pode ser definida recorrendo ao combinador *split* como

$$f \times g \ = \ < \ f.\pi_1,\ g.\pi_2 \ >$$

O diagrama de tipos para este combinador é



e a respectiva definição *pointwise* é

$$(f \times g)\ (a,b) \ = \ (f\ a,\ g\ b)$$

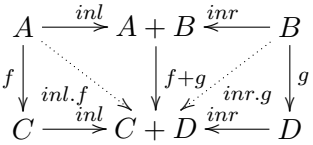
Dualmente, no co-produto, tem-se o combinador **soma**, que se representa pelo operador $+$, e que combina duas funções $f \ : \ A \rightarrow C$ e g : B →D dando uma função $f+g \ : \ A+B \rightarrow C+D$ e cuja definição *point-free* é

$$f+g \ = [\ inl.f,\ inr.g\ ]$$

e em *pointwise* é

$$\begin{array}{l} (f+g)\ (inl\ x) = \ inl\ (f\ x) \\ (f+g)\ (inr\ x) = \ inr\ (g\ x) \end{array}$$

Diagramaticamente vem



Os combinadores já apresentados eram todos binários. Surge agora um combinador unário, que para cada constante a fixada arbitrariamente dá a função constante a. Este combinador representa-se pelo símbolo **!** e a sua definição *pointwise* é

$$a!\ x \ = \ a \qquad \text{ou se se preferir a notação } \lambda \qquad a! \ = \ \lambda\ x.\ a$$

Estão assim introduzidos os combinadores mais usuais. Não quer isto contudo dizer, que este estilo de programação está limitado aos combinadores apresentados.

Mais e novos combinadores poderão sempre ser acrescentados para acomodar padrões frequentes de programas e permitindo reduzir os processos de transformação e/ou prova de propriedades.

Repare-se que para combinadores definidos exclusivamente à custa de outros combinadores, cujas leis já são conhecidas, as respectivas leis surgem rapidamente das leis dos combinadores que lhe servem de base. Como este processo se pode repetir, é possível criar combinadores cada vez mais complexos e cujas leis surgem naturalmente.

Tanto os combinadores apresentados como as funções básicas (aquelas que se identificam pelo respectivo nome) incluindo as especiais, nomeadamente as projecções e os injectores, são expressões funcionais pois quando aplicadas a argumentos comportam-se como funções.

Como já referido, um dos propósitos do estilo *point-free* é a transformação de programas, ou seja, a substituição de um programa por um outro equivalente ao original. Estes passos ocorrem por aplicação de leis - as leis de redução. Devido à falta de variáveis, estas leis são mais facilmente apresentadas e implementadas neste estilo, sendo as leis *point-free* mais genéricas.

Uma das críticas que é apontada a este estilo de programação é a de que é pouco intuitivo o significado dos programas *point-free*, dificultando portanto a sua escrita pelos programadores. Este é um dos motivos que mais tem contribuído para o pouco uso do estilo *point-free*. No entanto, talvez não seja o estilo que seja pouco intuitivo mas o facto de os programadores estarem habituados a pensar de forma diferente que o torne pouco utilizado.

Por todas estas características é vulgar fazer-se a analogia entre o cálculo *point-free* e as transformadas de *Fourier* que apesar de não tão intuitivas se adequam melhor à manipulação.

Antes de terminar esta secção deixa-se aqui um exemplo; o da função que calcula a média dos elementos de uma lista de números reais. A definição *pointwise* é

```
média :: [Float] -> Float
média l = (sum l) / (length l)
```

e a definição *point-free* é

```
média :: [Float] -> Float
média = div . sum /\ length
```

em que `div` é a versão *uncurried* de `/`.

2.2 Combinadores em Haskell: Pointless

Em **Haskell** o combinador composição integra a biblioteca que é importada por defeito e está portanto sempre acessível. Também as funções de projecção fst e snd se encontram nesta biblioteca *standard*.

Quanto aos outros podem ser usados em **Haskell** importando a biblioteca Pointless da autoria de Alcino Cunha e disponível *online* a partir de
http://wiki.di.uminho.pt/twiki/bin/view/PURe/PUReSoftware

Aqui encontram-se definidos os combinadores previamente apresentados com as seguintes representações

– *Split* das funções f e g representa-se `f /\ g`
– *Either* das funções f e g representa-se `f \/ g`
– Produto das funções f e g representa-se `f >< g`
– Soma das funções f e g representa-se `f -|- g`
– Função constante a representa-se `(a!)`

Também os injectores são representados e são-no pelas funções *inl* e *inr*, respectivamente para os injectores à esquerda e à direita.

Nesta mesma biblioteca e ainda no módulo Pointless.Combinators é ainda possível encontrar a função `app :: (a->b, a) -> b`, cujo resultado é o de aplicar a primeira componente do argumento (que é uma função) à segunda componente e ainda o combinador **guarda** que dado um predicado injecta o argumento num co-produto, sendo que a escolha do injector é determinada pelo resultado do predicado sobre o argumento. Este combinador é representado pelo operador infixo ? e sua definição *pointwise* é

$$p?\ x\ =\ \texttt{if}\ (p\ x)\ \texttt{then}\ inl\ x\ \texttt{else}\ inr\ x$$

ou mais claramente

$$p?\ =\ \lambda\ x.\ \texttt{if}\ (p\ x)\ \texttt{then}\ inl\ x\ \texttt{else}\ inr\ x$$

alternativamente, a definição pode ser dada como

$$(p?)\ x\ =\ \begin{cases} p\ x & \Rightarrow \quad inl\ x \\ \neg\ (p\ x) & \Rightarrow \quad inr\ x \end{cases}$$

Ainda nesta biblioteca, mas agora no módulo Pointless.Isomorphisms, estão definidas as funções que testemunham os isomorfismos de tipos mais elementares.

Um isomorfismo de tipos é um par de funções entre os tipos tais que as funções são mutuamente inversas, isto é, sejam os tipos X e Y e as funções $f\ :\ X \to Y$ e $g\ :\ Y \to X$, verifica-se que

$$\begin{cases} f\ .\ g = \ id \\ g\ .\ f = \ id \end{cases}$$

Diz-se então que X é isomorfo a Y e escreve-se $X \cong Y$.

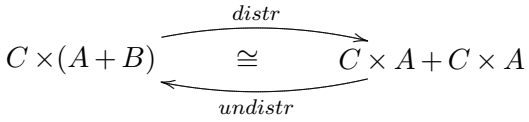
Na perspectiva da programação, dizer que os tipos são isomorfos significa que é possível converter os dados representados num tipo para a representação no outro tipo de dados sem que haja perda de informação.

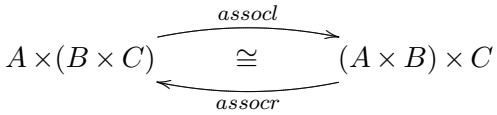
Os isomorfismos definidos então no módulo são:

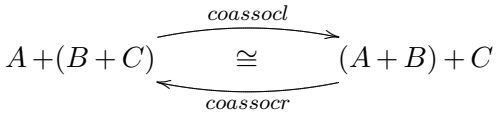
$$A \times B \underset{swap}{\overset{swap}{\cong}} B \times A$$

$$A + B \underset{coswap}{\overset{coswap}{\cong}} B + A$$

$$(A + B) \times C \underset{undistl}{\overset{distl}{\cong}} A \times C + B \times C$$







2.3 Tipos Indutivos

Até agora falou-se de tipos de dados genéricos. Contudo existem certos tipos de dados com propriedades particulares que interessa aqui referir - os tipos de dados indutivos.

Estes tipos de dados podem ser definidos recorrendo a eles próprio, designando-se tal fenómeno por recursividade.

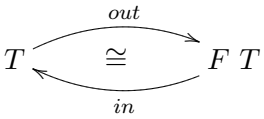
Algo que é conhecido destes tipos de dados é que têm a eles associados um functor de tipo. Sem se querer entrar em muito detalhe, o tipo indutivo é o menor ponto fixo (menor solução) para uma equação envolvendo um functor. Mais exactamente, o functor é uma aplicação quer de tipos para tipos quer de funções para funções que captura a estrutura da definição do tipo.

Um tipo indutivo é isomorfo à aplicação do functor ao próprio tipo, e é o menor tipo nestas condições.

Simbolicamente, para um tipo T e um functor F, escreve-se $T = \mu F$ e $T \cong F\ T$.

Se há um isomorfismo, existem as duas funções que o testemunham; neste caso as funções designam-se *in* e *out*; *in* a função que constrói um elemento do tipo indutivo e *out* a função que "destrói" um elemento do tipo no respectivo isomorfismo.

Simbolicamente



O functor F aplicado ao tipo T é um co-produto de tipos n-ário (um por cada modo de construir um elemento do tipo, ou seja, um por cada construtor).

Por exemplo, para as listas declaradas em **Haskell** como

```
data List = Nil | Cons Int List
```

$F\ List$ será $1 + Int \times List$

Como visto anteriormente, quando o domínio de uma função é um co-produto essa função é um *either*. Generalizando o resultado para o co-produto n-ário e o respectivo *either* n-ário, a função *in* é um *either* cuja aridade é o número de construtores. Mais, substituindo o tipo *List* por um seu isomorfo, seja

```
data List = Nil () | Cons (Int, List)
```

a função é o *either* dos construtores. Para o segundo exemplo das listas virá

$$in = [Nil, Cons]$$

Assim a função *in* é um *either* dos construtores do tipo, salvo o rearranjo dos construtores para a versão *uncurried* e a adição do argumento `()` aos construtores sem argumento.

2.4 Padrões Recursivos

Os programadores que escreviam funções que envolviam tipos de dados indutivos cedo se aperceberam que muitas dessas funções eram muito parecidas, isto é, era possível encontrar padrões que se repetiam entre as diferentes funções.

A partir do momento em que as linguagens passaram a suportar funções de ordem superior, ou seja, funções que recebem outras funções como argumento, capturaram-se esses padrões em funções.

Apresentar-se-ão de seguida os padrões de recursividade, de forma sucinta (ao longo da exposição relembre-se que T é μF)

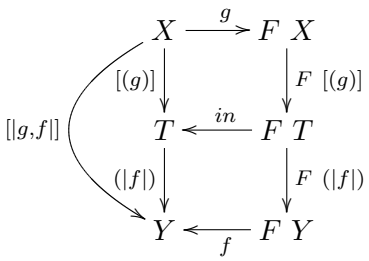
- **Catamorfismo**
  Este padrão depende essencialmente da estrutura do tipo indutivo que toma como argumento.
  Assim pense-se numa função f do tipo indutivo T para qualquer tipo X fixado previamente.
  Sabe-se já que é possível passar do tipo T para $F\ T$ pela função *out* e que o functor se aplica também a funções passando-se assim de $F\ T$ para $F\ X$ por $F\ f$. Suponha-se uma função $g\ :\ F\ X \to X$.
  Diagramaticamente

$$\begin{array}{ccc} T & \xrightarrow{out} & F\ T \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle F\ f} \\ X & \xleftarrow[g]{} & F\ X \end{array}$$

  $F\ f$ mantém a estrutura do functor no tipo, substituindo os elementos do tipo T (aqueles que criam a recursividade) por elementos do tipo `X` que resultam de aplicar a função f aos elementos do tipo T.
  A função g determina a função f pelo que g é designada de gene; gene do catamorfismo f. Deste modo, escreve-se $f\ =\ (|g|)$.
- **Anamorfismo**
  Padrão dual do anterior em que o tipo indutivo que "guia" o padrão é o do resultado da função, isto é, este padrão serve para criar elementos de uma certa estrutura indutiva.
  Sendo f o anamorfismo com $f\ :\ X \to T$ e g o respectivo gene, tem-se o diagrama

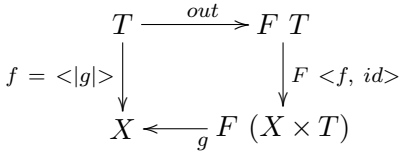
$$\begin{array}{ccc} X & \xrightarrow{g} & F\ X \\ {\scriptstyle f\ =\ [(g)]}\downarrow & & \downarrow{\scriptstyle F\ f} \\ T & \xleftarrow[in]{} & F\ T \end{array}$$

- **Hilomorfismo**
  Basicamente este padrão é a composição de um catamorfismo com um anamorfismo.
  Diagramaticamente



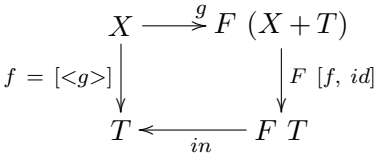
Este padrão tem assim dois genes; o gene g do anamorfismo e o gene f do catamorfismo.
O tipo de dados indutivo que guia o hilomorfismo $[|g, f|] \ : \ X \to Y$ é o da estrutura intermédia (T).

– **Paramorfismo**
Este padrão é uma variação do catamorfismo. No catamorfismo, após se efectuar a chamada recursiva, não é mais possível recuperar a informação da sub-árvore à qual é feita a chamada recursiva. Esta variante efectua a chamada recursiva e mantém a informação da sub-árvore, duplicando a sub-árvore e não aplicando a chamada recursiva à cópia.
Diagramaticamente



Note-se que neste padrão o tipo do gene é mais elaborado $g \ : \ F \ (X \times T) \to X$

– **Apomorfismo**
É uma variação do anamorfismo dual à variação do paramorfismo. Aqui cria-se um co-produto antes das invocações recursivas.
Diagramaticamente



Note-se que neste padrão o tipo do gene é $g \ : \ X \to F \ (X + T)$

– **Mapeamento**
Este padrão é usado para converter toda a informação numa estrutura mas mantendo intocável tal estrutura.
Para a explicação deste padrão é necessária a introdução dos conceitos de bifunctor e de tipos parametrizados. Far-se-á apenas esta introdução por meio do exemplo das listas.
Pense-se nas listas de elementos de um tipo A e nas listas de um tipo C. Em **Haskell** definir-se-ia

```
data List x = Nil | Cons x (List x)
```

Suponha-se uma função de A para C, seja $g \ : \ A \to C$. Como definir a função $f \ : \ List \ A \to List \ C$?
Usando o functor de listas tem-se

$$\begin{array}{ccc} List\ A & \xrightarrow[out_{List\ A}]{} & 1 + A \times (List\ A) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle 1+g\times f} \\ List\ C & \xleftarrow[in_{List\ C}]{} & 1 + C \times (List\ C) \end{array}$$

Para completar o diagrama, surgiu a função $1 + g \times f$.
É a função g que faz realmente o mapeamento, o resto é apenas navegação e preservação da estrutura.
Usando bifunctores, o diagrama anterior é convertido em

$$\begin{array}{ccc} List\ A & \xrightarrow[out_{List\ A}]{} & B(A,\ List\ A) \\ {\scriptstyle f\ =\ map\ g}\downarrow & & \downarrow{\scriptstyle B(g,\ f)} \\ List\ C & \xleftarrow[in_{List\ C}]{} & B(C,\ List\ C) \end{array}$$

A generalização (que apenas se apresenta mas não se justifica) para um tipo indutivo parametrizado (unariamente) T, tipos A e C e com uma função de conversão entre os últimos tipos referidos, $g\ :\ A \to C$ é

$$\begin{array}{ccc} T\ A & \xrightarrow[out_T\ A]{} & B(A,\ T\ A) \\ {\scriptstyle f\ =\ map\ g}\downarrow & & \downarrow{\scriptstyle B(g,\ f)} \\ T\ C & \xleftarrow[in_T\ C]{} & B(C,\ T\ C) \end{array}$$

Como se pretendia uma breve exposição sobre os padrões recursivos omitiu-se a teoria de álgebras, co-álgebras, functores e bifunctores entre outros assuntos. Estes assuntos são abordados de forma global e detalhada em [Oli99b] e [Ven00].

A Biblioteca Pointless e os Tipos Indutivos Relativamente ainda à biblioteca Pointless, de entre os muitos aspectos de interesse, destaca-se ainda a classe `FunctorOf` que permite que o programador associe, a um tipo de dados indutivo, o seu functor e as funções de `in` e `out`.

Criada a instância, a biblioteca equipa o tipo indutivo com o functor, as funções *in* e *out* e ainda os padrões de recursividade mapeamento, catamorfismo, anamorfismo, hilomorfismo, paramorfismo, apomorfismo e zygomorfismo.

3 Representação de Expressões Point-free

3.1 Tipo de Dados

Para se poder criar e manipular expressões *point-free* é necessário primeiro definir uma forma de as representar. Como visto anteriormente, as expressões *point-free* podem ser:

- **função básica** - interessa apenas o seu nome
- **composição** de sub-expressões
- **split** de sub-expressões - relembra-se que o *split* aplica duas funções a um mesmo argumento devolvendo o par de resultados
- **produto** de sub-expressões - duas funções são aplicadas, sendo que cada uma das funções é aplicada à respectiva componente do par dado como argumento
- **either** de sub-expressões - é aplicada uma ou outra das funções consoante o injector do argumento

- **soma** de sub-expressões - idêntica à anterior, só que enquanto a anterior destrói o injector, esta, após aplicar uma das funções, restitui o injector inicial
- **projecções** de um produto - para um par existem duas projecções que se designam vulgarmente primeira e segunda
- **injectores** de um co-produto - tradicionalmente os dois injectores designam-se esquerdo e direito
- **função constante** - permite representar as constantes da linguagem como um número, uma letra ou ainda uma *string* por uma função que descarta o argumento
- a função especial **identidade**
- a função **in** - que constrói um elemento de um tipo de dados indutivo tomando como argumento um elemento do seu functor
- a função **out** - é inversa de *in* e "destrói"um elemento do tipo de dados indutivo num elemento do seu functor aplicado ao tipo

Note-se que estas duas últimas funções são parametrizadas no tipo de dados indutivo (e consequentemente no seu único functor) mas que graças às potencialidades do *point-free* pode-se abstrair essa parametrização.

Mais adiante justificar-se-á a introdução das funções *in* e *out*.

Dado o facto de a linguagem de implementação da ferramenta ser o **Haskell**, estas ideias implementam-se muito directamente no tipo de dados indutivo que se segue.

```
data PFExp = Func String  -- ^ funções
     | Comp   PFExp PFExp -- ^ composição
     | Split  PFExp PFExp -- ^ split
     | Prod   PFExp PFExp -- ^ produto
     | Either PFExp PFExp -- ^ either
     | Sum    PFExp PFExp -- ^ soma
     -- projecções
     | Fst    -- ^ primeira projecção
     | Snd    -- ^ segunda projecção
     -- injectores
     | InL    -- ^ injector esquerdo
     | InR    -- ^ injector direito
     | Const String   -- ^ função constante
     | Id     -- ^ função identidade
     | In     -- ^ função in
     | Out    -- ^ função out
```

3.2 Limitações

Esta representação é contudo limitada na sua expressividade. O facto de representarmos uma função básica apenas pelo seu nome restringe a representação a funções cuja aridade (número de argumentos) é um. Deste modo perde-se a capacidade de expressar funções *curried*, ou seja, funções de N argumentos que são vistas como funções de um argumento que retornam uma nova função de N-1 argumentos.

Também as funções de ordem superior ficam excluídas com esta representação dado que não há informação sobre a(s) função(ões) argumento(s).

Atente-se ainda que com esta representação os construtores são meras funções (obrigatoriamente de aridade um).

Para este caso, em que a linguagem alvo de transformação é o **Haskell**, era possível eliminar os construtores *Fst, Snd, InL, InR, Id, In, Out* utilizando a seguinte tradução:

- Fst = Func "fst"

- Snd = Func "snd"
- InL = Func "Left"
- InR = Func "Right"
- Id = Func "id"
- In = Func "inn"
- Out = Func "out"

Para estas duas últimas traduções é também necessário tomar em consideração a utilização da biblioteca Pointless e algumas das definições nela contidas.

A opção por manter estes construtores ao nível da nossa representação deve-se às facilidades que daí advêm para a manipulação das expressões *point-free*. Além disso torna possível a alteração da linguagem alvo sem que isso implique a alteração do tipo de dados.

3.3 Visualização

Se se tem um tipo de dados, necessita-se de um modo de visualizar os elementos desse tipo. Embora o **Haskell** forneça uma visualização *standard* esta não é a mais simples e intuitiva, pelo que se optou por redefinir a função de visualização por forma a aumentar a semelhança com o que já foi exposto sobre os combinadores *point-free*. Uma das apostas de fundo foi visualizar os combinadores numa versão infixa, isto é, o símbolo que representa o combinador surge entre os seus argumentos (isto para os combinadores binários).

Assim foram tomadas as seguintes escolhas:

- **função básica** - é visualizada pelo respectivo nome
- **composição** de sub-expressões - o símbolo que representa este combinador é $\bullet$
- **split** de sub-expressões - o símbolo que representa este combinador é $/\backslash$
- **produto** de sub-expressões - o símbolo que representa este combinador é $><$
- **either** de sub-expressões - o símbolo que representa este combinador é $\backslash/$
- **soma** de sub-expressões - o símbolo que representa este combinador é $-|-$
- **primeira projecção** de um produto - é visualizada com a palavra **fst**
- **segunda projecção** de um produto - é visualizada com a palavra **snd**
- **injector esquerdo** de um co-produto - é visualizado com a palavra **inl**
- **injector direito** de um co-produto - é visualizado com a palavra **inr**
- **constantes** da linguagem - o símbolo que representa este combinador é **!** e é precedido da palavra que representa a constante
- a função especial **identidade** - é visualizado com a palavra **id**
- a função **in** - é visualizado com a palavra **inn**
- a função **out** - é visualizado com a palavra **out**

O leitor mais atento detectou a proximidade desta visualização com a utilização da biblioteca Pointless. Tal foi intencional para permitir o uso dessa biblioteca.

A versão actual de visualização é muito verbosa no sentido em que coloca demasiados parêntesis. Relembre-se que a definição de diferentes precedências para os operadores permite a omissão de muitos dos parêntesis.

De acordo com as metodologias de programação referidas foi também definido o catamorfismo para este tipo de dados. Contudo surge aqui uma diferença relativamente ao exposto genericamente; enquanto genericamente se considera um único gene que é tipicamente um *either* de funções mais simples, nesta implementação tem-se um tuplo de genes - um por cada construtor do tipo de dados `PFExp`.

Para mais detalhes sobre a tipo de dados `PFExp` assim como a sua visualização e o catamorfismo associado consultar o anexo B onde se encontra a implementação em **Haskell**.

4 Representação Interna do Código Haskell

Sendo pretensão desta ferramenta a conversão de código **Haskell** *pointwise* em código *point-free* são necessários meios para manipular esse código que se encontra em ficheiros. Duas abordagens são possíveis: a primeira consiste em manipulação directa de *strings*, a segunda consiste em usar um *parser* que gere uma árvore abstracta.

Dada a quantidade de manipulações necessárias, a primeira abordagem torna-se desajustada. Usar-se-á então a segunda com o *parser* disponibilizado na biblioteca Language.Haskell.Parser que é actualmente uma biblioteca distribuída com os compiladores e interpretadores de maior utilização. Este *parser* retorna um elemento do tipo `HsModule` que, na actualidade, é a representação *standard* de código **Haskell** e é providenciado pela biblioteca `Language.Haskell.Syntax` (na realidade este elemento é retornado mas encapsulado numa *Monad*, contudo tal não é neste momento relevante para a exposição).

Procurar-se-á dar uma breve explicação da forma como o código **Haskell** é representado neste tipo de dados. No entanto, dada a vastidão deste tipo de dados confinar-se-á a exposição aos que são relevantes no contexto deste projecto.

Um ficheiro de código **Haskell** é considerado um módulo (`HsModule`). Este `HsModule` contém informação sobre o nome do módulo, a localização no ficheiro fonte, a lista de exportações (aquilo que se permite que seja visto e utilizado do exterior do módulo), a lista de módulos importados e aquilo que é realmente relevante que é a lista de declarações/definições do módulo (apenas esta lista será alvo de manipulação neste projecto, sendo os restantes argumentos propagados para o *output*).

Em **Haskell** existem diferentes declarações, o que é testemunhado pelos vários construtores do tipo `HsDecl`, cujos nomes permitem, regra geral, identificar a semântica que lhes está associada.

Assim pode-se ter declarações de tipos de dados, declarações de tipos de dados indutivos, declarações de operadores infixos, declarações de classes, declarações de instâncias de classes, declarações de tipo de expressões (também conhecidas por assinatura de tipo da expressão) e declarações de funções entre outros.

Apenas se revestem de importância os construtores `HsDataDecl`, `HsInstDecl` e `HsFunBind`, respectivamente para as declarações de tipos de dados indutivos, instâncias de classes e funções.

Daqui em diante, os argumentos do tipo `SrcLoc` serão ignorados na exposição tal como o foram no tratamento na ferramenta pois representam informação de posicionamento no ficheiro de código fonte que é de todo desnecessária.

4.1 Tipo de Dados Indutivo

Comece-se pela declaração de tipos de dados indutivos cujo construtor é `HsDataDecl` e tem a seguinte assinatura:

```
HsDataDecl :: SrcLoc -> HsContext -> HsName -> [HsName]
      -> [HsConDecl] -> [HsQName] -> HsDecl
```

O terceiro argumento representa o nome atribuído ao tipo de dados indutivo, o quarto é a lista com os nomes dos parâmetros do tipo indutivo, o quinto e principal é a lista de construtores declarados para o tipo conjuntamente com as respectivas definições e o sexto argumento é a lista de nomes de classes cujas instâncias são automaticamente geradas pelo interpretador/compilador à custa da definição indutiva.

O segundo argumento representa o contexto dos parâmetros do tipo indutivo. Este contexto, que é argumento de outros construtores ao longo de toda a representação, contém informação tal como *o parâmetro* **a** *é um qualquer tipo de dados que pertence à classe* **Eq**. Este contexto pode conter múltiplas (ou nenhuma) informação semelhante à do exemplo, contudo tal é desprezado na manipulação actual.

Saliente-se que o objectivo da análise da definição de um tipo de dados indutivo é a geração de uma instância de classe, processo esse que será descrito posteriormente, confinando-se a exposição actual à semântica da representação de código **Haskell** na própria linguagem.

Quanto ao quinto argumento, do tipo `HsConDecl`, abrange construtores para *records* (actualmente não suportados) e construtores de outros tipos de dados; este último tem como argumentos, e por esta ordem, o nome do construtor e a lista de tipos dos argumentos do construtor definido (esta última referência a *argumentos* situa-se a um nível diferente).

Os tipos em **Haskell** podem ser, ou, dito de outro modo, os construtores do tipo `HsType` são:

- `HsFun Hstype HsType`
- `HsTyTuple [HsType]` - tuplos tendo como argumento a lista dos tipos das componentes do tipo (atenção pois o argumento na frase anterior refere-se ao do construtor `HsTyTuple` enquanto as componentes referidas estão a outro nível e dizem respeito às componentes do tipo que é um tuplo e é representado no tipo `HsType`)
- `HsTyApp HsType HsType` - aplicação de tipos; o primeiro argumento é aplicado ao segundo. O exemplo mais usual são as listas de inteiros com a aplicação de `[]` a `Int` para se obter o tipo `[Int]`.
- `HsTyVar HsName` - variáveis de tipo
- `HsTyCon HsQName` - construtores de tipo. Por exemplo `[]` para as listas nativas do **Haskell**.

Talvez o leitor já tenha reparado na semelhança dos nomes `HsName` e `HsQName`. Na verdade ambos os tipos servem para representar nomes de todo e qualquer elemento no código **Haskell** distinguindo-se por o segundo poder representar nomes qualificados, ou seja, nomes que são considerados não globalmente mas apenas no domínio de nomes do módulo (possivelmente hierárquico) indicado como argumento no construtor `Qual`.

Tanto para as declarações de tipos de dados indutivos que já se descreveu como para as declarações de instâncias de classes e de funções que se descreverão de imediato, omitiram-se e omitir-se-ão muitos pormenores por forma a abreviar a exposição. Contudo, os mais curiosos podem consultar a documentação da biblioteca `Language.Haskell.Syntax`, disponível *online* em
http://www.haskell.org/ghc/docs/latest/html/libraries/haskell-src/Language.Haskell.Syntax.html

4.2 Instância de Classe

As declarações de instâncias de classes representam-se com o construtor `HsInstDecl` cuja assinatura é:

```
HsInstDecl :: SrcLoc -> HsContext -> HsQName -> [HsType]
         -> [HsDecl] -> HsDecl
```

O terceiro argumento é o nome da classe para a qual se destina a instância, o quarto argumento é a lista de tipos para os quais se cria a instância (a lista não significa que se estão a criar instâncias para cada um dos tipos listados mas serve para acomodar as classes multi-parâmetro) e o quinto argumento é a lista com as declarações para a instância (tipicamente as únicas declarações que aqui se encontram são as assinaturas de tipo e as definições das funções da classe).

4.3 Função

Relativamente às definições de funções, estas são representadas pelo construtor `HsFunBind` que tem um único argumento que é uma lista de `HsMatch`. Este último tipo tem um só construtor, com o mesmo nome que o tipo - `HsMatch`, com a assinatura de tipo que se segue:

```
HsMatch :: SrcLoc -> HsName -> [HsPat] -> HsRhs -> [HsDecl]
       -> HsMatch
```

O segundo argumento é o nome da função, o terceiro é a lista de argumentos (estes argumentos estão num nível diferente pois são os argumentos representados), havendo lugar a *pattern-matching*, o quarto argumento corresponde à definição propriamente dita, vulgarmente designada **lado direito**, e o quinto argumento é uma lista de sub-declarações que são locais à definição actual (no ficheiro de código **Haskell** são as declarações que surgem após a palavra **where**.

Veja-se o exemplo:

```
length [] = 0
length (h:t) = 1 + length t
```

Este bloco de código surge representado por um `HsFunBind` e cada uma das linhas de código é representada por um `HsMatch` com `[]` e `(h:t)` padrões (HsPat) e `0` e `1 + length t` os lados direitos respectivamente para a primeira e a segunda linhas. Neste exemplo não há sub-declarações.

Esta aproximação é análoga à definição de funções por ramos na matemática, sendo que aqui o papel de ramos é desempenhado pelos `HsMatch`'s.

As definições propriamente ditas (tipo `HsRhs`) suportam a representação de guardas mas estas não são actualmente suportadas pela ferramenta.

Um exemplo muito simples de guardas é

```
max n m | n>=m = n
        | n<m = m
```

cujo significado é *o resultado de aplicar a função* `max` *aos argumentos* `n` *e* `m` *é* `n` *se se verificar a guarda* `n>=m` *e é* `m` *se se verificar a guarda* `n<m`.

Em qualquer dos casos (com ou sem guardas) surgem as expressões (no caso de guardas após a condição) com o corpo da definição. Estas expressões são representadas pelo tipo `HsExp` cujos construtores mais relevantes são:

- **HsVar** - variáveis
- **HsCon** - construtores. Por exemplo, o `EQ` do tipo `Ordering`
- **HsLit** - literais. Por exemplo, `1`, `'a'`, `"casa"` são três literais.
- **HsInfixApp** - operadores infixos. por exemplo `h:t` em que `:` é o operador infixo e `h` e `t` são as sub-expressões respectivamente à esquerda e à direita.
- **HsApp** - aplicação de uma função a uma expressão
- **HsNegApp** - sinal de - unário
- **HsLambda** - expressões λ
- **HsLet** - notação `let ... in ...`
- **HsIf** - notação `if ... then ... else ...`
- **HsCase** - notação `case ... of ...`
- **HsDo** - notação `do ...`
- **HsTuple** - tuplos com um número arbitrário de componentes
- **HsList** - listas por extensão
- **HsParen** - parêntesis

- **HsLeftSection** - versão *curried* para os operadores infixos em que o argumento à direita está em falta
- **HsRightSection** - versão *curried* para os operadores infixos em que o argumento à esquerda está em falta
- **HsEnumFrom** - enumerações; por exemplo [1..]
- **HsEnumFromTo** - enumerações; por exemplo [1..10]
- **HsEnumFromThen** - enumerações; por exemplo [1,3..]
- **HsEnumFromThenTo** - enumerações; por exemplo [1,3..10]
- **HsListComp** - listas por compreensão
- **HsExpTypeSig** - expressões com o tipo explícito
- **HsAsPat** - suporte para dupla referencia em que a segunda pode conter padrões. Por exemplo `l@(h:t)`
- **HsWildCard** - o símbolo _ em **Haskell**

A ferramenta, tal como se encontra actualmente, não suporta grande parte destes construtores, mas é desde já objectivo, a médio prazo, alargar o leque de expressões suportadas.

Para os padrões tem-se um conjunto mais reduzido de construtores mas cada construtor para padrões tem um correspondente em `HsExp`. A proximidade dos nomes deixa facilmente adivinhar essa correspondência. Assim é sempre possível converter um padrão (`HsPat`) numa expressão (`HsExp`).

Neste momento já se sabe representar tanto o código **Haskell** como as expressões *point-free* mas como fazer a conversão? Será sempre possível fazê-la? Se não, em que situações?

Este é o tema que se abordará de seguida.

5 Teoria de Conversão

Tal como para qualquer outro problema, o processo de conversão pode ser partido em sub-problemas aplicando assim o lema "*divide to conquer*".

Apresentam-se de seguida as várias fases em que se "partiu" a conversão.

5.1 1º Passo

A definição de uma função consiste tipicamente num conjunto de ramos (no sentido matemático de funções definidas por ramos). Cada ramo contém um lado esquerdo e um lado direito (relativamente ao sinal de `=`) e em ambos os lados podem ocorrer variáveis, variáveis essas que se pretende remover.

Foque-se a atenção num dos lados (é neste momento irrelevante se o esquerdo ou o direito, pois sob certa perspectiva eles são iguais).

Podem-se encontrar expressões como

```
f 1

g x

h (2,3)

j (Just 1)
```

com `f`,`g`, `h` e `j` funções de um argumento, `x` uma variável e `Just` um construtor de aridade um.

Por exemplo, `g x` significa a aplicação da função `g` à variável `x`. Mas `g`, por si só é já uma expressão *point-free*. Pode-se então converter `g x` na aplicação da expressão *point-free* `g` à variável `x`.

Analogamente, tentar-se-á converter cada expressão *pointwise* na aplicação de uma expressão *point-free* a uma variável ou a várias variáveis.

Surge então um primeiro algoritmo, em que se designa por **PF** a função de conversão de *pointwise*. Para cada expressão *pointwise* este algoritmo converte-a num par ordenado cuja primeira componente é uma expressão *point-free* e a segunda componente é uma ou mais variáveis. O significado deste par é que a expressão *pointwise* argumento é equivalente à aplicação da expressão *point-free* resultado à(s) variável(eis) que são a segunda componente deste par.

PF é definida então como se segue:

1. Para uma variável tem-se que é a aplicação da função identidade à variável em causa.

$$PF\ x\ =\ (id,\ x)$$

2. Para uma constante tem-se que é a aplicação de uma função, que devolve sempre o valor k, a um argumento fantasma.

$$PF\ k\ =\ (const\ k,\ \_)$$

3. Para a aplicação de uma função básica de aridade um, f, a uma sub-expressão, exp, e assumindo que se conhece a conversão da sub-expressão, tem-se a aplicação da composição da função com a expressão *point-free* (que advém da conversão da sub-expressão) às variáveis da, já referida, sub-expressão.

$$\begin{aligned} PF\ (f\ exp)\ =& let\ (g,x)\ =\ PF\ exp \\ & in\ (f\ .\ g,\ x) \end{aligned}$$

4. Para um par ordenado tem-se a aplicação do produto das expressões *point-free*, resultantes das conversões das componentes, ao par de variáveis cujas componentes advêm das sub-conversões.

$$\begin{aligned} PF\ (left, right)\ =& let\ (f,x)\ =\ PF\ left \\ & \quad\ (g,y)\ =\ PF\ right \\ & in\ (\ f \times g,\ (x,y)\ ) \end{aligned}$$

Reunindo-se esta informação, obtém-se o algoritmo seguinte

$$\begin{aligned} PF\ x\ &= (id,\ x) \\ \\ PF\ k\ &= (const\ k,\ \_) \\ \\ PF\ (f\ exp)\ &= let\ (g,x)\ =\ PF\ exp \\ &\quad\ in\ (f\ .\ g,\ x) \\ \\ PF\ (left, right) &= let\ (f,x)\ =\ PF\ left \\ &\quad\ (g,y)\ =\ PF\ right \\ &\quad\ in\ (\ f \times g,\ (x,y)\ ) \end{aligned}$$

com x uma variável, k uma constante, f uma função básica de aridade um e exp, $left$ e $right$ expressões quaisquer.

Desde já se excluem funções de aridade diferente de um e tuplos que não sejam pares ordenados. Contudo é sempre possível codificar tuplos como pares ordenados encadeados, isto é, $(a_1, a_2, \ldots, a_n)$ com $n \in \mathbb{N}$ e $n > 2$ pode ser convertido em $(a_1, (a_2, \ldots (a_{n-1}, a_n) \ldots))$ e a conversão pode ser efectuada em ambos os sentidos sem perda de informação; por outras palavras, os tuplos genéricos são isomorfos aos pares encadeados à direita.

Também para as funções de aridade superior a um é possível contornar a limitação passando uma função básica de aridade $n \in \mathbb{N}$, seja $f\ a_1\ a_2\ \ldots\ a_n$, para uma função básica $f'\ (a_1, a_2, \ldots, a_n)$ que, pelo já exposto, passa a $f''\ (a_1, (a_2, \ldots (a_{n-1}, a_n) \ldots))$.

Para as funções básicas de aridade zero, a solução que permite a aplicação do algoritmo anterior é torná-las funções de aridade um cujo argumento é o `()`.

Ultrapassar estas limitações significa que o utilizador tem de fazer manualmente as alterações sugeridas, por forma a que o algoritmo PF conduza a resultados. Talvez em versões futuras se consiga internalizar e automatizar modos de lidar com tais situações, poupando, ao programador, esforços complementares.

Retome-se a análise a um ramo de uma definição de uma função, com o seus lados esquerdo e direito, que tem forma $lhs = rhs$, aplica-se a função de conversão **PF** a ambos os membros e obtém-se

$$(f, e) = PF\ lhs$$
$$(g, e') = PF\ rhs$$

Note-se que f e g já são expressões *point-free* mas e e e' são variáveis ou pares, possivelmente encadeados, de variáveis.

Substituindo-se na igualdade $lhs = rhs$ e relembrando a semântica associada ao par retornado por PF vem

$$f\ e\ =\ g\ e'$$

Se e for igual a e', o problema está resolvido e vem $f = g$. Contudo, no caso geral, e e e' são diferentes. Contudo sabe-se que no lado direito de um ramo só podem aparecer variáveis referidas pelo lado esquerdo, sem que no entanto haja obrigatoriedade de usar no lado direito todas as variáveis do lado esquerdo, isto é, uma variável que ocorre no lado direito tem necessariamente de ocorrer no lado esquerdo mas uma variável pode ocorrer no lado esquerdo e não ocorrer no lado direito. Pode-se então afirmar que *o conjunto das variáveis que ocorrem no lado direito está contido no conjunto das variáveis que ocorrem no lado esquerdo.*

Suponha-se que era possível escrever e' como $variableArrange\ e$, com $variableArrange$ uma função.

Atendendo a esta suposição, a igualdade $f\ e\ =\ g\ e'$ passa a

$$f\ e\ =\ g\ (variableArrange\ e)$$

que se pode ainda reescrever, usando a definição de composição de funções, como

$$f\ e\ =\ (g\ .\ variableArrange)\ e$$

Usando a definição matemática de igualdade de funções e sendo e uma variável qualquer, e portanto passível de ser quantificada universalmente, vem

$$f\ =\ (g\ .\ variableArrange)$$

É pois agora necessário efectivar a suposição, definindo a função *variableArrange* que garante a igualdade $e' = variableArrange\ e$. O que se fará de seguida é apresentar as funções que geram a definição da função *variableArrange* que é necessária em cada ramo de uma função, sendo que esta geração depende de ambos e e e'.

5.2 Rearranjo das Variáveis

O processo de geração da função *variableArrange* desenrola-se em duas partes:

1. Tal como e, e' pode ser uma variável simples ou um par em que uma ou as duas componentes podem ser novos pares ou variáveis. Assim
   - Para o caso em que e' é uma variável simples, basta saber como aceder a essa variável em e. Chame-se a esta função que acede à variável x $acede_x$ e que apenas depende da estrutura e posição da variável x no lado esquerdo.
   - Para o caso em que e' é um tuplo $(e'_1,\ e'_2)$ descubra-se as funções *variableArrange* de cada uma das coordenadas para e. Sejam estas funções vA_1 e vA_2. Tem-se que

     $$\begin{aligned} e'_1 &= vA_1\ e \\ e'_2 &= vA_2\ e \end{aligned}$$

     de onde $(e'_1,\ e'_2) = (vA_1\ e,\ vA_2\ e)$ e que é o mesmo que

     $$e' = (e'_1,\ e'_2) = (vA_1\ /\backslash\ vA_2)\ e$$

     Algoritmicamente tem-se:

     $$\begin{aligned} vA\ x &= acede_x\ e \\ vA\ (a,b) &= let\ a' = vA\ a \\ &\qquad\quad b' = vA\ b \\ &\quad in\ a'\ /\backslash\ b' \end{aligned}$$

     O argumento de vA é a parte das variáveis do lado direito do ramo e e é a variável ou tuplo de variáveis do lado esquerdo do mesmo ramo (note-se que após a aplicação do algoritmo PF foi possível separar a parte *point-free* da parte com variáveis nos lados esquerdo e direito). A função vA devolverá a definição da função *variableArrange* e, apesar de não explicitamente expresso no algoritmo, vA depende das variáveis do ramo tanto no lado esquerdo como no direito (indicadas com e e e' respectivamente); note-se que, por abuso de linguagem, se refere e na definição acima sem que e surga nos argumentos.
2. Na definição acima surgiu esta nova função $acede_x$ que gera a definição da função *variableArrange* fixada a variável x do lado direito e que depende da(s) variável(eis) no lado esquerdo. O algoritmo implementado pela função $acede_x$ é:
   - Se se quer descobrir como aceder a uma variável x e o lado esquerdo é uma variável, uma das duas alternativas seguintes ocorrerá:
     - Ou é a mesma variável x e a função que retorna a variável x do lado esquerdo é a identidade
     - Ou as variáveis são diferentes, o que significaria que existia no lado direito uma variável não existente no lado esquerdo. Como visto anteriormente tal não sucede.

– Se se quer descobrir como aceder a uma variável x e o lado esquerdo é um tuplo, tendo em atenção que a variável x existe obrigatoriamente no lado esquerdo e só ocorre uma única vez (em **Haskell** uma variável só pode ocorrer uma vez no lado esquerdo de um ramo de uma função; isto é conhecido por padrões lineares), conclui-se que:
  - se a variável x aparece algures na primeira componente do tuplo, a função que lhe acederá será a composição da função que acede a x na primeira componente com a projecção da primeira componente.
  - Quando a variável x ocorre na segunda componente, o resultado é análogo só que agora usa-se a segunda projecção.

Isto pode representar-se como

$$\begin{array}{lll} acede_x\ x & = & id \\ acede_x\ (a,b) \mid x \text{ ocorre em } a & = & (acede_x\ a)\ .\ fst \\ \quad\quad\quad\quad\ \mid x \text{ ocorre em } b & = & (acede_x\ b)\ .\ snd \end{array}$$

Note-se que id não significa que $acede_x\ x$ é uma função que aguarda outro argumento e o devolve mas sim que a definição da função $variableArrange$ que se pretende gerar é , neste caso, a função identidade.
O contexto em que esta função $acede_x$ é utilizada, garante que os casos que foram propositadamente omitidos não ocorrem.

Conclui-se então que o rearranjo das variáveis depende, numa primeira fase, da estrutura e posicionamento das mesmas no lado direito e que sempre que se encontra uma variável simples no lado direito, a segunda fase apenas depende da estrutura e posicionamento da variável em causa no lado esquerdo.

Garantida que está a suposição, tem-se, no estado actual e para cada função, um conjunto de igualdades entre expressões *point-free*.

5.3 Junção dos Vários Ramos

O objectivo é agora manipular estas expressões até que as várias igualdades se fundam numa só, em que o lado esquerdo é unicamente o nome da função básica. Nesta altura a função básica cuja definição *pointwise* foi dada será definida, equivalentemente, pela expressão *point-free* calculada.

Suponha-se que se tem um tipo indutivo

$$IndType\ :=\ Const_1\ Type_1 \mid \ \ldots\ \mid Const_n\ Type_n$$

em que $n \in \mathbb{N}$, $Const_i$ são construtores de aridade um, para $i \in \{1,\ldots,n\}$ e $Type_i$ para $i \in \{1,\ldots,n\}$ são tipos de dados. Em **Haskell** viria

```
data IndType = Const_1 Type_1 | ... | Const_n Type_n
```

1. Suponha-se ainda que se tem uma função básica f total e cujo argumento é do tipo indutivo definido acima, ou seja, `f :: IndType -> TypeA` com `TypeA` um tipo de dados arbitrariamente fixado.
   Sendo f uma função total, no caso geral, terá um ramo por cada construtor do tipo indutivo argumento.
   Suponha-se a seguinte definição *pointwise* para f

$$\begin{array}{l} f\ (Const_1\ x_1)\ =\ rhs_1 \\ f\ (Const_2\ x_2)\ =\ rhs_2 \\ \vdots \\ f\ (Const_n\ x_n)\ =\ rhs_n \end{array}$$

com $x_1, x_2, \ldots, x_n$ variáveis ou tuplos, possivelmente encadeados, de variáveis e $rhs_1, rhs_2, \ldots, rhs_n$ expressões.
Aplicando as transformações propostas até agora surgiria

$$\begin{aligned} f\ .\ Const_1 &= rhsP_1\ .\ vA_1 \\ f\ .\ Const_2 &= rhsP_2\ .\ vA_2 \\ &\vdots \\ f\ .\ Const_n &= rhsP_n\ .\ vA_n \end{aligned}$$

tais que $rhs_i = (rhsP_i\ .\ vA_i)\ x_i$ e $rhsP_i$ e vA_i são expressões *point-free* para todo o $i \in \{1, \ldots, n\}$.
Tendo em conta a igualdade estrutural do *either* n-ário, obtém-se

$$\begin{aligned} &(f.Const_1) \backslash/ (f.Const_2) \backslash/ \ldots \backslash/ (f.Const_n) = \\ &= (rhsP_1.vA_1) \backslash/ (rhsP_2.vA_2) \backslash/ \ldots \backslash/ (rhsP_n.vA_n) \end{aligned}$$

Aplicando, agora, a generalização n-ária da fusão-+ em sentido inverso, vem

$$\begin{aligned} &f.(Const_1 \backslash/ Const_2 \backslash/ \ldots \backslash/ Const_n) = \\ &= (rhsP_1.vA_1) \backslash/ (rhsP_2.vA_2) \backslash/ \ldots \backslash/ (rhsP_n.vA_n) \end{aligned}$$

Tomando a definição da função *in* de um tipo de dados indutivo, que é o *either* dos construtores, surge

$$f.in = (rhsP_1.vA_1) \backslash/ (rhsP_2.vA_2) \backslash/ \ldots \backslash/ (rhsP_n.vA_n)$$

Compondo ambos os lados com a função *out*, aparece

$$f.in.out = \Big((rhsP_1.vA_1) \backslash/ (rhsP_2.vA_2) \backslash/ \ldots \backslash/ (rhsP_n.vA_n)\Big)\ .\ out$$

Como *in* e *out* são funções mutuamente inversas e a identidade é o elemento neutro da composição, vem

$$f = \Big((rhsP_1.vA_1) \backslash/ (rhsP_2.vA_2) \backslash/ \ldots \backslash/ (rhsP_n.vA_n)\Big)\ .\ out$$

Já se sabe então como transformar algumas definições *pointwise* na equivalente definição *point-free*, mas só no caso bastante particular de aplicação da função básica a cada um dos construtores do tipo indutivo que é argumento.

2. Pense-se agora em se relaxar as condições anteriores, por exemplo, pondere-se o caso que, após o passo de eliminação de variáveis, dá

$$\begin{aligned} f\ .\ Const_1\ .\ Const_{1,1} &= pfe_{1,1} \\ f\ .\ Const_1\ .\ Const_{1,2} &= pfe_{1,2} \\ &\vdots \\ f\ .\ Const_1\ .\ Const_{1,n_1} &= pfe_{1,n_1} \\ &\vdots \\ f\ .\ Const_m\ .\ Const_{m,1} &= pfe_{m,1} \\ f\ .\ Const_m\ .\ Const_{m,2} &= pfe_{m,2} \\ &\vdots \\ f\ .\ Const_m\ .\ Const_{m,n_m} &= pfe_{m,n_m} \end{aligned}$$

com $Const_1, \ldots, Const_m$ todos os construtores de um tipo de dados indutivo e para cada $i \in \{1, \ldots, m\}$ $Const_{i,1}, \ldots, Const_{i,n_i}$ são todos os construtores de um tipo indutivo.
Se se aplicar a lei anterior m-vezes, mas instanciando f com $f.Const_k$ para $k \in \{1, \ldots, m\}$, obtém-se

$$\begin{aligned} f.Const_1 &= (pfe_{1,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{1,n_1}) \cdot out \\ &\vdots \\ f.Const_m &= (pfe_{m,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{m,n_m}) \cdot out \end{aligned}$$

Novamente se está perante um caso onde é possível aplicar a mesma lei, obtendo-se desta vez

$$\begin{aligned} f = & \\ & \Big(\big(pfe_{1,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{1,n_1} \big).out \Big) \mathbin{\backslash/} \ldots \mathbin{\backslash/} \Big(\big(pfe_{m,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{m,n_m} \big).out \Big) \\ & \cdot out \end{aligned}$$

Generalizando a situação para a composição de mais do que dois construtores, desde que agrupando os ramos correctamente, e indo eliminando os construtores mais à direita nas composições do lado esquerdo torna-se já tratável a conversão para *point-free* de uma vastidão de casos.

3. Os únicos casos que ainda levantam problemas são aqueles em que existem produtos no lado esquerdo. Veja-se o caso, após a eliminação de variáveis

$$\begin{aligned} f \cdot ConstA_1 \times ConstB_1 &= pfe_{1,1} \\ f \cdot ConstA_1 \times ConstB_2 &= pfe_{1,2} \\ &\vdots \\ f \cdot ConstA_1 \times ConstB_n &= pfe_{1,n} \\ &\vdots \\ f \cdot ConstA_m \times ConstB_1 &= pfe_{m,1} \\ f \cdot ConstA_m \times ConstB_2 &= pfe_{m,2} \\ &\vdots \\ f \cdot ConstA_m \times ConstB_n &= pfe_{m,n} \end{aligned}$$

com $ConstA_i$ com $i \in \{1, \ldots, m\}$ todos os construtores de um tipo de dados indutivo, $ConstB_j$ com $j \in \{1, \ldots, n\}$ todos os construtores de outro tipo de dados indutivo (nada impede que sejam ambos o mesmo) e $pfe_{i,j}$ expressões *point-free* para $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$.
À partida pensar-se-ia que um *either* dos lados direitos, com a devida associatividade e o produto das funções *out* resolveria o problema. Mas $out \times out$ retorna um produto de co-produtos, provavelmente co-produtos de aridades diferentes, e os *eithers* dos lados direitos esperariam *eithers* de pares.
Pretende-se portanto converter a informação na forma

$$(A_1 + \ldots + A_m) \times (B_1 + \ldots + B_n)$$

em informação na forma

$$A_1 \times B_1 + \ldots + A_1 \times B_n + \ldots + A_m \times B_1 + \ldots + A_m \times B_n$$

Aplicando a distributividade "n-ária" à esquerda à expressão inicial, vem

$$A_1 \times (B_1 + \ldots + B_n) + \ldots + A_m \times (B_1 + \ldots + B_n)$$

Aplicando agora a distributividade "n-ária' à direita a cada parcela mais externa, vem

$$A_1 \times B_1 + \ldots + A_1 \times B_n + \ldots + A_m \times B_1 + \ldots + A_m \times B_n$$

que é o que se pretendia.
A transformação é então:

$$\begin{aligned} f \ = & (pfe_{1,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{1,n}) \mathbin{\backslash/} \ldots \mathbin{\backslash/} (pfe_{m,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{m,n}) \\ & . \underbrace{NdistR + \ldots + NdistR}_{m \text{ parcelas}} \\ & . \ MdistL \\ & . \ out_A \times out_B \end{aligned}$$

E assim se tem meios para converter uma série de definições *pointwise* nas suas equivalentes em *point-free*.

5.4 Geração das Funções *in* e *out*

Na lei anterior surge a expressão $out_A \times out_B$. Porém os elementos em índice podem ser removidos com o uso de classes. De facto tal já sucede na biblioteca Pointless que define uma classe FunctorOf, cujas instâncias têm de declarar o functor do tipo de dados (indutivo, claro está) e as funções *in* e *out* desse tipo indutivo. No entanto, é da responsabilidade do programador criar a instância para cada um dos tipos de dados indutivos que usar.

Sabe-se, contudo, que um tipo de dados indutivo regular determina univocamente o functor que lhe está associado, isto é, para cada tipo indutivo existe um único functor.

Pretende-se explorar esta propriedade gerando, a partir da definição de um tipo indutivo e de modo automático, a instância de FunctorOf, criando para o programador a mesma facilidade que a escrita no código de `deriving Eq` para a geração automática das funções de igualdade.

Atente-se então num tipo de dados indutivo genérico $IndType$

$$IndType\ p_1 \ldots p_l \ := \ Const_1\ T_{1,1} \ldots T_{1,n_1} \mid \ldots \mid Const_m\ T_{m,1} \ldots T_{m,n_m}$$

com $l \in \mathbb{N}_0$, $m \in \mathbb{N}$, $n_1, \ldots, n_m \in \mathbb{N}_0$, p_k parâmetros do tipo para $k \in \{1, \ldots, l\}$, $Const_i$ construtores do tipo indutivo para $i \in \{1, \ldots, m\}$ e para $i \in \{1, \ldots, m\}$ e $j \in \{1, \ldots, n_i\}$ $T_{i,j}$ tipos de dados que podem ser:

- variáveis de tipo, isto é, podem ser p_j para algum $j \in \{1, \ldots, l\}$
- ou construtores de tipo
- ou o tipo definido $(IndType\ p_1 \ldots p_l)$
- ou ainda tuplos, possivelmente encadeados, das três alternativas anteriores

Ficam assim excluídas as funções de tipo e as aplicações de tipos.

O functor associado a $IndType\ p_1 \ldots p_l$ será um co-produto de aridade m - o número de construtores do tipo indutivo. Simbolicamente escreve-se

$$X \;=\; U_1 + \ldots + U_m$$

Note-se que por abuso de linguagem se utilizam os símbolos + e × que também são utilizados nas expressões *point-free* sem que no entanto sejam o mesmo; estes são operadores entre tipos e os anteriores são operadores entre expressões *point-free*.

Para cada construtor $Const_i$ com $i \in \{1, \ldots, m\}$ os seus n_i argumentos (de tipos respectivamente $T_{i,1}, \ldots, T_{i,n_i}$) dão origem a um produto com n_i componentes. Uma excepção surge no entanto para os construtores de aridade zero ($n_i = 0$), caso em que se cria um argumento que é o 1 dos tipos de dados (o `()` no **Haskell**).

Assim para cada $i \in \{1, \ldots, m\}$

1. Se n_i é zero então U_i é 1
2. Se n_i é maior do que zero então U_i é $V_{i,1} \times \ldots \times V_{i,n_i}$ (o caso $n = 1$ é considerado aqui)
   Em que cada $V_{i,j}$ deriva de $T_{i,j}$ para todo $i \in \{1, \ldots, m\}$ e $j \in \{1, \ldots, n_i\}$ de acordo com as regras que se seguem:
   (a) Se $T_{i,j}$ é $IndType\ p_1 \ldots p_l$ então $V_{i,j}$ não é senão X; representa a recursividade do tipo.
   (b) Se $T_{i,j}$ é p_k para algum $k \in \{1, \ldots, l\}$ então $V_{i,j}$ é p_k
   (c) Se $T_{i,j}$ é um construtor de tipos de aridade zero (por exemplo, `Int`) então $V_{i,j}$ é $T_{i,j}$.
   (d) Se $T_{i,j}$ é um produto de tipos de aridade q então $V_{i,j}$ é um produto de q factores, em que a cada um deles são aplicadas as regras 2a a 2d, isto é, $V_{i,j}$ é $W_{i,j,1} \times \ldots \times W_{i,j,q}$ em que, tomando $T_{i,j}$ como $Y_{i,j,1} \times \ldots \times Y_{i,j,q}$, cada $W_{i,j,h}$ deriva de $Y_{i,j,h}$ para todo o $h \in \{1, \ldots, q\}$ usando as regras de derivação 2a a 2d.

Eis então como determinar o functor de um tipo de dados indutivo usando a definição do tipo.

Falta pois determinar as funções *in* e *out* que "passam", respectivamente, do "functor" para o tipo de dados e vice-versa. Contudo, e porque as funções são inversas e os índices l,m e n_i para todo o $i \in \{1, \ldots, m\}$ são finitos, tomando as funções como um conjunto de pares ordenados $(argumento, resultado)$, uma das funções é a outra trocando a ordem das componentes para todos os pares do conjunto.

Então analisar-se-á apenas a geração da função *out*, sendo a geração da função *in* análoga.

Suponha-se o tipo de dados indutivo anterior (e nas mesma condições), $IndType$.

Para cada construtor $Const_i$ cria-se o respectivo injector inj_i no co-produto de aridade m.

Simbolicamente

$$\begin{aligned} out\ (Const_i\ arg_{1,1}\ \ldots\ arg_{1,n_1}) &= inj_1\ b_1 \\ &\vdots \\ out\ (Const_m\ arg_{m,1}\ \ldots\ arg_{m,n_m}) &= inj_m\ b_m \end{aligned}$$

com $arg_{i,j}$ argumentos do tipo $T_{i,j}$ para $i \in \{1, \ldots, m\}$ e $j \in \{1, \ldots, n_i\}$.

Para cada $i \in \{1, \ldots, m\}$, b_i é dado por:

1. Se n_i é zero (o construtor $Const_i$ não tem argumentos) então b_i é a constante ().
   Note-se que () é o único elemento do tipo 1 que em **Haskell** é representado por ().

2. se n_i é maior do que zero então b_i será um tuplo de n_i componentes (este caso também considera um tuplo de uma única componente), seja $b_i = (c_{i,1}, \ldots, c_{i,n_i})$ sendo que, para cada $j \in \{1, \ldots, n_i\}$, $c_{i,j}$ é:
   (a) Se $arg_{i,j}$ é uma variável simples então $c_{i,j}$ é $arg_{i,j}$
   (b) Se $arg_{i,j}$ é um tuplo, seja $arg_{i,j} = (d_{i,j,1}, \ldots, d_{i,j,q})$ então $c_{i,j}$ é $(e_{i,j,1}, \ldots, e_{i,j,q})$ sendo que para cada $h \in \{1, \ldots, q\}$ a $d_{i,j,h}$ aplica-se a regra 2a ou a 2b para obter $e_{i,j,h}$

Neste momento já é possível gerar automaticamente toda uma instância da classe FunctorOf para um tipo de dados indutivo.

6 Conversão em Funcionamento

O primeira problema que surge quando se começa a implementar a conversão de acordo com as ideias já apresentadas é a parcialidade do processo; a conversão só é possível em alguns casos (as limitações são referidas ao longo deste relatório). E o que fazer quando um desses passos de conversão falha?

Tendo em atenção que a linguagem de implementação é o **Haskell** as *Monads* são a resposta óbvia para resolver esta situação.

Assim uma função que implementa um passo de conversão e tem tipo `A -> B` mas que pode falhar, passa a ter tipo `A -> MyMonad B` em que `MyMonad` é uma *Monad* a definir. Esta aproximação monádica permite a criação explícita de erros e trata automaticamente da propagação dos mesmos.

Inicialmente o objectivo da utilização de uma *Monad* era o de implementar apenas a parcialidade do processo, adequando-se para este fim a *Monad* `Maybe`. Contudo com o evoluir da implementação surgiu a necessidade de incluir também informação global. Com apenas algumas modificações (principalmente a redefinição de `MyMonad`) passou-se da *Monad* `Maybe` para a *Monad* `StateT St Maybe` com `St` um tipo de dados para acomodar a informação global (a *Monad* `StateT a m` está definida na biblioteca Control.Monad.State).

Os erros vão sendo propagados através da *Monad* mas a um nível superior eles têm de ser tratados. Onde?

Um ficheiro de código contém várias definições de funções e de tipos de dados indutivos. A conversão de uma função não depende de outra pelo que o âmbito dos erros deve ser restrito à definição onde este ocorre.

Assim quando a conversão de uma definição de uma função ocorre integralmente sem erros, a definição *pointwise* é substituída pela equivalente em *point-free*. Se se verificar algum erro na conversão dessa função os resultados parciais de conversão são ignorados e a definição *pointwise* é mantida.

Relativamente às definições de tipos de dados indutivos não há conversão das mesmas mas apenas geração de uma instância tomando estas definições como informação. Deste modo as definições são sempre mantidas e no caso de o processo de geração da instância FunctorOf para esse tipo ocorrer livre de qualquer erro, é acrescentada essa instância.

Sucintamente, sempre que ocorre um erro o código *pointwise* original é mantido ou a instância não é criada.

Na secção 5.1 aquando da explicação do algoritmo **PF** surge o caso de aplicação de uma função básica a uma expressão. Estas funções básicas têm aridade um, tal como as expressões *point-free*, que não são mais do que expressões funcionais de aridade um escritas no estilo *point-free*.

Assim é possível generalizar a regra 3 de **PF** para

$$PF\ (f\ exp) = let\ (g, x) = PF\ exp\ in\ (f\ .\ g,\ x)$$

A definição do algoritmo é a mesma só que agora f pode ser, além de uma função básica, uma qualquer expressão *point-free*. Tal como anteriormente *exp* é uma expressão **Haskell**, g é uma expressão *point-free* e x é outra expressão **Haskell**.

A implementação actual abrange esta generalização.

A implementação do algoritmo **PF** levantou um pequeno problema; as expressões *point-free* que dela resultavam eram muito longas e eram-no desnecessariamente. Assim optou-se por aplicar uma função simplificadora de expressões *point-free* ao resultado do algoritmo **PF**.

As regras de simplificação utilizadas são:

- $g\ .\ id = g$
- $id\ .\ g = g$
- $fst\ /\backslash\ snd = id$
- $id \times id = id$
- $inl\ \backslash/\ inr = id$
- $id + id = id$

Estas regras são aplicadas numa única travessia e se se pensar na árvore da expressão estas regras são aplicadas pela ordem acima e de forma *bottom up*.

Relativamente a 5.3, a implementação da junção dos vários ramos também levantou problemas.

Como visto anteriormente o processo pode-se aplicar repetidamente ao resultado da junção anterior tal como sucede quando há composições sucessivas de construtores. Optou-se então por implementar a junção de ramos pelo método do ponto fixo; vai-se aplicando o processo de junção, sucessivamente ao passo de junção anterior, até que um passo de junção produza o mesmo resultado que o passo anterior, sendo retornado este resultado como o resultado do método.

Tendo-se verificado que em alguns casos este método conduzia à não terminação modificou-se a implementação para o método do ponto fixo com limite de iterações; isto significa que se após um número previamente fixado de iterações o método não convergir (para o tal ponto fixo) o processo termina com o resultado desse momento.

Outra situação que na implementação careceu de atenção foi a associatividade da composição. Na explanação escreve-se $f\ .\ g\ .\ h$ com f, g e h funções para representar tanto $(f\ .\ g)\ .\ h$ como $f\ .\ (g\ .\ h)$. Contudo, na prática, a implementação distingue-as e surgem situações em que é necessário passar uma expressão com associatividade à esquerda como $(f\ .\ g)\ .\ h$ para a equivalente com associatividade à direita, $f\ .\ (g\ .\ h)$.

Em 5.3 no ponto 2 refere-se que, no caso de múltiplas composições de construtores, a sua eliminação ocorre da direita para a esquerda, tornando-se necessário associar todas as composições à esquerda para a aplicação das regras descritas.

Também relativamente à junção de ramos no caso de produtos de construtores surgiram dificuldades mas também simplificações.

A simplificação concerne as funções em que para todos os padrões de uma das componentes o resultado é o mesmo, isto é, funções em que só se pretende exaustão de padrões numa das componentes. É o caso de

```
plus :: (Nat,Nat) -> Nat
plus (Zero a, Zero b) = Zero b
plus (Zero a, Succ b) = Succ b
plus (Succ n, Zero b) = Succ (plus (n,Zero b) )
plus (Succ n, Succ b) = Succ (plus (n,Succ b) )
```

que pode, e deve, ser abreviado para

```
plus :: (Nat,Nat) -> Nat
plus (Zero a, c) = c
plus (Succ n, c) = Succ (plus (n,c) )
```

e cuja definição *point-free* é

```
plus :: (Nat, Nat) -> Nat
plus = snd \/ (Succ . plus) . distl . out >< id
```

Pensando na generalização surgem dois casos: exaustão de padrões apenas na primeira componente ou apenas na segunda. Tomando as definições após o passo de eliminação de variáveis vem:

1. Exaustão de padrões apenas na primeira componente

$$\begin{aligned} f \;.\; ConstA_1 \times id &= pfe_1 \\ &\vdots \\ f \;.\; ConstA_m \times id &= pfe_m \end{aligned}$$

com $ConstA_i$ com $i \in \{1, \ldots, m\}$ todos os construtores de um tipo de dados indutivo e pfe_j expressões *point-free* para $j \in \{1, \ldots, m\}$.
A transformação é

$$f = pfe_1 \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_m \;.\; MdistL \;.\; out \times id$$

2. Exaustão de padrões apenas na segunda componente

$$\begin{aligned} f \;.\; id \times constB_1 &= pfe_1 \\ &\vdots \\ f \;.\; id \times constB_n &= pfe_n \end{aligned}$$

com $ConstB_i$ com $i \in \{1, \ldots, n\}$ todos os construtores de um tipo de dados indutivo e pfe_j expressões *point-free* para $j \in \{1, \ldots, n\}$.
A transformação neste caso é

$$f = pfe_1 \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_m \;.\; NdistR \;.\; id \times out$$

Outra dificuldade encontrada concerne também a associatividade; na implementação os produtos e co-produtos de tipos assim como os *splits*, *eithers*, produtos, somas e composições de expressões *point-free* n-ários são as versões binárias associadas à direita.
Assim

$$
\begin{aligned}
f \;= {} & (pfe_{1,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{1,n}) \mathbin{\backslash/} \ldots \mathbin{\backslash/} (pfe_{m,1} \mathbin{\backslash/} \ldots \mathbin{\backslash/} pfe_{m,n}) \\
& \cdot \underbrace{NdistR + \ldots + NdistR}_{m \text{ parcelas}} \\
& \cdot MdistL \\
& \cdot out_A \times out_B
\end{aligned}
$$

apresentado em 5.3 no ponto 3, passa, e omitindo ainda os parêntesis da associatividade da composição, para

$$
\begin{aligned}
f \;= {} & \big(pfe_{1,1} \quad \mathbin{\backslash/} (\ldots (pfe_{1,n-1} \quad \mathbin{\backslash/} pfe_{1,n}) \quad \ldots)\big) \\
& \mathbin{\backslash/} \Big(\ldots \\
& \mathbin{\backslash/} \\
& \Big(\big(pfe_{m-1,1} \quad \mathbin{\backslash/} (\ldots (pfe_{m-1,n-1} \mathbin{\backslash/} pfe_{m-1,n}) \ldots)\big) \\
& \mathbin{\backslash/} \\
& \big(pfe_{m,1} \quad \mathbin{\backslash/} (\ldots (pfe_{m,n-1} \quad \mathbin{\backslash/} pfe_{m,n}) \quad \ldots)\big) \Big) \ldots \Big) \\
& \cdot \underbrace{NdistR + \big(\ldots + (NdistR + NdistR) \ldots\big)}_{m \text{ parcelas}} \\
& \cdot MdistL \\
& \cdot out_A \times out_B
\end{aligned}
$$

Surge ainda outra dificuldade; *MdistL* e *NdistR* não são funções mas sim famílias de funções. Repare-se que os tipos são

$$MdistL \;:\; T_1 + \big(\ldots + (T_{m-1} + T_m) \ldots\big) \times T' \longrightarrow T_1 \times T' + \big(\ldots + (T_{m-1} \times T' + T_m \times T') \ldots\big)$$

$$NdistR \;:\; T' \times T_1 + \big(\ldots + (T_{m-1} + T_m) \ldots\big) \longrightarrow T' \times T_1 + \big(\ldots + (T' \times T_{m-1} + T' \times T_m) \ldots\big)$$

Assim para cada $m \geq 2$ e cada $n \geq 2$, respectivamente, *MdistL* e *NdistR* materializam uma função cuja definição *point-free* pode ser obtida por recorrência.

MdistL define-se então como:

- caso $M = 2$ então *MdistL* é *distl*
- caso $M \geq 2$ então *MdistL* é $id \;+\; (M{-}1)distL \quad . \quad distl$

em que *distl* é a função definida em 2.2 e $(M{-}1)distL$ é *MdistL*com M a ser $M - 1$.

Analogamente define-se *NdistR* como

- caso $N = 2$ então *NdistR* é *distr*
- caso $N \geq 2$ então *NdistR* é $id \;+\; (N{-}1)distR \quad . \quad distr$

com *distr* e $(N{-}1)distR$ análogas a *distl* e $(M{-}1)distL$ respectivamente.

A implementação de produtos e co-produtos de tipos e de *splits*, *eithers*, produtos, somas e composições de expressões *point-free* n-ários como as versões binárias associadas à direita também obrigam a uma reformulação da geração da instância FunctorOf.

Considerando ainda que na biblioteca Pointless a representação das definições dos functores usa os símbolos :+: para os co-produtos e :∗: para os produtos, a secção 5.4 sofre as seguintes alterações:

- $X = U_1 + \ldots + U_m$ passa a $X = U_1 :+: \left(\ldots :+: (U_{m-1} :+: U_m)\ldots\right)$
- as regras para a definição de U_i passam a
  1. Se n_i é zero então U_i é `Const ()`
  2. Se n_i é maior do que zero então U_i é $V_{i,1} :*: \left(\ldots :*: (V_{i,n_i-1} :*: V_{i,n_i})\ldots\right)$
     Em que cada $V_{i,j}$ deriva de $T_{i,j}$ para todo $i \in \{1,\ldots,m\}$ e $j \in \{1,\ldots,n_i\}$ de acordo com as regras que se seguem:
     (a) Se $T_{i,j}$ é $IndType\ p_1 \ldots p_l$ então $V_{i,j}$ não é senão `Id`; representa a recursividade do tipo.
     (b) Se $T_{i,j}$ é p_k para algum $k \in \{1,\ldots,l\}$ então $V_{i,j}$ é `Const` p_k
     (c) Se $T_{i,j}$ é um construtor de tipos de aridade zero (por exemplo, `Int`) então $V_{i,j}$ é `Const` $T_{i,j}$.
     (d) Se $T_{i,j}$ é um produto de tipos de aridade q então $V_{i,j}$ é um produto de q "factores", em que a cada um deles são aplicadas as regras 2a a 2d, isto é, $V_{i,j}$ é $W_{i,j,1} :*: \left(\ldots :*: (W_{i,j,q-1} :*: W_{i,j,q})\ldots\right)$ em que, tomando $T_{i,j}$ como $Y_{i,j,1} \times \ldots \times Y_{i,j,q}$, cada $W_{i,j,h}$ deriva de $Y_{i,j,h}$ para todo o $h \in \{1,\ldots,q\}$ usando as regras de derivação 2a a 2d.
- Obviamente, a alteração dos tipos de dados implica alterações nas funções *in* e *out*. Assim inj_i define-se como:
  - para $i = 1$, $\quad inj_i$ é Inl
  - para $i \in \{2,\ldots,n-1\}$, $\quad inj_i$ é $\underbrace{Inr\ .\ \cdots\ .\ Inr}_{i-1 \text{ parcelas}}\ .\ Inl$
  - para $i = n$, $\quad inj_i$ é $\underbrace{Inr\ .\ \cdots\ .\ Inr}_{n-1 \text{ parcelas}}$

  As regras de geração de b_i passam a
  1. Se n_i é zero (o construtor $Const_i$ não tem argumentos) então b_i é `Const ()`.
  2. se n_i é maior do que zero então b_i será um tuplo de n_i componentes (este caso também considera um tuplo de uma única componente), seja
     $c_{i,1} :*: \left(\ldots :*: (c_{i,n_i-1} :*: c_{i,n_i})\ldots\right)$ sendo que, para cada $j \in \{1,\ldots,n_i\}$, $c_{i,j}$ é:
     (a) Se $arg_{i,j}$ é uma variável simples então duas situações podem acontecer:
     - se a variável argumento $arg_{i,j}$ é do tipo $IndType\ p_1 \ldots p_l$ então $c_{i,j}$ é `Id` $arg_{i,j}$
     - se a variável argumento $arg_{i,j}$ não é do tipo $IndType\ p_1 \ldots p_l$ então $c_{i,j}$ é `Const` $arg_{i,j}$

     (b) Se $arg_{i,j}$ é um tuplo, seja $arg_{i,j} = (d_{i,j,1},\ldots,d_{i,j,q})$ então $c_{i,j}$ é $e_{i,j,1} :*: \left(\ldots :*: (e_{i,j,q-1} :*: e_{i,j,q})\ldots\right)$ sendo que para cada $h \in \{1,\ldots,q\}$ a $d_{i,j,h}$ aplica-se a regra 2a ou a 2b para obter $e_{i,j,h}$

6.1 Exemplos

Termina-se este capítulo apresentando alguns exemplos de conversão. Estes exemplos são obtidos automaticamente pelo uso da ferramenta de conversão que implementa as ideias até agora explanadas.

Eis então os exemplos:

- `data Nat = Zero () | Succ Nat deriving (Show,Eq)`

```
  instance FunctorOf (Const () :+: Id) Nat where
  inn' (Inl (Const ())) = Zero ()
  inn' (Inr (Id v1))    = Succ v1
  out' (Zero ()) = Inl (Const ())
  out' (Succ v1) = Inr (Id v1)

- data List a = Nil () | Cons (a, (List a) ) deriving (Show,Eq)

  instance FunctorOf ( Const () :+:  Const a :*: Id ) (List a)
   where
  inn' (Inl (Const ()))             = Nil ()
  inn' (Inr ((Const v1 :*: Id v2))) = Cons (v1, v2)
  out' (Nil ())        = Inl (Const ())
  out' (Cons (v1, v2)) = Inr ((Const v1 :*: Id v2))

- fact :: Nat -> Nat
  fact (Zero a) = Succ (Zero a)
  fact (Succ n) = mult (Succ n, fact n)

  fact :: Nat -> Nat
  fact
    = (((Succ . Zero) \/ ((mult . (Succ >< fact)) . (id /\ id))) . out)

- fib :: Nat -> Nat
  fib (Zero a) = Succ (Zero a)
  fib (Succ (Zero a)) = Succ (Zero a)
  fib (Succ (Succ a)) = plus (fib (Succ a), fib a)

  fib :: Nat -> Nat
  fib
    = (((Succ . Zero) \/
  (((Succ . Zero) \/ ((plus . ((fib . Succ) >< fib)) . (id /\ id))) .
     out))
         . out)

- nat2Int :: Nat -> Int
  nat2Int (Zero _) = 0
  nat2Int (Succ n) = 1 + nat2Int n

  nat2Int :: Nat -> Int
  nat2Int
    = ((((!) 0) \/ ((uncurry (+) . (((!) 1) >< nat2Int)) . (id /\ id)))
         . out)

- zip1 :: (List a, List b) -> List (a,b)
  zip1 (Nil a, Nil _) = Nil a
  zip1 (Nil a, Cons _) = Nil a
  zip1 (Cons _ , Nil b) = Nil b
  zip1 (Cons (x,xs), Cons (y,ys)) = Cons ((x,y), zip1 (xs,ys))

  zip1 :: (List a, List b) -> List (a, b)
  zip1
    = ((((Nil . fst) \/ (Nil . fst)) \/
  ((Nil . snd) \/
     ((Cons . (id >< zip1)) .
        (((fst . fst) /\ (fst . snd)) /\ ((snd . fst) /\ (snd . snd))))))
         . ((distr -|- distr) . (distl . (out >< out))))
```

7 Ideias em Evolução

Como referido por diversas vezes, a ferramenta no estado actual não trata algumas situações. Para algumas dessas situações foram surgindo ideias que apesar de não terem sido incorporadas na ferramenta, se pretende registar aqui.

Algumas ideias não foram integradas por dificuldades de tempo e de implementação, enquanto que outras carecem ainda de generalização e/ou fundamentação teórica adequada, pelo que deverão ser tidas como pontos de partida para soluções e não como soluções propriamente ditas.

No que concerne as funções *curried* é possível afirmar as seguintes igualdades

```
f1 a       = f1 a
f2 a b     = uncurry f2 (a,b)
f3 a b c   = uncurry (uncurry f3) ((a,b),c)
f4 a b c d = (uncurry . uncurry . uncurry $ f4) (((a,b),c),d)
```

a generalização parece ser

$$f\ a_1\ \ldots\ a_n\ =\ (\underbrace{uncurry\ .\ \cdots\ .\ uncurry}_{n-1\text{ vezes}}\ \$\ f)\ (\ldots(a_1, a_2)\ldots,\ a_n)$$

para $n \in \mathbb{N}$ mas finito. No caso de n ser um, a composição de zero *uncurry*'s é o elemento neutro da composição - a função identidade.

Contudo surgem problemas pois a expressão do lado direito da igualdade não é representável nas expressões *point-free* actuais.

Um exemplo engraçado com uma função *curried* e de ordem superior e respectiva manipulação é

$$\begin{aligned} f\ a\ (g\ b) &= (uncurry\ f)\ (a,\ g\ b) \\ &= uncurry\ f\ .\ (id \times g)\ \$\ (a, b) \end{aligned}$$

Mas como mecanizar o processo?

Também aqui a última expressão à direita não é representável na notação *point-free* vigente.

Outro resultado que fará possivelmente falta para suportar funções *curried* é

$$uncurry\ f\ =\ g \quad \Longrightarrow \quad f\ =\ curry\ g$$

Outra situação vulgar em **Haskell** é a junção de casos cuja definição é igual, não fazendo *pattern-matching* exaustivo, por exemplo

```
ordering2Bool EQ = True
ordering2Bool _  = False
```

Pretende-se, futuramente, equipar a ferramenta com uma pré manipulação para

```
ordering2Bool EQ = True
ordering2Bool LT = False
ordering2Bool GT = False
```

Afim de que, após esta reescrita, seja possível utilizar a ferramenta, tal como está, para obter a definição *point-free* equivalente

```
ordering2Bool  = (True!) \/ (False!) \/ (False!) . out
```

Também para expressões `if ... then ... else ...`se pondera uma solução; dada a expressão

$$\texttt{if}\ (cond\ x)\ \texttt{then}\ f1\ x\ \texttt{else}\ f2\ x$$

Com *cond* uma função de um argumento que retorna um booleano, x uma variável ou tuplos, possivelmente encadeados, de variáveis e $f1$ e $f2$ funções de um argumento com o mesmo tipo para o resultado.

A conversão será possivelmente

$$\big[f1,\ f2\big]\ .\ (cond\ ?)\ \$\ x$$

Quanto a padrões suponha-se o seguinte caso

```
f x | pred_1 x  = rhs_1 x
      ⋮
    | pred_n x  = rhs_n x
    | otherwise = rhsdefault x
```

com $n \in \mathbb{N}$, x uma variável ou tuplos, possivelmente encadeados, de variáveis, $\texttt{pred}_i$ para $i \in \{1, \ldots, n\}$ é uma função booleana de aridade um e para cada $i \in \{1, \ldots, n\}$ $\texttt{rhs}_i$ é uma função de aridade um, sendo que todos os $\texttt{rhs}_i$ têm o mesmo tipo para o resultado e que é ainda o mesmo de `rhsdefault`.

Pensando os padrões como `if`'s encadeados, vem

```
f x = if pred_1 x then rhs_1 x
                  else ...
                       if pred_n x then rhs_n x
                                   else rhsdefault x
```

Utilizando a regra de conversão proposta para os `if ... then ... else ...`, vem

$$f\ =\ [rhs_1, \ldots, [rhs_n,\ rhsdefault]\ .\ (pred_n\ ?) \ldots]\ .\ (pred_1\ ?)$$

As limitações para as regras de conversão tanto para os `if ... then ... else ...` como para os padrões são acentuadas; as variáveis têm de ser sempre a mesma e as funções de aridade um referidas têm de ser já expressões *point-free*.

Atente-se agora na definição *point-free* que se obtém actualmente para a função básica que calcula o comprimento de uma lista

$$len\ =\ Zero \backslash/ (Succ\ .\ len\ .\ snd)\ .\ out$$

É facto bem conhecido que tal função básica é um catamorfismo de listas. Manualmente faz-se a manipulação que se segue

$$\begin{aligned}
len = &\ Zero \backslash/ (Succ\ .\ len\ .\ snd)\ .\ out \\
= &\ Zero \backslash/ (Succ\ .\ snd\ .\ id \times len)\ .\ out \\
= &\ (Zero\ .\ id) \backslash/ \Big((Succ\ .\ snd)\ .\ id \times len\Big)\ .\ out \\
= &\ Zero \backslash/ (Succ\ .\ snd)\ .\ id + id \times len\ .\ out \\
= &\ Zero \backslash/ (Succ\ .\ snd)\ .\ rec\ .\ out \\
= &\ cata\Big(Zero \backslash/ (Succ\ .\ snd)\Big)
\end{aligned}$$

com $rec = F\ len$ para F o functor de listas

A possibilidade de automatizar este processo fica em aberto para trabalhos futuros.

8 Conclusões

Em várias das referências bibliográficas é defendido que o estilo *point-free* é o mais adequado para raciocinar, transformar ou provar propriedades de/sobre programas.

Em todos estes trabalhos, o intuito de mecanizar tais tarefas apresenta uma lacuna comum; o código *pointwise* é convertido em código *point-free*, em forma *ad-hoc* sem que sejam apresentadas regras, ou quais as justificações dessa conversão.

Este trabalho vem suprimir essa lacuna e apresenta uma primeira teoria para a conversão de código *pointwise* em código *point-free*; diz-se uma primeira teoria pois esta não abrange toda a extensão do código *pointwise*, nomeadamente:

- funções *curried*
- abstracções λ
- funções de ordem superior
- instruções de controlo como guardas e `if ... then ... else ...` (embora uma sugestão seja já apresentada para estas)
- instruções de "referência" como `let ... in ...` e `where ...` (estas instruções necessitarão de um suporte baseado em teoria de substituição)

não são ainda tomadas em consideração na teoria apresentada.

Além da conversão apresenta-se alguma teoria que, articulada com a teoria subjacente à biblioteca Pointless, permite definir automaticamente uma vasta gama de padrões recursivos para a maioria dos tipos de dados indutivos, tendo em consideração apenas a definição destes.

Ainda neste trabalho, complementou-se a teoria com uma ferramenta que implementa uma parte substancial dessa teoria. Assim o leitor é convidado a experimentar a ferramenta, cuja utilização é descrita no apêndice A.

No capítulo *Ideias Em Evolução*(7) fizeram-se já algumas sugestões para situações que não foram tratadas. Assim como trabalho futuro sugere-se :

- a extensão da teoria às restantes "construções" *pointwise*, acima enumerados,
- o aniquilamento das limitações de implementação,
- e ainda a investigação de teoria e respectiva implementação para se converter definições *point-free* que são padrões recursivos, mas em que estes não explicitamente declarados, nas respectivas definições com padrões recursivos explícitos.

Agora falando pessoalmente, eu, o autor (não autor exclusivo pois este trabalho teve contributos significativos do meu orientador e de Alcino Cunha), faço uma avaliação positiva deste projecto; com ele adquiri novos conhecimentos sobre a linguagem **Haskell**, sobre o estilo de programação *point-free* e os seus combinadores, sobre os tipos de dados indutivos e as suas propriedades e sobre os padrões de recursividade, que estão associados aos tipos indutivos.

Mas, principalmente, este projecto "*abriu-me as portas*" para um novo mundo, o da investigação científica, facto que foi impulsionado pela minha "inserção" no **PURe café**.

Referências

BdM97. Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

CP04. Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. Technical Report DI-PURe-04:02:03, Departamento de Informática, Universidade do Minho, February 2004. Available from `http://www.di.uminho.pt/pure`.

Cun. Manuel Alcino Cunha. Point-free programming with hylomorphisms. Unpublished note.

Gib02. Jeremy Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer-Verlag, 2002.

MFP91. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

Oli99a. José Nuno Oliveira. An introduction to pointfree programming, 1999. Draft document.

Oli99b. José Nuno Oliveira. Recursion in the pointfree style, 1999. Draft document.

Ven00. Varmo Vene. Categorical programming with inductive and coinductive types, August 2000.

A Manual do Utilizador

Actualmente a ferramenta consiste num ficheiro executável. A partir de um *shell* (por exemplo *bash*) é possível utilizar a ferramenta de duas formas distintas:

1. invocando a ferramenta passando como argumento, na *bash*, o nome do ficheiro **Haskell** com o código *pointwise* a converter. Neste caso o resultado, outro ficheiro de código **Haskell** total ou parcialmente *point-free*, é colocado num ficheiro com o nome do ficheiro de entrada mas prefixado com `GER`.
   Por exemplo, supondo que o código **Haskell** *pointwise* está no ficheiro `Example.hs` e fazendo a invocação seguinte na *bash*

   ```
   pointfree Example.hs
   ```

   surge um ficheiro `GERExample.hs` com o resultado.
2. invocando a ferramenta como filtro, utilizando os redireccionamentos de *bash*. Esta forma permite criar comandos de *bash* mais elaborados com execução de outras operações, nomeadamente articulação com ferramentas geradoras de código ou de extracção de código de ficheiros em *literate Haskell*.
   Para o exemplo anterior, far-se-ia a invocação

   ```
   pointfree < Example.hs > GERExample.hs
   ```

A invocação da ferramenta em si não levanta grandes questões mas para que os resultados obtidos sejam o mais interessantes e completos possível é necessário tomar alguns cuidados com o código *pointwise*:

- Sempre que numa definição de uma função surjam expressões que envolvam
  - `if ... then ... else ...`
  - `case ... of ...`
  - `let ... in ...`
  - `where ...`
  - tuplos que não são pares ordenados (os tuplos podem sempre ser convertidos em pares encadeados)
  - guardas
  - ou ainda notação monádica

  a definição *pointwise* do *input* é imediatamente ignorada pelo processo de conversão.
- Actualmente só a informação presente no módulo corrente é passível de ser tratada, pelo que definições de funções sobre tipos de dados indutivos definidos noutros módulos (mesmo que o `Prelude`) não são convertidas para *point-free*, como seria expectável.
- Os construtores dos tipos de dados indutivos e as funções terão de ter aridade *um* para que seja possível a conversão. A única excepção são os operadores infixos, com aridade dois, que são suportados.
  Assim as funções e os construtores *curried* terão de ser alterados pelo programador para as respectivas versões *uncurried* em que os argumentos estão em pares ordenados encadeados.
  Por exemplo

  ```
  f a b c = a + b + c
  ```

  não será convertida, mas

  ```
  g (a, (b, c)) = a + b + c
  ```

já é convertida para *point-free*.

– O programador é responsável por garantir que a definição é exaustiva nos padrões e que estes são disjuntos, isto é, para uma função com um argumento de um tipo de dados indutivo têm de existir tantos ramos quantos os construtores do tipo indutivo (um ramo por cada construtor, com os construtores explicitamente presentes nos padrões).
Por exemplo

```
data Nat = Zero () | Succ Nat

f (Zero a) = 1
```

não é convertido, tal como não o é se se acrescentar, no fim do código anterior, a linha `f _ = 2`. Mas

```
data Nat = Zero () | Succ Nat

f (Zero a) = 1
f (Succ n) = 2
```

já é convertido.
Também o caso

```
data Tipo = Const1 () | Const2 () | Const3 ()

f :: Tipo -> Int
f (Const1 _) = 1
f _          = 2
```

não é convertido, mas se o programador alterar para

```
data Tipo = Const1 () | Const2 () | Const3 ()

f :: Tipo -> Int
f (Const1 _) = 1
f (Const2 _) = 2
f (Const3 _) = 2
```

já é convertido para *point-free*.

Contudo, algumas flexibilidades foram já introduzidas:

– É possível alterar a ordem dos ramos numa definição sem que isso altere a conversão. Por exemplo

```
fib (Zero a) = Succ (Zero a)
fib (Succ (Zero a)) = Succ (Zero a)
fib (Succ (Succ a)) = plus (fib (Succ a), fib a)
```

e

```
fib (Succ (Succ a)) = plus (fib (Succ a), fib a)
fib (Zero a) = Succ (Zero a)
fib (Succ (Zero a)) = Succ (Zero a)
```

resultam na mesma definição em *point-free*.

– No caso de pares de construtores no argumento de uma função é possível trocar a ordem dos ramos e não efectuar exaustão de padrões numa das componentes.
Por exemplo

```
plus (Zero a, c) = c
plus (Succ n, c) = Succ (plus (n,c) )
```

abrevia

```
plus (Zero a, Zero b) = Zero b
plus (Zero a, Succ b) = Succ b
plus (Succ n, Zero b) = Succ (plus (n,Zero b) )
plus (Succ n, Succ b) = Succ (plus (n,Succ b) )
```

ou ainda, trocando a ordem dos ramos da definição anterior

```
plus (Succ n, Zero b) = Succ (plus (n,Zero b) )
plus (Zero a, Succ b) = Succ b
plus (Zero a, Zero b) = Zero b
plus (Succ n, Succ b) = Succ (plus (n,Succ b) )
```

As duas últimas definições conduzem à mesma definição em *point-free* e esta é equivalente (mas diferente) da definição *point-free* gerada pela primeira definição *pointwise* da função `plus`.

Qualquer dúvida, sugestão ou anomalia (*bug*), por favor envie *email* para pointfrezador@sapo.pt.

B Tipo de Dados PFExp

```
module PFExpType (PFExp (Func
           , Comp
           , Split
           , Prod
           , Either
           , Sum
           , Fst
           , Snd
           , InL
           , InR
           , Const
           , Id
           , In
           , Out
           )
     -- instance Eq, Show
     , cataPFExp
                ) where

-- | Data type to represent point-free expressions
data PFExp = Func String  -- ^ function
     | Comp   PFExp PFExp -- ^ composition
     | Split  PFExp PFExp -- ^ split
```

```
     | Prod   PFExp PFExp -- ^ product
     | Either PFExp PFExp -- ^ either
     | Sum    PFExp PFExp -- ^ sum
     -- split projections
     | Fst    -- ^ first projection of a pair
     | Snd    -- ^ second projection of a pair
     -- either injections
     | InL    -- ^ left injector of co-product
     | InR    -- ^ right injector of co-product
     | Const String   -- ^ the const function like in Haskell
     | Id     -- ^ the identity function
     | In     -- ^ the in function
     | Out    -- ^ the out function
     deriving Eq



auxP n e1 e2 simb = parA n
      . showsPrec 1 e1
      . showString simb
      . showsPrec 1 e2
      . parF n
      where parA 0 = showString ""
            parA 1 = showChar '('
            parF 0 = showString ""
            parF 1 = showChar ')'

instance Show PFExp where
  showsPrec _ (Func str)     = showString str
  showsPrec n (Comp e1 e2)   = showsPrec 1 e1 . showString " . " . showsPrec 1 e2
  showsPrec n (Split e1 e2)  = auxP n e1 e2 " /\\ "
  showsPrec n (Prod e1 e2)   = auxP n e1 e2 " >< "
  showsPrec n (Either e1 e2) = auxP n e1 e2 " \\/ "
  showsPrec n (Sum e1 e2)    = auxP n e1 e2 " -|- "
  showsPrec _ Fst            = showString "fst"
  showsPrec _ Snd            = showString "snd"
  showsPrec _ InL            = showString "inl"
  showsPrec _ InR            = showString "inr"
  showsPrec _ (Const str)    = showString "const " . showString str
  showsPrec _ Id             = showString "id"
  showsPrec _ In             = showString "inn"
  showsPrec _ Out            = showString "out"


-- | the cataphorphism function for the 'PFExp' datatype.
cataPFExp :: (String -> a -- ^ for functions
     , a -> a -> a  -- ^ for composition
     , a -> a -> a  -- ^ for split
     , a -> a -> a  -- ^ for product
     , a -> a -> a  -- ^ for either
     , a -> a -> a  -- ^ for sum
```

```
     , a      -- ^ for fst
     , a      -- ^ for snd
     , a      -- ^ for i1
     , a      -- ^ for i2
     , String -> a  -- ^ for constant function
     , a      -- ^ for id function
     , a      -- ^ for in function
     , a)     -- ^ for out function
     -> PFExp     -- ^ the input pointfree expression
     -> a     -- ^ the final result
cataPFExp (ff, fc, fs, fp, fe, fm, p1, p2, i1, i2, c, i, fi, fo) = cata
  where cata (Func   string)      = ff string
        cata (Comp   pFExp1 pFExp2) = fc (cata pFExp1) (cata pFExp2)
        cata (Split  pFExp1 pFExp2) = fs (cata pFExp1) (cata pFExp2)
        cata (Prod   pFExp1 pFExp2) = fp (cata pFExp1) (cata pFExp2)
        cata (Either pFExp1 pFExp2) = fe (cata pFExp1) (cata pFExp2)
        cata (Sum    pFExp1 pFExp2) = fm (cata pFExp1) (cata pFExp2)
        cata Fst        = p1
        cata Snd        = p2
        cata InL        = i1
        cata InR        = i2
        cata (Const string)       = c string
        cata Id         = i
        cata In         = fi
        cata Out        = fo
```