

JN O Universidade do

Minho LIVEIRA

**PROJETO PROGRAMA
POR
CÁLCULO**

(PROJECTO de livro de texto em preparação)

Última atualização: fevereiro 2018

Conteúdo

Preâmbulo	1
1. Introdução	3
I Cálculo com funções	5
2 Uma Introdução à Programação pointfree	7
2.1 Apresentando funções e tipos.....	8
2.2 aplicação funcional.....	9
2,3 igualdade funcional e composição.....	10
2.4 funções identidade.....	12
2,5 funções constante.....	13
2,6 Monics e épicos.....	15
2.7 Isos.....	16
2.8 Colando funções que não compõem - produtos.....	18
2.9 funções colagem que não compõem - co-produtos.....	24
2.10 produtos e co-produtos de mistura.....	28
2.11 tipos de dados elementares.....	30
2,12 propriedades naturais.....	33
2,13 propriedades universais.....	35
2.14 Guardas e McCarthy condicional de.....	38
2.15 funções colagem que não compõem - exponenciais.....	42
2,16 produtos e subprodutos finitária.....	50
2,17 tipos de dados iniciais e terminais.....	52
2,18 somas e produtos em HASKELL..... 53	
2,19 exercícios.....	57
2.20 notas bibliografia.....	60

3 recursão no estilo pointfree	63
3.1 Motivação.....	63
3.2 A partir de números naturais para F1 sequências nite.....	69
3.3 Introdução de tipos de dados indutivos.....	75
3.4 Observando um tipo de dados indutivo.....	81
3.5 síntese de um tipo de dados indutivo.....	84
3.6 Introdução (lista) catas, anas e hylos.....	86
3.7 tipos indutivos mais geral.....	92
3.8 Functores.....	93
3.9 functors polinomiais.....	95
3.10 tipos indutivos polinomiais.....	97
3.11 F- ágebras e F- homomorphisms.....	98
3.12 F- catamorphisms.....	99
3.13 parametrização e tipo functors.....	102
3.14 Um catálogo de tipos indutivos polinomiais padrão.....	107
3.15 Functores e tipo functors em HASKELL.....	110
3.16 A lei mútua-recursão.....	111
3.17 “Banana-split”: um corolário da lei mútua-recursão.....	120
3.18 indutivo isomorfismo tipo de dados.....	123
3.19 nota bibliografia.....	123
 4 Por Monads Matéria	 125
4.1 funções parciais.....	125
4.2 Colocação funções parciais juntos.....	126
4.3 Listas.....	129
4.4 Monads.....	130
4.4.1 Propriedades envolvendo composição (Kleisli).....	132
4.5 aplicação Monádica (ligao).....	133
4.6 Sequenciamento eo Faz- notação.....	134
4.7 Geradores e comprehensões.....	134
4.8 Monads em HASKELL.....	136
4.8.1 Monádica de I / O.....	138
4.9 O mônade estado.....	140
4.10 'Monadi fi cação' de código Haskell fácil.....	147
4.11 Onde monads vem?.....	152
4.12 notas bibliografia.....	155

II Calculando com Relations	157
5 Especificar programas funcionais	159
6 Quando tudo se torna uma relação	161
7 Teoremas de graça: uma abordagem calculacional	163
8 Design by Contract - calculationally	165
9 Programas como relacionais Hylomorphisms	167
10 tipos de dados Quasi-indutivos	169
11 calculacional Programa de refinamento	171
12 pensamento relacional	173
Calculando com III Matrizes	175
13 Para uma Álgebra Linear de Programação	177
A Apêndice	179
biblioteca de suporte A.1 Haskell.	179
biblioteca de suporte A.2 liga.	184

Lista de Exercícios

Exercício 2.1.....	14	Exercício 2.2.....	
.....14		Exercício 2.3.....	16
.....24		Exercício 2.5.....	27
Exercício 2.6.....		Exercício 2.7.....	
.....30		Exercício 2.8.....	30
.....30		Exercício 2.10.....	
Exercício 2.11.....		Exercício 2.12.....	
.....33		Exercício 2.13.....	34
.....34		Exercício 2.15.....	
Exercício 2.16.....		Exercício 2.17.....	
.....37		Exercício 2.18.....	37
.....37		Exercício 2.20.....	
Exercício 2.21.....		Exercício 2.22.....	
.....41		Exercício 2.23.....	41
.....49		Exercício 2.25.....	
Exercício 2.26.....		Exercício 2.27.....	
.....49		Exercício 2.28.....	50
Exercício 2.26.....		Exercício 2.27.....	
.....49		Exercício 2.28.....	
.....49			50

Exercício 2.29.....	51	Exercício 2.30.....	
..... 53 Exercício 2.31.....	57	Exercício 2.32.....	
..... 57 Exercício 2.33.....	57		
Exercício 2.34.....	57	Exercício 2.35.....	
..... 58 Exercício 2.36.....	58	Exercício 2.37.....	
..... 59 Exercício 2.38.....	59		
Exercício 2.39.....	59	Exercício 2.40.....	
..... 59 Exercício 2.41.....	60	Exercício 2.42.....	
..... 60 Exercício 2.43.....	60		
Exercício 3.1.....	68	Exercício 3.2.....	
..... 68 Exercício 3.3.....	68	Exercício 3.4.....	
..... 68 Exercício 3.5.....	68		
Exercício 3.6.....	80	Exercício 3.7.....	
..... 91 Exercício 3.8.....	91	Exercício 3.9.....	
..... 91 Exercício 3.10.....	97		
Exercício 3.11.....	106	Exercício 3.12.....	
..... 107 Exercício 3.13.....	108	Exercício 3.14.....	
..... 108 Exercício 3.15.....			
109 Exercício 3.16.....	109	Exercício 3.17.....	
..... 110 Exercício 3.18.....	111	Exercício	
3.19.....	111	Exercício 3.20.....	
..... 116 Exercício 3.21.....	116	Exercício 3.22.....	
..... 117 Exercício 3.23.....	117		
..... 116 Exercício 3.22.....	117	Exercício 3.23.....	
..... 117	116	Exercício 3.22.....	
117 Exercício 3.23.....	117		

Exercício 3.24.....	118	Exercício 3.25.....	
..... 120 Exercício 3.26.....	122	Exercício 3.27.....	
..... 122 Exercício 3.28.....			
122 Exercício 4.1.....	128	Exercício 4.2.....	
..... 129 Exercício 4.3.....	129	Exercício 4.4.....	
..... 133 Exercício 4.5.....			
..... 135 Exercício 4.6.....	136	Exercício 4.7.....	
..... 137 Exercício 4.8.....	139	Exercício	
4.9.....	139	Exercício 4.10.....	
..... 151 Exercício 4.11.....	151	Exercício 4.12.....	
..... 155.....	151	Exercício 4.11.....	
..... 151 Exercício 4.12.....	155	
..... 151 Exercício 4.11.....	151		
Exercício 4.12.....	155		

Preâmbulo

Este livro, que surgiu a partir da experiência de pesquisa e ensino do autor, tem estado em preparação há muitos anos. Seu principal objetivo é chamar a atenção dos profissionais de software para uma abordagem calculacional ao desenho de artefatos de software que variam de algoritmos simples e funções para o fí cação específica e realização de sistemas de informação.

Coloque em outras palavras, o livro convida designers de software para elevar os padrões e adotar técnicas de desenvolvimento maduros encontrados em outras disciplinas de engenharia, que (como regra) estão enraizadas em uma base matemática som. *composicionalidade e parametricity* são fundamentais para toda a disciplina, garantindo a escalabilidade da secretaria da escola exerce a grandes problemas em um ambiente indústria.

É interessante notar que, enquanto cunhar a frase *Engenharia de software* na década de 1960, os nossos colegas da época já estavam a prometer tais padrões de alta qualidade. Em março de 1967, ACM Presidente Anthony Oettinger fez um discurso no qual ele disse: “(...) o fí cientí c, componente rigorosa da computação, é mais como matemática que é como física (...) Qualquer que seja, por um lado ele tem componentes da mais pura da matemática e, por outro lado dos mais sujos da engenharia [35].

Como uma disciplina, engenharia de software foi anunciado na conferência de Garmisch NATO em 1968, de cujo relatório [34] o seguinte trecho é citado:

No final de 1967, o grupo de estudo recomendou a realização de uma conferência de trabalho de Engenharia de Software. A frase 'engenharia de software' foi escolhida deliberadamente como sendo provocador, no implicando a necessidade de fabricação de software para basear-se nos tipos de fundamentos teóricos e disciplinas práticas, que são tradicionais nas sucursais estabelecidas de engenharia.

Provocante ou não, a necessidade de fundamentos teóricos de som tem sido claramente sob preocupação desde o início da disciplina - exatamente cinqüenta anos

atrás, no momento da escrita. No entanto, como “científica” fazer tais fundamentos vir a ser, agora que cinco décadas desde que decorrido?

Trinta anos mais tarde (1997), Richard Bird and Oege de Moore publicou um livro [6], em cujo carro prefácio Hoare escreve:

notação de programação pode ser expresso por “formulæ e equações (...) que partilham a elegância daqueles que estão subjacentes física e química ou qualquer outro ramo da ciência básica”.

As fórmulas e equações mencionadas nesta citação são aqueles de uma disciplina conhecida como a *Álgebra de programação*. Muitos outros têm contribuído para esse corpo de conhecimentos, nomeadamente Roland Backhouse e seus colegas Eindhoven e Nottingham, ver, por exemplo. [1, 2], Jeremy Gibbons e Ralf Hinze em Oxford ver, por exemplo [15], entre muitos outros. Infelizmente, as referências [1, 2] ainda são inéditos.

Quando o autor deste projecto de livro decidiu ensinar *Álgebra de Programação* aos estudantes 2º ano de graus theMinho em ciência da computação, de volta à 1998, ele encontrou livro [6] cult fi muito dif para os alunos a seguir, principalmente devido à sua categorial demasiado explícita (alegórica) fiavour. Então, ele decidiu começar a escrever slides e notas ajudando os alunos a ler o livro. Eventualmente, essas notas se tornou capítulos 2 a 4 da versão atual da monografia. O mesmo procedimento foi levado ao ensinar a abordagem relacional de [6] para estudantes do ano 5º 4º e (hoje em nível de mestrado).

Este projecto de livro é, em geral incompleta, a maioria dos capítulos estar ainda em *forma de slides*¹. Esses capítulos acabados meias são omitidos na atual print-out. Ao todo, a idéia é mostrar que a engenharia de software e, em particular, programação de computadores pode adotar a *Método C científica* como outros ramos da engenharia fazer.

Universidade do Minho, Braga, fevereiro 2018

Jos'e N. Oliveira

¹ Para os slides que eventualmente conduzirá à segunda parte deste livro ver relatório técnico [37]. A terceira parte irá abordar uma álgebra linear de programação destina-se ing razão- quantitativa sobre o software. Isto é ainda menos estável, mas uma série de documentos existem sobre o tema, a partir de [36].

parte I

Calcular com Funções

Capítulo 2

Uma Introdução à Programação pointfree

Todo mundo está familiarizado com o conceito de um *função* uma vez que a secretária da escola. A intuição funcional atravessa a matemática de ponta a ponta, porque ele tem uma sólida semântica enraizada em um sistema matemático bem conhecido - a classe de “todas” as séries e funções de set-teóricas.

A programação funcional significa literalmente “programação com funções”. As linguagens de programação, tais como LISP ou HASKELL nos permitem programar com funções. No entanto, a intuição funcional é muito mais abrangente do que a produção de código que é executado em um computador. Desde o trabalho pioneiro de John McCarthy - o inventor do LISP - no início de 1960, sabe-se que outros ramos da programação pode ser estruturado, ou expressa funcionalmente. A ideia de produzir programas de *Cálculo*, isto é, a de calcular programas de fi cientes ef fora de resumo, os fi cientes INEF tem uma longa tradição na programação funcional.

Este livro é estruturado em torno da ideia de que a programação funcional pode ser usado como base para o ensino de programação como um todo, a partir da função sucessor $n \rightarrow n + 1$ ao design grande sistema de informação.¹

Este capítulo fornece uma introdução leve à teoria da programação funcional. A ênfase principal é sobre *composicionalidade*, uma das principais vantagens de “pensar funcionalmente”, explicando como construir novas funções fora de outras funções que usam um conjunto mínimo de prede fi combinators funcionais NED. Isto leva a um estilo de programação que é *ponto de acesso gratuito* no sentido de que funcionar

¹ Esta ideia aborda programação em um sentido amplo, incluindo, por exemplo *reversível* e *programação quantum*, em que a programação funcional já desempenha papéis importantes [32, 31, 14].

8 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

descrições dispensar variáveis (também conhecidos como *pontos*).

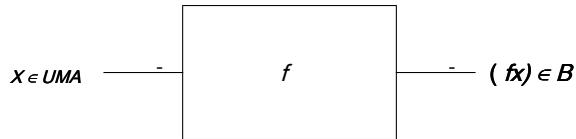
Muitas questões técnicas são deliberadamente ignorados e adiada para capítulos posteriores. A maioria dos exemplos de programação será fornecido na forma HASKELL linguagem de programação funcional. A.1 apêndice inclui as listas de alguns HASKELL módulos que complementam o HASKELL *Prelude padrão* e ajudar a “animar” os principais conceitos introduzidos neste capítulo.

2.1 Apresentando funções e tipos

A definição de uma função

$$f : A \rightarrow B \quad (2.1)$$

pode ser considerado como uma espécie de abstracção “processo”: é uma “caixa negra” que produz uma saída, uma vez que é fornecido com uma entrada:



A caixa não é realmente necessário para transmitir a abstração, um único rotulado seta su fi cing:

$$UMA \xrightarrow{f} B$$

Esta notação simplificada concentra-se no que é realmente relevante sobre *f*- que ele pode ser considerado como uma espécie de “contrato”:

f compromete-se para produzir um *B*-valor, desde que seja fornecido com um *UMA*-valor.

Como é de tal valor produzido? Em muitas situações se deseja ignorá-lo, porque é apenas *utilização* função *f*. Em outros, no entanto, um pode querer inspecionar os internos da “caixa preta”, a fim de conhecer a função de *regra de cálculo*. Por exemplo,

$$\begin{aligned} \text{succ} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{succ } n \text{ def } &= n + 1 \end{aligned}$$

expressa a regra de cálculo do *sucessor* função - a função *succ* que finds “o próximo número natural” - em termos de número natural adição e de número natural 1. O que acima entende por um “contrato” corresponde à assinatura da função, que é expresso pela seta $N \rightarrow N$ no caso de *succ* e que, aliás, pode ser compartilhado por outras funções, por exemplo $sq\ n \text{ def} = n^2$.

Na terminologia de programação se diz que *succ* e *sq* têm o mesmo “tipo”. Tipos desempenhar um papel preeminente na programação funcional (como fazem em outros programação paradigmas). Informalmente, eles fornecem a “cola”, ou material de interface, para colocar as funções em conjunto para obter funções mais complexas. Formalmente, uma disciplina “verificação de tipo” pode ser expressa em termos de regras de composição que verificam para wellformedness expressão funcional.

Tornou-se padrão para utilizar setas para indicar assinaturas de função ou tipos de função, recordando (2.1). Para indicar o facto de funcionar *f* aceita argumentos do tipo *UMA* e produz resultados de tipo *B*, usaremos as seguintes notações intercambiáveis: $f : B \leftarrow A$, $f : A \rightarrow B$, B , $\frac{\text{ao } f}{UMA \text{ ou } UMA \text{ } f} // B$. Isto corresponde a escrita $f :: a \rightarrow b$ no HASKELL linguagem de programação funcional, onde as variáveis de tipo são indicados por letras minúsculas. *UMA* será referido como o *domínio* do *f* e *B* será referido como o *código* do *f*. Ambos *UMA* e *B* são símbolos ou expressões que denotam conjuntos de valores, na maioria das vezes chamado *tipos*.

2.2 aplicação funcional

O que queremos funções para? Se esta pergunta a um médico ou engenheiro, a resposta é muito provável que seja: um quer funções para modelagem e raciocínio sobre o comportamento das coisas reais.

Por exemplo, a função *distância* $t = 60 \times f$ poderia ser escrito por um estudante de física da escola para modelar a distância (em, digamos, quilômetros) um carro irá conduzir (por hora) a uma velocidade média 60 km / hora. Quando questionados sobre o quanto longe o carro passou em 2,5 horas, tal modelo fornece uma resposta imediata: basta avaliar *distância* 2,5 obter 150 km.

Então nós temos um propósito ingênuo de funções: nós queremos que eles sejam *aplicado* a argumentos a fim de obter resultados. Funcional *aplicação* é denotada por justaposição, por exemplo $f a$ para *B, $\frac{\text{ao } f}{UMA \text{ e } a \in UMA}$, e associados para a esquerda: $f x y$ denota $(f x) y$ ao invés de $f(x, y)$.*

2,3 igualdade funcional e composição

A aplicação não é tudo o que queremos fazer com funções. Muito em breve o nosso estudante de física será capaz de falar sobre as propriedades do *distância* modelo, por exemplo, que a propriedade

$$\text{distância} (2 \times t) = 2 \times (\text{distância } t) \quad (2.2)$$

detém. Mais tarde, podemos aprender com ela ou ele que a mesma propriedade pode ser reescrita como *distância* (*duas vezes t*) = *duas vezes* (*distância t*), por função introduzir

duas vezes X = $2 \times x$. Ou mesmo simplesmente como

$$\text{distância} \cdot \text{duas vezes duas vezes} = \cdot \text{distância} \quad (2.3)$$

Onde " · " Denota encadeamento função de seta, como sugerido pelo desenho



onde espaço e tempo são modeladas por números reais R.

Este exemplo trivial ilustra alguns aspectos relevantes do paradigma de programação funcional. Qual versão do imóvel apresentado acima é “melhor”? a versão mencionar explicitamente variável *t* e parênteses que exigem (2.2)? a variável versão esconderijo *t* mas recorrer a funcionar *duas vezes* (2.3)? ou mesmo diagrama (2.4) sozinho?

A expressão (2.3) é claramente mais compacto do que (2.2). A tendência para a economia notação e compacidade é bem conhecido em toda a história da matemática. No século 16, por exemplo, algebrists iria escrever *12.cu.~* *p.18.ce.~ p.27.co.~ p.17* para o que é hoje em dia escrito como *12 X₃ + 18 X₂ + 27 x + 17*. Podemos achar tal sincopado notação estranha, mas não devemos esquecer que a seu tempo foi substituindo denotações expressão ainda mais obscuros e longas.

Por que as pessoas olham para notações compactas? Uma notação compacta leva a documentos mais curtos (menos linhas de código de programação) na qual os padrões são mais fáceis de identificar e raciocinar sobre. As propriedades podem ser expressas em claras, equações de uma linha longa, que são fáceis de memorizar. E diagramas como (2.4) pode ser facilmente desenhadas que nos permitem visualizar matemática em um formato gráfico.

Algumas pessoas argumentam que tais notação compacta “pointfree” (isto é, a notação que esconde variáveis, ou a função “de finição pontos”) é muito críptica para ser útil como um meio de programação prática. Na verdade, a programação pointfree

linguagens como de um Iverson PL ou FP Backus' ter sido mais respeitado do que amado pela comunidade programadores. Praticamente todas as linguagens de programação comerciais exigem variáveis e assim implementar a notação mais tradicional "pontual".

Ao longo deste livro, vamos adotar ambos, dependendo do contexto. A nossa programação escolhida meio H-ASKELL -misturas o pontual e estilos de programação pointfree de uma forma muito bem sucedida. A fim de mudar de um para o outro, precisamos de duas "pontes": uma igualdade de elevação para o nível funcional e o outro aplicativo função de elevação.

Em matéria de igualdade, note que o “=” entre (2,2) difere do que em (2.3): enquanto os antigos estados que dois números reais são o mesmo número, este último afirma que dois $R \leftarrow R$ funções são a mesma função. Formalmente, vamos dizer que duas funções $f, g: B \leftarrow UMA$ são iguais se eles concordam a nível pontual, que é²

$$f = g \text{ sse } \langle \forall um: a \in A: fa =_B ga \rangle \quad (2.5)$$

onde $=_B$ denota a igualdade no B -nível. Regra (2,5) é conhecido como *igualdade extensional*.

Quanto à aplicação, o estilo pointfree substitui-lo pelo conceito mais genérico de funcional *composição* sugerida pela função de seta encadeamento: onde quer que duas funções são tais que o tipo de destino de um deles, digamos B

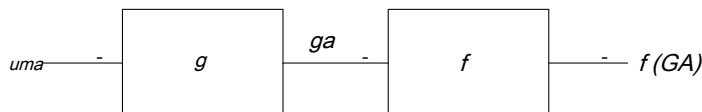
é o mesmo que o tipo de fonte do outro, digamos C é $\xrightarrow{\text{ao } f} B$, em seguida, uma outra função pode ser definida, $C \xrightarrow{\text{ao } f \cdot g} UMA$. Chamou o *composição do fe g; ou "f depois de g"* - que “colas” *fe g* juntos:

$$(f \cdot g) um \underset{\text{def}}{=} f(GA) \quad (2.6)$$

Esta situação é representada pela seta-diagrama a seguir



ou por bloco-diagrama



² Notação ed quantificação $\langle \forall x: P: Q \rangle$ significa: "para todos X no intervalo P , Q detém", Onde P e Q são expressões lógicas que envolvem x . Esta notação será utilizada esporadicamente neste livro.

12 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Portanto, a regra-tipo associado a composição funcional pode ser expressa como se segue:³

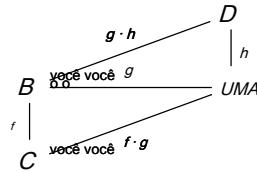
$$\begin{array}{c} C \xrightarrow{\alpha\alpha^f} B \\ B \xrightarrow{\alpha\alpha^g} UMA \\ \hline C \xrightarrow{\alpha\alpha^{f \cdot g}} UMA \end{array}$$

Composição é certamente o mais básico de todos os combinadores funcionais. É o primeiro tipo fi de “cola” que vem à mente quando os programadores precisa combinar ou funções da cadeia (ou processos) para obter funções mais elaboradas (ou processos).⁴

Isso é por causa de uma de suas propriedades mais relevantes,

$$(f \cdot g) \cdot h = f \cdot (g \cdot h) \quad (2.8)$$

representado por diagrama



que compartilha o padrão de, por exemplo,

$$(a + b) + c = a + (b + c)$$

e por isso é chamado de *associativo* propriedade da composição. Isso nos permite mover-se entre parênteses em expressões pointfree envolvendo composições funcionais, ou mesmo omiti-los por completo, por exemplo, por escrito $f \cdot g \cdot h \cdot Eu$ como uma abreviatura do $((f \cdot g) \cdot h) \cdot Eu$, ou de $(f \cdot (g \cdot h)) \cdot Eu$, ou de $f \cdot ((g \cdot h) \cdot Eu)$, etc. Para obter uma cadeia de n -muitos função composições a notação $\otimes_{i=1}^n f_{Eu}$ será aceitável como abreviatura de

$f_1 \cdot \dots \cdot f_n$.

2.4 funções de Identidade

Como livre devemos FUL fi II o “me dar uma *UMA* e eu vou dar-lhe uma *B*” Contrato da equação (2.1)? Em geral, a escolha de *f* não é única. Alguns *f*s vai fazer tão pouco quanto possível, enquanto outros vão laboriosamente calcular saídas não-triviais. Em um dos

³ Esta e outras regras-tipo para vir adotar o costume de layout “fractional”, lembra a usada em aritmética escolares para adição, subtração, etc.

⁴ Ele ainda tem um lugar em linguagens de script como UNIX Shell s, onde $f | g$ é a contrapartida shell de $g \cdot f$, para “processos” apropriadas *f* e *g*.

os extremos, nós fizemos funções que “não fazer nada” para nós, isto é, o valor acrescentado da sua produção quando comparado com a sua entrada equivale a nada: $f_a = \text{uma}$. Nesse caso $B = \text{UMA}$, é claro, e f é dito ser a *identidade* funcionar em UMA :

$$\begin{array}{l} \text{identidade}_B \text{UMA} \leftarrow \text{Ajuda uma uma} \\ \text{def} \\ = \text{uma} \end{array} \quad (2,9)$$

Note-se que cada tipo X “Tem” a sua identidade identidade_X . Subscritos será omitido onde quer implícita no contexto. Por exemplo, a notação da seta $N \xrightarrow{\text{identidade}} N$ nos salva da escrita identidade_N . Então, nós, muitas vezes, se referem a “a” função identidade em vez de “uma” função identidade.

Como úteis são funções de identidade? À primeira vista, eles parecem bastante desinteressante. Mas a interação entre composição e identidade, capturado pela seguinte equação,

$$f \cdot id = id \cdot f = f \quad (2,10)$$

será apreciado mais tarde. Esta propriedade compartilha o padrão de, por exemplo,

$$a + 0 = 0 + a = a$$

É por isso que dizemos que *identidade* é o *unidade (identidade)* da composição. Em um diagrama, (2.10) se parece com isso:



Note-se a analogia gráfica de diagramas (2,4) e (2,11). O último é interessante no sentido de que é *genérico*, segurando para cada f . Diagramas deste tipo são muito comuns e expressar propriedades importantes (e bastante ‘naturais’) de funções, como veremos mais adiante.

2,5 funções constantes

Em frente às funções de identidade, que não perdem qualquer informação, nós fizemos funções que perdem todos (ou quase todos) informações. Independentemente da sua entrada, a saída dessas funções é sempre o mesmo valor.

14 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Deixe C ser um domínio de dados não vazio e deixar $c \in C$. Então nós definiremos a seguinte propriedade para arbitrária UMA :

$$\underline{c} : UMA \rightarrow C \text{ c}\underset{\text{def}}{=} \underline{c}$$
(2.12)

A seguinte propriedade define funções constantes a nível pointfree,

$$\underline{c} \cdot f = c \quad \underline{\quad} \quad (2.13)$$

e é representado por um diagrama semelhante ao da (2.11):

$$\begin{array}{ccc} C & \xrightarrow{\text{oo } \underline{c}} & UMA \\ | & & | \\ \text{idéntidade} & & f \\ | & & | \\ C & \xrightarrow{\text{oo } \underline{c}} & B \end{array} \quad (2.14)$$

Claramente, $\underline{c} \cdot f = c \cdot g$, para qualquer f, g ; o que significa que qualquer diferença de que podem existir no comportamento entre tais funções é perdida.

Note-se que, estritamente falando, símbolo \underline{c} indica duas funções diferentes no diagrama (2.14): um, o que deveria ter escritas c_{UMA} , aceita entradas de UMA enquanto o outro, que deveria ter escrito c_B , aceita entradas de B :

$$\underline{c}_B \cdot f = c_{UMA} \quad \underline{\quad} \quad (2.15)$$

Esta propriedade será referido como o Constant fusão propriedade.

Tal como acontece com função identidade, subscritos serão omitido onde quer implícita no contexto.

Exercício 2.1. Use (2.5) para mostrar que $f \cdot h = h \cdot f = f$ tem a solução única $h = \text{idéntidade}$, cf. (2.10).

2

Exercício 2.2. o HASKELL Prelúdio prevê funções constantes: você escreve `const c para c`. Verifique se HASKELL atribui o mesmo tipo de expressões $f \cdot (\text{const } c)$ e $\text{const } (fc)$, para cada f e c . O que mais você pode dizer sobre essas expressões funcionais? Justificar.

2

2,6 Monics e épicos

função identidade e funções constantes são pontos limite do espectro funcional em relação à preservação da informação. Todas as outras funções são entre: elas perdem “alguma” informação, o que é considerado como desinteressante por algum motivo. Esta observação suporta o seguinte aforismo sobre uma faceta da programação funcional: é a *arte de transformar ou perda de informações de uma forma controlada e precisa*. Ou seja, a arte de construir a observação exata dos dados que fíts em um contexto ou exigência particular.

Como funções perder informações? Basicamente de duas maneiras diferentes: podem ser “cego” suficiente para confundir entradas diferentes, mapeando-os para a mesma saída, ou podem ignorar valores de sua codomain. Por exemplo, c confunde todos entradas por mapeamento-los todos para c . Além disso, ele ignora todos os valores da sua codomain além de c .

Funções que não confundir entradas são chamados *monics* (ou *injective* funções) e obedecer a seguinte propriedade: $B \xrightarrow{f} C$, E se $f \cdot h = f \cdot k$ então $h = k$, cf. diagrama

$$B \xrightarrow{f} C \quad \text{UMAé monic se, para cada par de funções } UMA \quad \xrightarrow{h, k} C$$

(Dizemos que f é “pós-canceláveis”). É fácil verificar que “a” função identidade é monic,

$$\begin{aligned} & \text{idéntidade} \cdot h = id \cdot k \Rightarrow h = K \\ \equiv & \{ \text{por (2.10)} \} \\ h = K & \Rightarrow h = K \\ \equiv & \{ \text{lógica de predicados} \} \\ & \text{VERDADEIRO} \end{aligned}$$

e que qualquer função constante c não é monic:

$$\begin{aligned} & c \cdot h = c \cdot k \Rightarrow h = K \\ \equiv & \{ \text{por (2.15)} \} \\ & \underline{c} = c \underline{\exists} h = K \\ \equiv & \{ \text{igualdade função é reflexiva} \} \text{ VERDADEIRO} \Rightarrow \\ & h = K \end{aligned}$$

$$\equiv \{ \text{lógica de predicado} \}$$

$$h = K$$

Assim, a implicação não se sustenta em geral (apenas se $h = k$).

Funções que não ignoram valores de sua codomain são chamados *épicos* (ou *surjective* funções) e obedecer a seguinte propriedade: *UMA* $\xrightarrow{\text{ao } f} B$ é *épico* se, por cada par de funções C $\xrightarrow{\text{ao } h, k} \text{UMA}$ se $h \cdot F = k \cdot F$ então $h = k$, cf. diagrama

$$C \xrightarrow{\text{ao } h, k} \text{UMA} \xrightarrow{\text{ao } f} B$$

(Dizemos que f é “pré-canceláveis”). Como esperado, a função identidade são épicas:

$$h \cdot ID = k \cdot \text{identidade} \Rightarrow h = K$$

$$\equiv \{ \text{por (2.10)} \}$$

$$h = K \Rightarrow h = K$$

$$\equiv \{ \text{lógica de predicados} \}$$

VERDADEIRO

Exercício 2.3. Em que circunstâncias é um épico função constante? Justificar.

2

2.7 Isos

Uma função $B \xrightarrow{\text{ao } f} \text{UMA}$ que é tanto mórfico e épica é dito ser *iso* (um isomorfismo, ou uma função bijective). Nesta situação, f sempre tem um *Converse* (ou *inverso*) $B_f \xrightarrow{\text{ao } f^{-1}} \text{UMA}$, que é tal que

$$f \cdot f^{-1} = \text{identidade}_B \wedge f^{-1} \cdot f = \text{id}_{\text{UMA}} \quad (2.16)$$

(ou seja f é invertível).

Isomorfismos são funções muito importantes porque eles convertem dados de um “formato”, diz *UMA*, para outro formato, digamos B , sem perda de informação. assim f e f^{-1} são fiéis protocolos entre os dois formatos *UMA* e B . Claro,

estes formatos contêm o mesmo “quantidade” de informações, embora os mesmos dados adota uma “forma” diferente em cada um deles. Em matemática, diz-se que *UMA* é *isomorphic* para *B* e se escreve $UMA \sim = B$ para expressar este fato.

domínios de dados isomórficas são considerados como “abstratamente” o mesmo. Note-se que, em geral, existe uma vasta gama de isos entre dois domínios de dados isomórficas. Por exemplo, vamos *dia da semana* sendo o conjunto de dias da semana,

$$\begin{aligned} \text{Dia da semana} = \\ \{ \text{Domingo, segunda-feira, Terça, quarta, quinta, sexta, sábado} \} \end{aligned}$$

e deixá-símbolo 7 denotam o conjunto $\{1, 2, 3, 4, 5, 6, 7\}$, qual é o *segmento inicial* do N contendo exatamente sete elementos. A seguinte função f , que associa cada dia da semana com o seu número “ordinal”,

$$\begin{aligned} f: \text{dia da semana} &\rightarrow 7 \\ f \text{ segunda-feira} &= 1 \\ f \text{ terça-feira} &= 2 \\ f \text{ quarta-feira} &= 3 \\ f \text{ quinta-feira} &= 4 \\ f \text{ sexta-feira} &= 5 \\ f \text{ sábado} &= 6 \\ f \text{ domingo} &= 7 \end{aligned}$$

é iso (suposição $f \sim =$). Claramente, $fd = i$ significa “ d é o *Eu*-º dia da semana”. Mas note-se que a função $gd \underset{\text{def}}{=} REM(fd, 7)$

1 é também um iso entre dia da semana e 7 . Enquanto

f Saudações *Segunda-feira* o primeiro dia da semana, g locais *domingo* nessa posição. Ambos f e g somos testemunhas de isomorfismo

$$\text{dia da semana} \sim = 7 \tag{2.17}$$

Isomorfismos são bastante flexível no raciocínio pontual. Se, por algum motivo, $f \cdot$ é encontrado mais acessível do que isomorfismo f no raciocínio, então as regras de manobra

$$f \cdot g = h \equiv g = f \cdot \cdot h \tag{2.18}$$

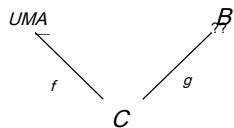
$$g \cdot f = h \equiv g = h \cdot f \cdot \tag{2.19}$$

pode ser de ajuda.

Finalmente, note que todas as classes de funções referido até agora - constantes, identidades, épicos, monics e isos - são fechados sob composição, isto é, a composição de duas constantes é uma constante, a composição dos dois épicos é épica, etc.

2.8 Colando funções que não compõem-produtos

composição função foi apresentada acima como base para a colagem de funções em conjunto, a fim de construir funções mais complexas. No entanto, nem todas as duas funções podem ser coladas por composição. Por exemplo, funções $f: A \leftarrow C$ e $g: B \leftarrow C$ não compõem um com o outro porque o domínio de um deles não é o codomain do outro. No entanto, ambos f e g compartilham o mesmo domínio C . Então, algo que podemos fazer em relação a colagem f e g juntos é desenhar um diagrama de expressar este fato, algo como



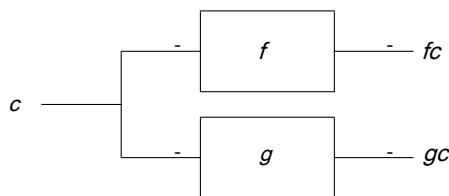
Porque f e g compartilham o mesmo domínio, as suas saídas podem ser emparelhados, ou seja, podemos escrever par ordenado (fc, gc) para cada $c \in C$. Tais pares pertencem ao produto cartesiano de UMA e B , isto é, ao conjunto

$$UMA \times B \underset{\text{def}}{=} \{(uma, b) / uma \in UMA \wedge b \in B\}$$

Assim, podemos pensar na operação quais pares as saídas de f e g como um novo combinator função $\langle f, g \rangle$ definida da seguinte forma:

$$\begin{aligned} \langle f, g \rangle &: C \rightarrow UMA \times B \\ \langle f, g \rangle c \underset{\text{def}}{=} & (fc, gc) \end{aligned} \tag{2.20}$$

Tradicionalmente, o combinator emparelhamento $\langle f, g \rangle$ é pronunciado “*f Dividido g*” (Ou “par f e g ”) E pode ser representado pelo seguinte ‘bloco’, ou ‘dados de fluxo’ diagrama:



2.8. COLAGEM funções que não componho - PRODUTOS 19

Função $\langle f, g \rangle$ mantém as informações de ambos f e g no produto cartesiano mesma maneira $UMA \times B$ mantém as informações de UMA e B . Assim, da mesma forma UMA de dados ou

B os dados podem ser recuperados a partir de $UMA \times B$ dados via o implícito projeções π_1 ou π_2 .

$$UMA^{\bullet, 0} \xrightarrow{\pi_1} UMA \times B \xrightarrow{\pi_2} \parallel B \quad (2,21)$$

definido por

$$\pi_1(a, b) = a \text{ e } \pi_2(a, b) = b$$

f e g podem ser recuperados a partir de $\langle f, g \rangle$ através dos mesmos projeções:

$$\pi_1 \cdot \langle f, g \rangle = f \text{ e } \pi_2 \cdot \langle f, g \rangle = g \quad (2,22)$$

Este fato (ou um par de fatos) será referido como o \times -cancelamento propriedade e é ilustrado no diagrama a seguir o que coloca tudo junto:

$$\begin{array}{ccc} UMA^{\bullet, 0} & \xrightarrow{\pi_1} & UMA \times B \\ & \searrow f & \downarrow \langle f, g \rangle \\ & C & \end{array} \quad \begin{array}{ccc} & & \xrightarrow{\pi_2} \parallel B \\ & \nearrow g & \\ C & & \end{array} \quad (2,23)$$

Em resumo, o tipo-regra associada ao combinator “split” é expresso por

$$\begin{array}{c} UMA^{\bullet, 0} \xrightarrow{f} C \\ B \xrightarrow{g} C \\ \hline UMA \times B \xrightarrow{\langle f, g \rangle} C \end{array}$$

UMA Dividido surge sempre que duas funções não compõem mas compartilham o mesmo domínio. O que cerca de colagem duas funções que falham um tal requisito, *por exemplo*

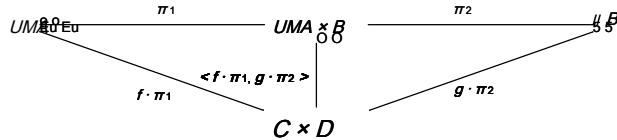
$$\begin{array}{c} UMA^{\bullet, 0} \xrightarrow{f} C \\ B \xrightarrow{g} D \\ \dots ? \end{array}$$

o $\langle f, g \rangle$ Dividido combinação não funciona mais. No entanto, uma forma de “ponte” os domínios de f e g , C e D respectivamente, é a considerá-los como alvos das projeções π_1 e π_2 do $C \times D$:

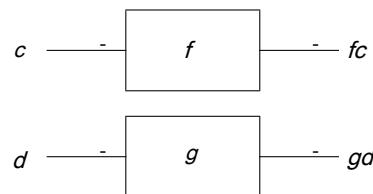
$$\begin{array}{ccc} UMA^{\bullet, 0} & \xrightarrow{\pi_1} & UMA \times B & \xrightarrow{\pi_2} \parallel B \\ f \downarrow & & & \downarrow g \\ C & \xrightarrow{\pi_1} & C \times D & \xrightarrow{\pi_2} \parallel D \end{array}$$

20 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

A partir deste diagrama $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ surge



mapeamento $C \times D$ para $UMA \times B$. Corresponde à aplicação “paralelo” de f e g que é sugerido pela seguinte figura:



combinação funcional $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ aparece com tanta frequência que ela merece notação especial - que vai ser expresso por $f \times g$. Então, por definição, temos

$$f \times g \text{ def } = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \quad (2.24)$$

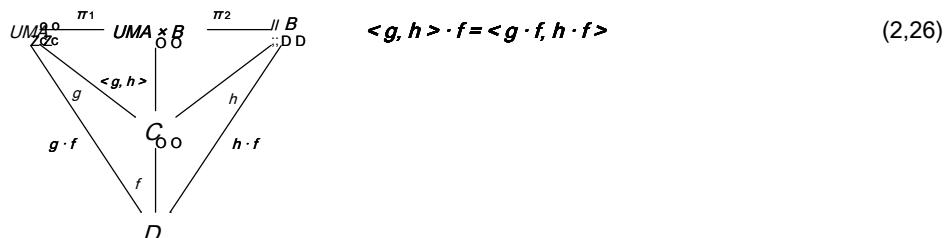
que é pronunciado “produto da f e g ” E tem de digitação-rule

$$\begin{array}{c} UMA \xrightarrow{f} C \\ B \xrightarrow{g} D \\ \hline UMA \times B \xrightarrow{f \times g} C \times D \end{array} \quad (2.25)$$

Observe a sobrecarga do símbolo “ \times ”, O qual é utilizado para designar tanto o produto cartesiano e produto funcional. Esta escolha de notação será feita totalmente justificada mais tarde.

Qual é a interação entre os combinadores funcionais $f \cdot g$ (composição), $\langle f, g \rangle$

(Dividido) e $f \times g$ (produtos) ? composição e Dividido se relacionam entre si por meio da seguinte propriedade, conhecida como \times - fusão:⁵



⁵ Note como essa lei pode ser considerada como uma prestação pointfree de (2.20).

2.8. COLAGEM funções que não componho - PRODUTOS 21

Isto mostra que *Dividido* é-distributiva direito com respeito à composição. Esquerda-distributividade não se sustenta, mas há algo que podemos dizer sobre $f \cdot \langle g, h \rangle$ em caso $f = i \times j$:

$$\begin{aligned}
& (\mathbf{Eu} \times j) \cdot \langle g, h \rangle \\
= & \quad \{ \text{por (2,24)} \} \\
& \langle \mathbf{Eu} \cdot \pi_1, j \cdot \pi_2 \rangle \cdot \langle g, h \rangle \\
= & \quad \{ \text{por } \times\text{-fusão (2,26)} \} \\
& \langle (\mathbf{Eu} \cdot \pi_1) \cdot \langle g, h \rangle, (j \cdot \pi_2) \cdot \langle g, h \rangle \rangle \\
= & \quad \{ \text{por (2,8)} \} \\
& \langle \mathbf{Eu} \cdot (\pi_1 \cdot \langle g, h \rangle), j \cdot (\pi_2 \cdot \langle g, h \rangle) \rangle \\
= & \quad \{ \text{por } \times\text{-cancelamento (2,22)} \} \\
& \langle \mathbf{Eu} \cdot g, j \cdot h \rangle
\end{aligned}$$

A lei que acabamos de derivada é conhecida como \times -absorção. (A intuição por trás dessa terminologia é que “*Dividido absorve* \times ”, Como um tipo especial de fusão.) É uma consequência da \times -fusão e \times -cancelamento e é descrito como se segue:

$$\begin{array}{ccc}
\begin{array}{c}
U \xrightarrow{\pi_1} UMA \times B \xrightarrow{\pi_2} B \\
\downarrow Eu \qquad \downarrow j \\
D \xrightarrow{\pi_1} D \times E \xrightarrow{\pi_2} E
\end{array}
&
\begin{array}{c}
\text{---} \\
g \qquad h
\end{array}
&
\begin{array}{c}
C
\end{array}
\end{array}
\quad (\mathbf{Eu} \times j) \cdot \langle g, h \rangle = \langle \mathbf{Eu} \cdot g, j \cdot h \rangle \quad (2,27)$$

Este diagrama fornece-nos com mais dois resultados sobre os produtos e projecções que possam ser facilmente justificados:

$$Eu \cdot \pi_1 = \pi_1 \cdot (\mathbf{Eu} \times j) \quad (2,28)$$

$$j \cdot \pi_2 = \pi_2 \cdot (\mathbf{Eu} \times j) \quad (2,29)$$

Duas propriedades especiais $f \times g$ são apresentados a seguir. O primeiro um expressa uma espécie de “bi-distribuição” de \times no que diz respeito à composição:

$$(g \cdot h) \times (\mathbf{Eu} \cdot j) = (g \times \mathbf{Eu}) \cdot (h \times j) \quad (2,30)$$

22 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Vamos nos referir a esta propriedade como o *x - propriedade functor*. A outra propriedade, que nós referimos como o *x - propriedade functor-id*, tem a ver com funções de identidade:

$$\text{identidade}_{UMA} \times \text{identidade}_B = \text{identidade}_{UMA \times B} \quad (2.31)$$

Estas duas propriedades serão identificadas como o *Propriedades functorial* do produto. Mais uma vez, esta escolha de terminologia será explicada mais tarde.

Vamos finalmente analisar a situação particular em que um *Dividido* é construído envolvendo projeções π_1 e π_2 só. Estes apresentam propriedades interessantes, por exemplo

$\langle \pi_1, \pi_2 \rangle = \text{identidade}$. Esta propriedade é conhecida como *x - reflexão* e é descrito como se segue:⁶

$$\begin{array}{ccc} UMA & \xrightarrow{\pi_1} & UMA \times B \\ \downarrow c & & \downarrow \text{identidade}_{UMA \times B} \\ UMA & \xrightarrow{\pi_2} & B \end{array} \quad \langle \pi_1, \pi_2 \rangle = \text{identidade}_{UMA \times B} \quad (2.32)$$

Sobre o quê $\langle \pi_2, \pi_1 \rangle$? Isto corresponde a um diagrama

$$\begin{array}{ccc} B & \xrightarrow{\pi_1} & B \times UMA \\ \downarrow c & & \downarrow \langle \pi_2, \pi_1 \rangle \\ B & \xrightarrow{\pi_2} & UMA \end{array}$$

que se parece muito o mesmo se submetido a um 180° rotação dos ponteiros do relógio (assim *UMA* e *B* troca de uns com os outros). Isto sugere que troca - o nome que adotamos para $\langle \pi_2, \pi_1 \rangle$ - é sua própria inversa; isto é verificado facilmente como segue:

$$\begin{aligned} & \text{troca} \cdot \text{troca} \\ = & \{ \text{por definição troca} \text{ def} \quad = \langle \pi_2, \pi_1 \rangle \} \\ & \langle \pi_2, \pi_1 \rangle \cdot \text{troca} \\ = & \{ \text{por } x\text{- fusão (2.26)} \} \\ & \langle \pi_2 \cdot \text{troca}, \pi_1 \cdot \text{troca} \rangle \\ = & \{ \text{de finição de troca duas vezes} \} \\ & \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \\ = & \{ \text{por } x\text{- cancelamento (2.22)} \} \end{aligned}$$

⁶ Para obter uma explicação da palavra "reflexão" Em nome escolhido para esta lei (e para outras que virão) ver secção 2.13 mais tarde.

2.8. COLAGEM funções que não componho - PRODUTOS 23

$$\begin{aligned} & \langle \pi_1, \pi_2 \rangle \\ = & \{ \text{por } x - \text{re flexão (2,32)} \} \\ & \text{identidade} \end{aligned}$$

Assim sendo, troca é iso e estabelece o seguinte isomorfismo

$$UMA \times B \sim = B \times UMA \quad (2,33)$$

que é conhecido como o *propriedade comutativa* do produto.

O "tipo de dados de produto" $UMA \times B$ é essencial para o processamento de informações e está disponível em praticamente todas as linguagens de programação. em HASKELL se escreve (UMA, B) denotar $UMA \times B$, para UMA e B dois tipos de dados prede fi nidos, fst denotar π_1 e snd denotar π_2 . Na linguagem de programação C este tipo de dados é chamado de "tipo de dados struct",

```
struct {
    Um primeiro; B
    segundo;
};
```

enquanto em Pascal ele é chamado de "tipo de dados record":

```
registro
    primeiro um;
    segunda: final B;
```

Isomorfismo (2,33) pode ser re-interpretada neste contexto como uma garantia de que *não se perder (ou ganhar) qualquer coisa em troca de campos em tipos de dados de registro*. C ou Pascal programadores sabem também que Gravação de nidificação campo tem o mesmo status, que é dizer que, por exemplo, tipo de dados

<pre>registro F: A; S: registro</pre>	<pre>registro F: Registro</pre>
$F: B; S:$	é o mesmo que em abstracto
$C; \text{fim};$	$\text{final}; S; C; \text{fim};$

Na verdade, esta é uma outra isomorfismo bem conhecido, conhecido como o *propriedade associativa* do produto:

$$UMA \times (B \times C) \sim = (UMA \times B) \times C \quad (2,34)$$

Este é estabelecida pela $UMA \times (B \times C)$ $\xrightarrow{\text{assocr}} (UMA \times B) \times C$, que é pronunciada “Associado à direita” e é definido por

$$\text{assocr def} = \langle \pi_1 \cdot \pi_1, \pi_2 \times \text{idéntidade} \rangle \quad (2,35)$$

Seção A.1 no apêndice apresenta uma extensão para o HASKELL *Prelude padrão* que faz com que tais como isomorfismos troca e assocr acessível. Neste módulo, a sintaxe concreta escolhida para $\langle f, g \rangle$ é $f \times g$ dividida e o escolhido para $f \times g$ é $f > g$.

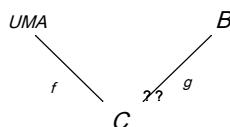
Exercício 2.4. Confiar em (2.24) para revelar as propriedades (2,30) e (2,31).

2

2.9 Colando funções que não compõem - co-produtos

o *Dividido* combinator funcional surgiu na seção anterior como uma espécie de cola para combinar duas funções que não compõem mas compartilham o mesmo domínio. A situação “dual” de duas funções não combináveis $F: C \leftarrow UMA$ e $g: C \leftarrow B$

que, contudo, compartilham o mesmo codomain está representado na



É claro que o tipo de cola que precisamos, neste caso, deve torná-lo possível aplicar f no caso que estamos no “UMA-Side” ou aplicar g no caso que estamos no “B-lado” do diagrama. Vamos escrever $[f, g]$ para designar o novo tipo de combinator. Sua codomain será C . E sobre seu domínio?

Precisamos descrever o tipo de dados que é “tanto um UMA ou um B ”. Desde a UMA e B são conjuntos, podemos pensar $UMA \cup B$ como um tal tipo de dados. Isso funciona no caso UMA e B são conjuntos disjuntos, mas onde quer que a intersecção $UMA \cap B$ é não-vazia é indecidível se um valor $X \in UMA \cap B$ é um “ UMA -valor” ou um “ B -valor”. No limite, se $A = B$ então $UMA \cup B = A = B$, isto é, não inventaram um

novo tipo de dados em tudo. Estes cuidados pode ser contornada pelo recurso a *união disjunta*,

$$A + B \text{ def } = \{ Eu_1 \text{ uma } / \text{uma} \in UMA \} \cup \{ Eu_2 \text{ b } / b \in B \}$$

assumindo as funções de “marcação”

$$Eu_1 \text{ a } = (t_1, a), Eu_2 \text{ b } = (t_2, b) \quad (2,36)$$

com tipos $7 UMA \quad Eu_1 : A + B \xrightarrow{\alpha_0} B$. Conhecer os valores exatos de marcas t_1 e t_2 é não é essencial para a compreensão do conceito de uma união disjunta. Ele su escritórios para saber que Eu_1 e Eu_2 marcar de forma diferente ($t_1 \neq t_2$) e consistentemente.

Os valores de $A + B$ pode ser pensado como “cópias” de UMA ou B valores que são “carimbados”, com diferentes marcas, a fim de garantir que os valores que são simultaneamente em UMA e B não se misturam. Por exemplo, a realizações da seguinte $A + B$ na linguagem de programação C,

```
struct {
    int tag; /* 1,2 */ union {
        Um IFA; B
        IFB;
    } dados;
};
```

ou em Pascal,

```
registro
caso
tag: número inteiro
de x = 1: (P:
A); 2: (S: B) fim;
```

adotar etiquetas inteiros. Na HASKELL *Prelude Padrão*, a $A + B$ tipo de dados é realizada por

dados De qualquer ab = Esquerda um | b direita

⁷As funções de marcação Eu_1 e Eu_2 são geralmente referidos como o *injeções* da união disjunta.

26 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Assim, Esquerda e Certo pode ser pensado como as injeções Eu_1 e Eu_2 nesta realização.

Nesse nível de abstração, união disjunta $A + B$ é chamado de *coproduct* do UMA e B , em cima do que define a nova combinator $[f, g]$ (pronuncia-se “quer f ou g ”) do seguinte modo:

$$\begin{aligned} [f, g] &: A + B \xrightarrow{\{x = i_1 \text{ uma } \rightarrow fa\}} C \\ [f, g] X_{\text{def}} &= x = i_2 b \Rightarrow gb \end{aligned} \tag{2.37}$$

Como fizemos para os produtos, podemos expressar tudo isso em um diagrama:

$$\begin{array}{ccc} \text{UMA}_{Eu_1} & A + B & B \\ & \swarrow f & \xrightarrow{\alpha_0^{Eu_2}} \\ & \# & \downarrow [f, g] \\ C & \nearrow g & \end{array} \tag{2.38}$$

É interessante notar que este diagrama é semelhante ao traçado para produtos -o que tem de inverter as setas, substituir projecções por injeções e o *Dividido* pela seta ou 1. Isso expressa o fato de que *produtos* e *coproduct* está *dual* construções matemáticas (compare com *seno* e *co-seno* em trigonometria). Esta dualidade é de grande economia conceitual, porque tudo o que podemos dizer sobre o produto $UMA \times B$ pode ser reformulada para coproduto $A + B$. Por exemplo, podemos introduzir a soma de duas funções $f + g$ como a noção dupla para produto $f \times g$:

$$f + g \underset{\text{def}}{=} [Eu_1 \cdot f, i_2 \cdot g] \tag{2.39}$$

A lista de + -leis seguir fornece evidência eloquente dessa dualidade:

+ -cancelamento:

$$\begin{array}{ccc} \text{UMA}_{Eu_1} & A + B & B \\ & \swarrow g & \xrightarrow{\alpha_0^{Eu_2}} \\ & \# & \downarrow [g, h] \\ C & \nearrow h & \end{array} \quad [g, h] \cdot Eu_1 = g, [g, h] \cdot Eu_2 = h \tag{2.40}$$

+ -refl exão:

$$\begin{array}{ccc} \text{UMA}_{Eu_1} & A + B & B \\ & \swarrow Eu_1 & \xrightarrow{\alpha_0^{Eu_2}} \\ & \# & \downarrow \text{id}^{A+B} \\ A + B & \nearrow Eu_2 & \end{array} \quad [Eu_1, Eu_2] = \text{id}^{A+B} \tag{2.41}$$

+ - fusão:

$$\begin{array}{ccc}
 \text{UMA} \xrightarrow{\text{Eu}_{1\#}} A + B & \xrightarrow{\text{Eu}_2} B & \\
 \downarrow g \quad \# \quad \downarrow [g, h] & \downarrow f & \\
 D & C & \\
 \downarrow f \quad \# \quad \downarrow h & & \\
 & & B
 \end{array} \quad f \cdot [g, h] = [f \cdot g, f \cdot h] \quad (2.42)$$

+ - absorção :

$$\begin{array}{ccc}
 \text{UMA} \xrightarrow{\text{Eu}_{1\#}} A + B & \xrightarrow{\text{Eu}_2} B & \\
 \downarrow Eu \quad i+j & \downarrow j & \\
 D \xrightarrow{\text{Eu}_{1\#}} D + E & \xrightarrow{\text{Eu}_2} E & \\
 \downarrow g \quad \# \quad \downarrow [g, h] & \downarrow h & \\
 C & &
 \end{array} \quad [g, h] \cdot (i+j) = [g \cdot i, h \cdot j] \quad (2.43)$$

+ - functor:

$$(g \cdot h) + (l \cdot j) = (g + Eu) \cdot (h + j) \quad (2.44)$$

+ - functor-id:

$$\text{idénticidade}_A + \text{idénticidade}_B = \text{idénticidade}_{A+B} \quad (2.45)$$

Em resumo, a digitação-regras do ou e soma combinadores são como se segue:

$$\begin{array}{ccc}
 C \xrightarrow{\text{ao } f} \text{UMA} & & C \xrightarrow{\text{ao } f} \text{UMA} \\
 C \xrightarrow{\text{ao } g} B & \xrightarrow{\text{ao } g} & D \xrightarrow{\text{ao } g} B \\
 \hline
 C \xrightarrow{\text{ao } [f, g]} A + B & & C + D \xrightarrow{\text{ao } f+g} A + B
 \end{array} \quad (2.46)$$

Exercício 2.5. Por analogia (dualidade) com troca, mostram que $\text{[Eu}_2, \text{Eu}_1]$ é sua própria inversa e assim esse fato

$$A + B \sim = B + A \quad (2.47)$$

detém.

2

Exercício 2.6. Dualize (2.35), ou seja, escrever o *iso* que testemunhas fato

$$A + (B + C) \sim = (A + B) + C \quad (2.48)$$

da direita para esquerda. Use o ou sintaxe disponível a partir do HASKELL Prelude padrão para codificar esta iso em HASKELL.

2

2.10 produtos de mistura e co-produtos

construções de tipo de dados $UMA \times B$ e $A + B$ foram introduzidos acima, como os dispositivos necessários para a expressão do codomain de *splits* ($UMA \times B$) ou o domínio de *eithers* ($A + B$). Portanto, um valores de mapeamento função de um co-produto (digamos $A + B$) para valores de um produto (digamos $UMA \times B$), pode ser expresso alternativamente como uma *ou* ou como um *Dividido*. No primeiro caso, ambos os componentes do *ou* combinator são *divide*. Neste último caso, ambos os componentes do *Dividido* combinator são *eithers*.

Esta troca de formato no definindo tais funções é conhecido como o *Direito de câmbio*. Afirma a igualdade funcional que segue:

$$[<f, g>, <h, k>] = <[f, h], [g, k]> \quad (2.49)$$

Ele pode ser verificado por tipo de inferência que tanto o lado esquerdo e as expressões lado direito

dessa igualdade têm o tipo $B \times D \leftarrow A + C$, para B

$$D \xrightarrow{\alpha\alpha^g} UMA \quad B \xrightarrow{\alpha\alpha^h} C \quad \text{e} \quad D \xrightarrow{\alpha\alpha^k} C.$$

$$\xrightarrow{\alpha\alpha^f} UMA$$

Um exemplo de uma função que está no formato de-lei de câmbio é isomorfismo

$$UMA \times (B + C) \xrightarrow{\alpha\alpha^{\text{undistr}}} (UMA \times B) + (A \times C) \quad (2.50)$$

(pronunciar *undistr* como “un-distribuir-direito”), que é definido por

$$\text{undistr} \underset{\text{def}}{=} [\text{idéntidade} \times Eu_1, \text{idéntidade} \times Eu_2] \quad (2.51)$$

e testemunhas do fato de que o produto distribui através do co-produto:

$$UMA \times (B + C) \sim = (UMA \times B) + (A \times C) \quad (2.52)$$

Neste contexto, suponha que nós sabemos de três funções D e F

$\xrightarrow{\alpha\alpha h} C$. By (2.46) podemos inferir $E + F \xrightarrow{\alpha\alpha g+h} B + C$. Então, por (2.25) que infer

$$D \times (E + F) \xrightarrow{\alpha\alpha f \times (g+h)} UMA \times (B + C) \quad (2.53)$$

Assim, faz sentido combinar produtos e somas de funções e as expressões que denotam essas combinações têm a mesma “forma” (ou padrão simbólico) como as expressões que denotam o seu domínio e gama - $\alpha\ldots \times (\cdot\ldots + \cdot\ldots)$

“Moldar” neste exemplo. Na verdade, se nós abstrato tal padrão através de algum símbolo, digamos F - isto é, se de fí ne

$$F(\alpha, \beta, \gamma)_{\text{def}} = \alpha \times (\beta + \gamma)$$

- então podemos escrever $F(D, E, F) \xrightarrow{\alpha\alpha F(f, g, h)} F(A, B, C)$ para (2.53).

Este tipo de abstração funciona para cada combinação de produtos e co-produtos. Por exemplo, se nós agora abstrato lado direito de (2.50) via padrão

$$G(\alpha, \beta, \gamma)_{\text{def}} = (\alpha \times \beta) + (\alpha \times \gamma)$$

temos $G(f, g, h) = (f \times g) + (f \times h)$, uma função que mapeia $G(A, B, C) = (UMA \times B) + (A \times C)$ para $G(D, E, F) = (D \times E) + (D \times F)$. Tudo isso pode ser colocado em um diagrama

$$\begin{array}{ccc} F(A, B, C) & \xrightarrow{\text{undistr}} & G(A, B, C) \\ F(f, g, h) & | & | \\ F(D, E, F) & & G(D, E, F) \end{array}$$

que se desenvolve a

$$\begin{array}{ccc} UMA \times (B + C) & \xrightarrow{\text{undistr}} & (UMA \times B) + (A \times C) \\ f \times (g+h) & | & | \\ D \times (E + F) & & (D \times E) + (D \times F) \end{array} \quad (2.54)$$

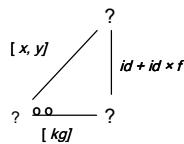
30 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

uma vez o F e G padrões são instanciado. Um tópico interessante que decorre (completando) neste diagrama será discutido na próxima seção.

Exercício 2.7. Aplicar o direito de câmbio para undistr.

2

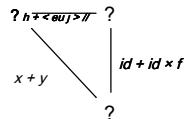
Exercício 2.8. Completar o “?” S em diagrama



e depois resolver a equação implícita para X e y.

2

Exercício 2.9. Repetir o exercício de 2,8 no que diz respeito ao diagrama



2

Exercício 2.10. Mostre que $\langle [f, h] \cdot (\pi_1 + \pi_1), [g, k] \cdot (\pi_2 + \pi_2) \rangle$ reduz-se a $[f \times g, h \times k]$.

2

2.11 tipos de dados elementares

Até agora temos falado principalmente sobre tipos de dados arbitrários representados por letras maiúsculas *UMA*, *B*, *etc* (minúsculas *uma*, *b*, *etc* no HASKELL ilustrações). Também mencionamos *R*, *Bool* e *N* e, em particular, o fato de que podemos associar a cada

número natural n Está **segmento inicial** $n = \{1, 2, \dots, N\}$. Nós estendemos isso para Nº declarando $0 = \{\}$ e para $n > 0$, $n + 1 = \{n + 1\} \cup n$.

segmentos iniciais podem ser identificados com os tipos enumerados e são considerados como tipos de dados primitivos em nossa notação. Nós adotamos a convenção de que tipos de dados primitivos estão escritos no *sem serif* font e, portanto, estritamente falando, n é distinta n : este último representa um número natural, enquanto o primeiro indica um tipo de dados.

tipo de dados 0

Entre tais tipos enumerados, 0 é o menor porque ele está vazio. Isto é o **Vazio tipo de dados em HASKELL**, que não tem nenhum construtor em tudo. Tipo de dados 0 (que tendem a escrever simplesmente como 0) pode não parecer muito “útil” na prática, mas é de interesse teórico. Por exemplo, é fácil verificar que as seguintes propriedades “óbvias” segurar,

$$A + 0 \sim = UMA \quad (2,55)$$

$$UMA \times 0 \sim = 0 \quad (2,56)$$

onde a segunda é realmente uma igualdade: $UMA \times 0 = 0$.

tipo de dados 1

Em seguida na sequência de segmentos iniciais, encontramos 1, que é conjunto singleton $\{1\}$.

Como útil é esse tipo de dados? Note-se que cada tipo de dados **UMA** contendo exatamente um elemento é isomorfo a $\{1\}$, por exemplo $A = \{\text{NADA}\}$, $A = \{0\}$, $A = \{1\}$, $A = \{\text{FALSO}\}$, etc.. Nós representamos esta classe de tipos únicos por 1.

Lembre-se que tipos de dados isomórficas têm o mesmo poder expressivo e por isso são “abstratamente idêntica”. Assim, a escolha real de habitante por tipo de dados 1 é irrelevante, e podemos substituir qualquer Singleton especial definida por outro sem perda de informação. Isto é evidente a partir do seguinte, isomorfismo observando,

$$UMA \times 1 \sim = UMA \quad (2,57)$$

que pode ser lido informalmente como segue: se o segundo componente de um registro (“struct”) não pode mudar, então é inútil e pode ser ignorado. Seletor π_1 é, neste contexto, um iso mapeamento do lado esquerdo de (2,57) para o seu lado direito. Seu inverso é $\langle id, c \rangle$ Onde c é uma escolha particular de habitante por tipo de dados 1.

Em resumo, quando se refere ao tipo de dados 1 que significa um tipo arbitrário Singleton, e existe uma iso única (e o seu inverso) entre dois desses tipos únicos.

32 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

o HASKELL Representativo de 1 é tipo de dados (), chamado de *tipo de unidade*, que contém exatamente o construtor (). Pode parecer confuso para denotar o tipo e seu habitante exclusivo pelo mesmo símbolo, mas não é, pois HASKELL mantém o controle de tipos e construtores em conjuntos de símbolos separados.

Note que qualquer função do tipo de $UMA \rightarrow 1$ é obrigado a ser uma função constante. Esta função, geralmente chamado de “bang”, ou “afundar” função, é indicado por um ponto de exclamação:

$$! : UMA \rightarrow 1 \quad (2,58)$$

Claramente, é o único função do seu tipo. (Você pode escrever um diferente, do mesmo tipo?)

Finalmente, o que podemos dizer sobre $1 + UMA$? cada função B $\frac{\text{ao } f}{\text{ação deste tipo é obrigado a ser um ou } [b_0, g]} 1 + UMA$ obser-
 $\frac{\text{ao } g}{\text{muito semelhantes para o tratamento de um ponteiro em C ou P Ascal: nós "puxar uma corda" e quer não}} UMA$ Isto é
 recebemos nada (1) ou temos algo útil do tipo B . Em tal contexto, a programação “nada” acima significa um fí valor NED prede NADA. Esta analogia suporta a nossa preferência na sequela para NADA como habitante canónica do tipo de dados

1. Na verdade, vamos nos referir a $1 + UMA$ (ou $A + 1$) como o “ponteiro para UMA ” tipo de dados. Isto corresponde ao Talvez Tipo de construtor do HASKELL *Prelude padrão*.

tipo de dados 2

Vamos inspecionar o $1 + 1$ exemplo da construção “ponteiro” apenas mencionado acima. qualquer observação B $\frac{\text{ao } f}{\text{ções: }} 1 + 1$ pode ser decomposto em duas fun- constante ções: $f = /b_1, b_2$. Agora, suponha que $B = [b_1, b_2]$ para $b_1 \neq b_2$. Então $1 + 1 \sim = B$ vai realizar, por qualquer escolha de habitantes b_1 e b_2 . Portanto, estamos em uma situação semelhante à 1: usaremos símbolo 2 para representar a classe abstrata de todas essas B s contendo exactamente dois elementos. Portanto, podemos escrever:

$$1 + 1 \sim = 2$$

Claro, $Bool = \{ VERDADEIRO, FALSO \}$ e segmento inicial $2 = \{ 1, 2 \}$ Há, nesta classe abstrata. Na seqüência, vamos mostrar alguma preferência para a escolha particular de habitantes $b_1 = VERDADE$ e $b_2 = FALSO$, o que nos permite usar o símbolo 2 Em lugares onde $Bool$ é esperado. Claramente,

$$2 \times UMA \sim = A + A \quad (2,59)$$

Exercício 2.11. Derivar o isomorfismo

$$(B + C) \times UMA \xrightarrow{\text{o.o}} (B \times A) + (C \times A) \quad (2.60)$$

de undistr (2.50) e outros isomorfismos estudados até agora.

2

Exercício 2.12. Além disso, mostram que a (2.59) segue a partir de (2.60) e, em termos práticos, se relacionam HASKELL expressão

ou (*split (const True) id*) (*split (const False) id*)

ao mesmo isomorfismo (2.59).

2

2,12 propriedades naturais

Vamos retomar a discussão sobre undistr e as duas outras funções no diagrama (2.54). Que tal usar undistr -se a encerrar este diagrama, na parte inferior? Note-se que de fínição (2.51) trabalha para D, E e F da mesma forma que faz para A, B e C . (Na verdade, a escolha particular de símbolos A, B e C . Em (2.50) foi bastante arbitrária) Portanto, temos:

$$\begin{array}{ccc} UMA \times (B + C) & \xrightarrow{\text{undistr}} & (UMA \times B) + (A \times C) \\ f \times (g \text{ } H+) & | & | \\ D \times (E + F) & \xrightarrow[\text{undistr}]{\text{o.o}} & (D \times E) + (D \times F) \end{array}$$

que expressa uma propriedade muito importante de undistr:

$$(f \times (g + h)) \cdot \text{undistr} = \text{undistr} \cdot ((f \times g) + (F \times h)) \quad (2.61)$$

Isso é chamado de *natural* propriedade de undistr. Este tipo de alojamento (muitas vezes chamado de "livre" ao invés de "natural") Não é um privilégio de undistr. Por uma questão de fato, todas as funções interface padrões como F ou G acima irá exibir o seu próprio *natural* propriedade. Além disso, já citamos *natural* Propriedades sem

34 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

mencioná-lo. Recordar (2.10), por exemplo. Esta propriedade (estabelecendo *identidade* Enquanto o *unidade* de composição) é, afinal de contas, o *natural*/propriedade de *identidade*. Neste caso, temos $F \alpha = G \alpha = \alpha$, como pode ser facilmente observado no diagrama (2.11).

Em geral, *natural*/propriedades são descritos por diagramas em que dois “cópias” do operador de interesse são desenhados como setas horizontais:

$$\begin{array}{ccc}
 \text{UMA} & F \text{ UMA}^{\alpha\alpha} \xrightarrow{\varphi} & G \text{ UMA} \\
 f \downarrow & F f \downarrow & \downarrow G f \\
 B & F B \xrightarrow{\alpha\alpha} & G B \\
 & \varphi &
 \end{array} \quad (2.62)$$

Observe que f é universalmente quantificado, ou seja, o *natural*/propriedade vale para todos os $f: B \leftarrow \text{UMA}$.

Diagrama (2.62) corresponde aos padrões unárias F e G . Como vimos com *undistr*, Outras funções (g, h etc) entram em jogo para os padrões multiary. A r muito importante que será atribuído ao longo deste livro a estes F, G, etc “formas” ou padrões que são partilhadas por expressões funcionais pointfree e por seu domínio e expressões codomain. A partir do capítulo 3 em diante vamos nos referir a eles pelo seu nome próprio - “functor” - que é padrão em matemática e ciência da computação. Então nós também irá explicar os nomes atribuídos às propriedades tais como, por exemplo, (2.30) ou (2.44).

Exercício 2.13. Mostre que (2.28) e (2.29) são natural propriedades. Dualize estas propriedades. dica: diagrama de Sensibilidade (2.43).

2

Exercício 2.14. estabelecer a natural propriedades do troca (2.33) e assoc (2.35) isomorfismos.

2

Exercício 2.15. Desenhe a propriedade natural da função $\varphi = \text{troca} \cdot (\text{id} \times \text{troca})$ como um diagrama, que é, identificar F e G na (2.62) para esse caso.

Faça o mesmo para $\varphi = \text{coswap} \cdot (\text{Swap} + \text{swap})$ Onde coswap = [Eu2, Eu1].

2

2.13 propriedades Universal

construções funcionais $\langle f, g \rangle$ e $[f, g]$ - e os seus derivados $f \times g$ e $f + g$

- proporcionar uma boa ilustração sobre o que se entende por um *combinator programa* em uma abordagem de composição para a programação: o combinador é apresentada equipado com uma concisa *um conjunto de propriedades* que permitem aos programadores para transformar programas, razão sobre eles e executar cálculos úteis. Isto levanta uma *metodologia de programação* que é científica e estável.

Essas propriedades têm nomes padrão, tais como *cancelamento, reflexão, fusão, absorção, etc..* Onde é que estes nomes vêm? Como regra geral, para cada combinador ser de fí nido um tem de definir construções adequados ao -level “interface” ⁸, *por exemplo*

UMA × B e A + B. Estes não são escolhidos ou inventado ao acaso: cada um é definido de uma forma tal que a combinador associado é exclusivamente definido. Esta é assegurada por uma chamada propriedade universal a partir do qual os outros podem derivada.

tome produto *UMA × B*, por exemplo. Sua propriedade universal afirma que, para cada par de setas *UMA*

$$\frac{\text{oo}^f}{\text{oo}^g} C \text{ e } B \quad \frac{\text{oo}^g}{\text{oo}^k} C , \text{ existe uma seta } UMA \times B \quad \frac{\text{oo}^{\langle f, g \rangle}}{\text{oo}^k} C$$

de tal modo que

$$k = \langle f, g \rangle \Leftrightarrow \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases} \quad (2.63)$$

detém - diagrama de Sensibilidade (2.23) - para todos *UMA × B* $\frac{\text{oo}^k}{\text{oo}^{\langle f, g \rangle}} C .$

Note-se que (2.63) é uma *equivalência*, afirmado implicitamente que $\langle f, g \rangle$ é o *único arrow* satisfazer a propriedade à direita. Na verdade, ler (2.63) na \Rightarrow direção e deixe *k* estar $\langle f, g \rangle$. Então $\pi_1 \cdot \langle f, g \rangle = f$ e $\pi_2 \cdot \langle f, g \rangle = g$ vai realizar, o que significa que $\langle f, g \rangle$ eficazmente obedece a propriedade à direita. Em outras palavras, nós deduzimos \times - cancelamento (2.22). Leitura (2.63) no \Leftarrow sentido, entendemos que, se alguma *k* satisfaz tais propriedades, em seguida, ele “tem de ser” o mesmo como seta $\langle f, g \rangle$.

A relevância da propriedade universal (2.63) é que ele oferece uma forma de *resolver equações* de forma $k = \langle f, g \rangle$. Tomemos por exemplo o seguinte exercício: pode a identidade pode ser expressa, ou “refletida”, usando este combinador? Nós apenas resolver o equiation $id = \langle f, g \rangle$ para *f e g*:

$$id = \langle f, g \rangle \equiv$$

{propriedade universal (2.63)}

⁸ No contexto atual, *programas* “São” funções e progra- *Interfaces* “São” os tipos de dados envolvidos em assinaturas funcionais.

$$\{ \pi_1 \cdot id = f$$

$$\pi_2 \cdot ID = g$$

$$\equiv \{ \text{por (2.10)} \}$$

$$\{ \pi_1 = f$$

$$\pi_2 = g$$

A equação tem as soluções exclusivas $f = \pi_1$ e $g = \pi_2$ que, uma vez substituído no próprio equação, rendimento

$$id = < \pi_1, \pi_2 >$$

ou seja, nada, mas o \times -re fl exão lei (2.32).

Todas as outras leis podem ser calculados a partir da propriedade universal de uma forma similar. Por exemplo, a \times -fusion (2.26) A lei é obtida resolvendo a equação $k = < eu j >$ novamente para f e g ; mas desta vez fixando $k = < eu j > \cdot h$, assumindo eu e h dado:⁹:

$$< eu j > \cdot h = < f, g > \equiv$$

{ propriedade universal (2.63)}

$$\{ \pi_1 \cdot (< eu j > \cdot h) = f$$

$$\pi_2 \cdot (< eu j > \cdot h) = g$$

$\equiv \{ \text{composição é associativa (2.8)} \}$

$$\{ (\pi_1 \cdot < eu j >) \cdot h = f$$

$$(\pi_2 \cdot < eu j >) \cdot h = g$$

$\equiv \{ \text{por } \times\text{-cancelamento (derivado acima)} \}$

$$\{ Eu \cdot h = f$$

$$j \cdot h = g$$

Substituting as soluções $f = i \cdot h$ e $g = j \cdot h$ na equação, temos o \times -lei de fusão: $< eu j > \cdot h = < Eu \cdot h, j \cdot h >$.

Isso levará cerca do mesmo esforço para derivar *Dividido igualdade estrutural*

$$< eu j > = < f, g > \Leftrightarrow \{ i = f \quad j = g \} \tag{2.64}$$

⁹ Resolvendo equações deste tipo é uma reminiscência de muitos cálculos semelhantes realizados em matemática da escola e cursos de física. O espírito é o mesmo. A diferença é que desta vez não está a calcular os débitos da bomba de água, acelerações, velocidades, ou outras entidades físicas: as soluções de nossas equações são (apenas) funcional *programas*.

de propriedade universal (2,63) - deixe $k = \langle e u j \rangle$ e resolver.

Argumentos semelhantes podem ser construídas em torno da propriedade universal do co-produto,

$$k = [f, g] \Leftrightarrow \begin{cases} k \cdot Eu_1 = f \\ k \cdot Eu_2 = g \end{cases} \quad (2,65)$$

a partir do qual a igualdade estrutural ou s pode ser inferido,

$$[i, j] = [f, g] \Leftrightarrow \begin{cases} i = f \\ j = g \end{cases} \quad (2,66)$$

bem como as outras propriedades que sabemos sobre este combinator.

Exercício 2.16. Mostre que assocr (2,35) é iso, resolvendo a equação $\text{assocr} \cdot \text{assocd} = \text{id}$ para assocr . dica: não ignorar o papel da propriedade universal (2,63) no cálculo.

2

Exercício 2.17. Provar a igualdade: $[\langle F, K \rangle, \langle g, k \rangle] = \langle [f, g], k \rangle$

2

Exercício 2.18. Derivar + -O cancelamento (2,40), + -re flexão (2,41) e + -fusion (2,42) de propriedade universal (2,65). Em seguida, obter o lei cambial (2,49) a partir da propriedade universal de produto (2,63) ou co-produto (2,65).

2

Exercício 2.19. Função $\text{coassocr} = [id + i1, Eu_2 \cdot Eu_2]$ é um testemunho de isomorfismo $(A + B) + C \sim A + (B + C)$ da esquerda para a direita. Calcule seu inverso coassoci resolvendo a equação

$$\left[\frac{x, [y, z]}{\overbrace{}^{\text{coassoc}}} \right] \cdot \text{coassocr} = id \quad (2,67)$$

para x, y e z .

2

Exercício 2.20. Deixe δ ser uma função do que você sabe que $\pi_1 \cdot \delta = id$ e $\pi_2 \cdot \delta = id$ aguarde. Mostram que, necessariamente, δ satisfaz a propriedade natural $(f \times f) \cdot \delta = \delta \cdot f$.

2

2.14 Guardas e McCarthy condicional de

A maioria das linguagens de programação funcional e notações atender pontuais expressões condicionais da forma

E se px então gx outro hx (2.68)

que avalia a gx em caso px detém e hx de outro modo, que é

{ $px \Rightarrow gx$

$\neg (px) \Rightarrow hx$

dado algum predicado Bool

$\text{UMA} \xrightarrow{\text{ao } p} \text{UMA}$

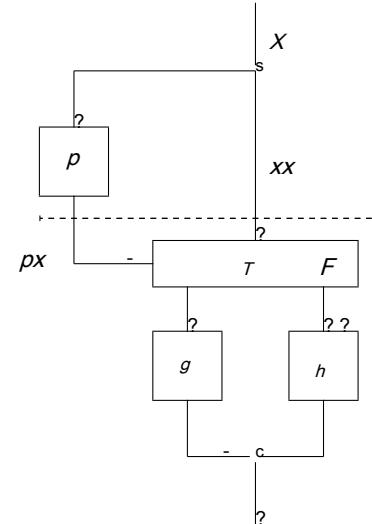
alguns “então” -função B

$\text{UMA} \xrightarrow{\text{ao } g} \text{UMA}$ e

alguns -função “else” B

$\text{UMA} \xrightarrow{\text{ao } h} \text{UMA}$

Pode (2.68) ser escrito no estilo pointfree?



O desenho acima é uma tentativa para expressar uma expressão condicional, tal como um -diagrama “bloco”. Em primeiro lugar, a entrada de X é copiado, a cópia esquerda sendo passado para predicar p obtendo-se o booleano px . Pode-se facilmente de f_i ne esta peça usando $cópia = <eu fiz>$.

A parte informal do diagrama representa a TF “Switch”: deve canalizar X para g em caso px muda o T -de saída, ou canal X para h de outra forma.

À primeira vista, este TF porta deve ser do tipo $B \times \text{UMA} \rightarrow \text{UMA} \times \text{UMA}$. Mas a saída não pode ser $\text{UMA} \times \text{UMA}$, Como f_i ou g agir em alternância, não em paralelo - deve sim ser $A + \text{UMA}$, caso em que o último passo é alcançado apenas através da execução [g, h].

Como é que um interruptor de nosso modelo baseado em produtos de partida de condicionais para um baseado em coproducto?

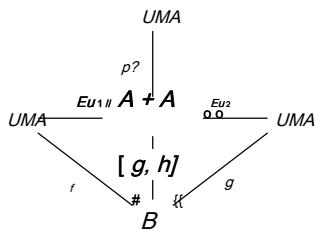
A observação importante é que o tipo $B \times UMA$ marcado pela linha tracejada no bloco-diagrama é isomorfo $A + UMA$, recordação (2,59). Ou seja, as informações capturadas pelo par $(px, x) \in B \times UMA$ pode ser convertido num único $y \in A + A$

sem perda de informação. Vamos de fato ne um novo combinador para isso, denotada $p ?$:

$$\begin{aligned} & \{ pa \Rightarrow Eu_1 \text{ uma} \\ (p?) a = & \neg (pa) \Rightarrow Eu_2 \text{ uma} \end{aligned} \quad (2,69)$$

Nós chamamos $A + A \xrightarrow{\alpha \circ p?} UMA$ uma *guarda*, ou melhor, o guarda associado a um determinado predicate Bool $\xrightarrow{\alpha \circ p} UMA$. Em certo sentido, guarda $p?$ é mais “informativa” do que p sozinha: ela fornece informações sobre o resultado do teste p em alguma entrada *uma*, codificado em termos das injecções coproduto (*Eu*₁ para *verdade* resultado e *Eu*₂ para *falso* resultado, respectivamente) sem perder a entrada *uma* em si.

Finalmente, a composição $[g, h] \cdot p ?$, representado no seguinte diagrama



tem (2,68) no sentido de pontual. É um combinador funcional conhecido denominado “McCarthy condicional”¹⁰ e usualmente representada pela expressão $p \rightarrow g, h$. No total, temos a definição

$$p \rightarrow g, h \underset{\text{def}}{=} [g, h] \cdot p ? \quad (2,70)$$

o que sugere que, para raciocinar sobre condicionais, pode-se procurar ajuda na álgebra de co-produtos. Na verdade, o seguinte fato,

$$f \cdot (p \rightarrow g, h) = p \rightarrow f \cdot g, f \cdot h \quad (2,71)$$

que nos referiremos como o *primeira lei de fusão condicional de McCarthy*¹¹, não é senão uma consequência imediata de -fusion + (2,42).

¹⁰ Depois John McCarthy, o cientista da computação da primeira definida-lo.

¹¹ Para o segundo ir para exercer 2,22.

40 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Vamos introduzir e des casos fi ne de predicado p contanto que eles são necessários. Uma suposição particularmente importante da nossa notação deve, no entanto, ser mencionado neste momento: vamos supor que, para cada tipo de dados UMA , o predicado de igualdade Bool

$$\frac{a =_{UMA} a}{UMA \times UMA} \text{ é definida de uma forma que garante três básicos}$$

Propriedades: reflexividade ($a =_{UMA} a$ para cada uma), transitividade ($a =_{UMA} b$ e $b =_{UMA} c$ implica $a =_{UMA} c$) e simetria ($a =_{UMA} b$ se $b =_{UMA} a$). Subscrito UMA in $=_{UMA}$ será descartado onde quer implícita no contexto.

em HASKELL programação, o predicado de igualdade para um tipo se torna disponível, declarando o tipo como uma instância da classe `Eq`, que exporta predicado igualdade (`(==)`). Isto não significa, no entanto, garantir os reflexiva, transitiva e simetria propriedades, que precisam ser provada por argumentos matemáticos dedicados.

Encerramos esta seção com uma ilustração de como *inteligente* álgebra pointfree pode ser no raciocínio sobre as funções que *um não é realmente de ne fi explicitamente*. Ele também mostra como relevante o *propriedades naturais* Estudou na seção 2.12 são. A questão é que a nossa definição de de um guarda (2,69) é pontual e provavelmente inadequados para provar fatos tais como, por exemplo,

$$p? \cdot F = (f + f) \cdot (p \cdot f) \quad (2,72)$$

Pensando melhor, em vez de “inventar” (2,69), podemos (e talvez deveria!) Têm de fi nida

$$\frac{\begin{array}{c} UMA \xrightarrow[p, \text{idéntica}]{} 2 \times UMA \\ \text{---} \\ p? \end{array}}{UMA \xrightarrow[4 \cdot 4]{\text{idéntica}} A + A} \quad (2,73)$$

que na verdade expressa bem de perto a nossa estratégia de mudança de produtos para co-produtos na definição de ($p?$). isomorfismo α (2.59) é o assunto do exercício 2.12. Será que precisamos de definir explicitamente? Talvez não: a partir de seu tipo,

$2 \times UMA \rightarrow A + A$, nós imediatamente inferir a sua propriedade natural (ou “livre”):

$$\alpha \cdot (\text{idéntica} \times f) = (F + f) \cdot \alpha \quad (2,74)$$

Acontece que este é o *conhecimento* precisamos de cerca de α a fim de provar (2,72), como o seguinte cálculo mostra:

$$\begin{aligned} & p? \cdot f \\ = & \{(2,73); \langle p, \text{idéntica} \rangle \cdot f = \langle p \cdot f, f \rangle\} \\ & \alpha \cdot \langle p \cdot f, f \rangle \\ = & \{\times - \text{absorção (2,27)}\} \end{aligned}$$

$$\begin{aligned}
 & a \cdot (\text{id} \times f) \cdot \langle p \cdot f, \text{id} \rangle \\
 = & \quad \{ \text{teorema livre } a \text{ (2.74)} \} \\
 & (f + f) \cdot a \cdot \langle p \cdot f, \text{id} \rangle \\
 = & \quad \{ \text{novamente (2.73); } \langle p, \text{id} \rangle \cdot f = \langle p \cdot f, f \rangle \} \\
 & (f + f) \cdot (p \cdot f) \\
 \end{aligned}$$

2

Outros exemplos desse tipo de raciocínio, com base em propriedades naturais (gratuito) de isomorfismos - e muitas vezes em “manobras”-los em torno de usar as leis (2.18,2.19) - será dada mais tarde neste livro.

Quanto menos um tem que escrever para resolver um problema, melhor. Uma economiza tempo e cérebro de um, aumentando a produtividade. Isso é muitas vezes chamado *elegância* quando se aplica um método fi c científica.
(Infelizmente, estar preparado para muita falta dela no software de engenharia de campo!)

Exercício 2.21. Prove que a seguinte igualdade entre duas expressões condicionais

$$\begin{aligned}
 & k(E \text{ se } px \text{ então } fx \text{ outro } hx, E \text{ se } px \text{ então } gx \text{ outro } ix) \\
 = & E \text{ se } px \text{ então } k(\lambda ap fx, \lambda ap gx) \text{ outro } k(hx, ix)
 \end{aligned}$$

detém, reescrevendo-a no estilo pointfree (usando combinator condicional do McCarthy) e aplicando a lei cambial (2.49), entre outros.

2

Exercício 2.22. provar a primeira lei de fusão condicional de McCarthy (2.71). Então, a partir de (2.70) e alojamento (2.72), inferir a segunda ocorrência dessa lei:

$$(p \rightarrow f, g) \cdot h = (p \cdot h) \rightarrow (f \cdot h), (g \cdot h) \tag{2.75}$$

2

Exercício 2.23. Provar que a propriedade

$$\langle f, (p \rightarrow q, h) \rangle = p \rightarrow \langle f, q \rangle, \langle f, h \rangle \tag{2.76}$$

e seu corolário

$$(p \rightarrow g; h) \times f = p \cdot \pi_1 \rightarrow g \times f, h \times f \quad (2.77)$$

manter, assumindo o fato básico:

$$p \rightarrow f, f = f \quad (2.78)$$

2

2.15 funções colagem que não compõem - ponentials ex-

Agora que fizemos a distinção entre os pointfree e pontuais anotações funcionais razoavelmente claras, é instrutivo para revisitar seção 2.2 e identificar *aplicação funcional* como a “ponte” entre os pointfree e pontuais mundos. No entanto, devemos dizer “uma ponte” em vez de “a ponte”, no presente seção, enriquecer essa interface com outro “ponte” que é muito relevante para a programação.

Suponha que nos é dada a tarefa de combinar duas funções, um binário B
 $\text{ao } f \text{ } C \times \text{UMA}$
 eo outro unário: D $\text{ao } g \text{ } \text{UMA}$. É claro que nenhuma das combinações $f \cdot g$;
 $\langle f, g \rangle$ ou $[f, g]$ é bem-digitado. Assim, f e g não podem ser colocados juntos diretamente - eles exigem alguma interface extra.

Observe que $\langle f, g \rangle$ seria bem definida no caso do C componente de f ’s domínio poderia ser de alguma forma ‘ignorado’. Suponha-se, na verdade, que em algum contexto particular, o argumento primeiro de f passa a ser “irrelevante”, ou para ser congelado para alguns $c \in C$. É fácil derivar uma nova função

$$\begin{aligned} f_c &: \text{UMA} \rightarrow B \text{ } f_c \text{uma def} \\ &= f(c, a) \end{aligned}$$

de f que combina muito bem com g através da *Dividido* combinator: $\langle f, g \rangle$ é welde definido e carrega tipo $B \times D \leftarrow \text{UMA}$. Por exemplo, suponha que $C = A$ e f é o predicado de igualdade $=$ on UMA . Então Bool

$\text{ao } =_c \text{ } \text{UMA}$ é o “igual a c ” predicado em UMA valores:

$$=_c \text{uma def} = a = c \quad (2.79)$$

Como outro exemplo, a função de recolha *duas vezes* (2.3) que pode ser definida como x_2 usando a nova notação.

No entanto, precisamos ser mais cuidadosos sobre o que se entende por f_c . Tal como aplicação funcional, expressão f_c interfaces a pointfree e os níveis pontuais - que envolve uma função (f) e um valor (c). Mas pelo $B \xrightarrow{\text{ap}} C \times UMA$, lá é uma distinção importante entre f_c e f_c - enquanto o primeiro denota um valor de tipo B , ou seja $f_c \in B$, f_c indica uma função do tipo $B \leftarrow UMA$. Vamos dizer que $f_c \in B_{UMA}$ através da introdução de uma nova construção de tipo de dados que iremos referir como o *exponencial*:

$$B_{UMA}^{\text{def}} = \{ g / g: B \leftarrow UMA \} \quad (2.80)$$

Há fortes razões para adoptar o B_{UMA} notação, em detrimento dos mais óbvios $B \leftarrow UMA$ ou $UMA \rightarrow B$ alternativas, como veremos em breve.

o B_{UMA} tipo de dados exponencial é portanto habitado por funções a partir UMA para B , isto é, a declaração funcional $g: B \leftarrow UMA$ significa o mesmo que $g \in B_{UMA}$. E o que queremos funções para? Nós queremos aplicá-las. Por isso, é natural para introduzir o *Aplique* operador

$$\begin{aligned} AP: B &\xrightarrow{\text{ap}} B_{UMA} \times UMA \\ AP(f, a)_{\text{def}} &= fa \end{aligned} \quad (2.81)$$

que se aplica uma função fa um argumento *uma*.

Voltar a função binária genérica $B \xrightarrow{\text{f}} C \times UMA$, vamos agora pensar no OP peração que, para cada $c \in C$, produz $f_c \in B_{UMA}$. Isso pode ser considerado como uma função da assinatura $B_{UMA} \leftarrow C$ que exprime f como uma espécie de C -família indexada de funções de assinatura $B \leftarrow UMA$. Iremos designar uma tal função por f (ler "fComo" "ftransposta"). Intuitivamente, nós queremos f e fa ser relacionados entre si pela seguinte propriedade:

$$f(c, a) = (fc) \text{ uma} \quad (2.82)$$

Dado c e *uma*, ambas as expressões designam o mesmo valor. Mas, em certo sentido, f é mais tolerante do que f : enquanto o último é binário e requer *ambos* argumentos (c, a) para se tornar disponível antes da aplicação, o primeiro é o prazer de ser fornecido com c primeiro e com *uma*. Mais tarde, se realmente exigido pelo processo de avaliação.

similarmente a $UMA \times B$ e $A + B$, exponencial B_{UMA} envolve uma propriedade universal,

$$k = f \Leftrightarrow f = ap \cdot (k \times \text{idéntidade}) \quad (2.83)$$

44 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

a partir do qual as leis para o cancelamento, re fluxão e fusão pode ser derivada:

Exponenciais cancelamento:

$$\begin{array}{ccc}
 \begin{array}{c} B_{UMA} \\ \text{---} \\ f \\ \text{---} \\ C \end{array} &
 \begin{array}{c} B_{UMA} \times UMA_{ap} \\ \text{---} \\ f \times \text{id} \\ \text{---} \\ C \times UMA \end{array} &
 \overline{f = ap \cdot (f \times \text{id})} \\
 & & (2.84)
 \end{array}$$

Exponenciais re fluxão:

$$\begin{array}{ccc}
 \begin{array}{c} B_{UMA} \\ \text{---} \\ BA \\ \text{---} \\ B_{Ajuda} \end{array} &
 \begin{array}{c} B_{UMA} \times UMA_{ap} \\ \text{---} \\ \text{id} \times \text{id} \\ \text{---} \\ B_{UMA} \times UMA \end{array} &
 \overline{ap = id_{B_{UMA}}} \\
 & & (2.85)
 \end{array}$$

Exponenciais fusão:

$$\begin{array}{ccc}
 \begin{array}{c} B_{UMA} \\ \text{---} \\ g \\ \text{---} \\ C_{UMA} \\ \text{---} \\ f \\ \text{---} \\ D \end{array} &
 \begin{array}{c} B_{UMA} \times UMA_{ap} \\ \text{---} \\ g \times \text{id} \\ \text{---} \\ C \times UMA \\ \text{---} \\ f \times \text{id} \\ \text{---} \\ D \times UMA \end{array} &
 \overline{g \cdot (f \times ID) = g \cdot f} \\
 & & (2.86)
 \end{array}$$

Note-se que a lei de cancelamento é nula, mas fato (2.82) escrito no estilo pointfree.

Existe uma lei de absorção para exponenciais? A resposta é afirmativa mas primeiro é preciso introduzir um novo combinador funcional que surge como a transposta de $f \cdot ap$ no diagrama a seguir:

$$\begin{array}{ccc}
 \begin{array}{c} D_{UMA} \times UMA_{ap} \\ \text{---} \\ f \cdot ap \times \text{id} \\ \text{---} \\ B_{UMA} \times UMA \end{array} &
 \begin{array}{c} D_{UMA} \\ \text{---} \\ f \\ \text{---} \\ D \end{array} &
 \overline{D_{UMA} \times UMA_{ap} \xrightarrow{\quad\quad\quad} D_{UMA}}
 \end{array}$$

Vamos denotar isso, f_{UMA} e seu tipo de regra é a seguinte:

$$\begin{array}{c}
 \begin{array}{c} C \xrightarrow{f} B \\ \hline C_{UMA} \xrightarrow{ap} B_{Afuma} \end{array}
 \end{array}$$

Pode ser demonstrado que, uma vez UMA e C

$$\begin{array}{ccc} & \xrightarrow{\quad f \quad} & B \\ \text{aceita alguma função de entrada } B & \xrightarrow{\quad g \quad} & UMA \end{array}$$

são fixos, f_{UMA} é a função que resulta de aplicar f a UMA como argumento e produz função $f \cdot g$. Como resultado (ver exercício 2.39), assim f_{UMA} é a “compor com f ” Combinador funcional:

$$(f_A)g \text{ def } = f \cdot g \quad (2.87)$$

Agora estamos prontos para entender as leis que se seguem:

Exponenciais absorção:

$$\begin{array}{c} D_{UMA} \\ | \\ B_{f_{UMA}} \\ | \\ g \\ \hline C \end{array} \quad \begin{array}{c} D_{UMA} \times UMA \\ | \\ f_{UMA} \times \text{id} \\ | \\ B_{UMA} \times UMA \\ | \\ g \times \text{id} \\ \hline C \times UMA \end{array} \quad \begin{array}{c} f \cdot g = f_{UMA} \cdot g \\ | \\ f \\ | \\ B \\ \hline g \\ | \\ C \times UMA \end{array} \quad (2.88)$$

(Note-se como, a partir deste, também temos $f_A = f \cdot ap$).

Exponenciais-functor:

$$(g \cdot h)_A = g_{UMA} \cdot h_{UMA} \quad (2.89)$$

Exponenciais-functor-id:

$$\text{id}_{UMA} = \text{id}_A \quad (2.90)$$

Por que a notação exponencial. Para concluir esta seção, precisamos explicar por que adotamos a aparentemente esotérico B_{UMA} notação para a “função de UMA para B ” tipo de dados. Esta é a oportunidade de relacionar o que vimos acima com duas funções (de ordem superior) que são muito familiares aos programadores funcionais. Na HASKELL Prelude são definidos assim:

```
Curry :: ((uma, b) -> c) -> (uma -> b -> c)
Curry Fab = f(a, b)

uncurry :: (uma -> b -> c) -> (uma, b) -> c
uncurry f(a, b) = Fab
```

Em nossa notação para tipos, Curry funções mapas no espaço função $C_{UMA \times B}$ para funções (em C_B) e uncurry funções mapas do último espaço funcional para o primeiro.

Vamos calcular o significado de Curry removendo as variáveis de sua definição:

$$\begin{aligned}
 & \{ \underline{\quad} \} \overset{g}{\underline{\quad}} \{ \\
 & (\text{c}\underset{\downarrow}{\text{voc}}\text{e}_\text{ my } f \ a) b = f(a, b) \\
 & \} \{ \underset{\overline{f}}{\quad} \\
 & \equiv \{ \text{introduzir } g \} \\
 & GB = f(a, b) \\
 & \equiv \{ \text{Desde a } GB = p(g, b) \text{ (2.81)} \} \\
 & AP(g, b) = f(a, b) \\
 & \equiv \{ g = fa; \text{natural-identidade} \} \\
 & ap(\overline{f} um, ID b) = f(a, b) \\
 & \equiv \{ \text{produto de funções: } (f \times g)(x, y) = (fx, Gy) \} \\
 & AP((\overline{f} \times id)(a, b)) = f(a, b) \\
 & \equiv \{ \text{composição} \} \\
 & (ap \cdot (\overline{f} \times id))(a, b) = f(a, b) \\
 & \equiv \{ \text{extensionalidade (2.5), ou seja, a remoção de pontos } uma \text{ e } b \} \\
 & ap \cdot (\overline{f} \times ID) = f
 \end{aligned}$$

Do exposto infere-se que a definição de Curry é uma re-affirmação do direito de cancelamento (2.84). Isso é,

$$\text{Curry } f \stackrel{\text{def}}{=} f \quad \text{---} \tag{2.91}$$

e Curry é transposição em HASKELL-falar.¹²

Em seguida, fazer o mesmo para a definição de uncurry:

$$\begin{array}{c}
 \text{voc}\text{e}_\text{nc ur ry } f \ (uma, b) = Fab \\
 \} \{ \underset{k}{\quad}
 \end{array}$$

¹² Esta terminologia amplamente adotado em outras linguagens funcionais.

$$\begin{aligned}
 &\equiv \{ \text{introduzir } k; \text{ lado esquerdo como calculado acima} \} \\
 k(a, b) &= (ap \cdot (f \times \text{idéntidade})) (uma, b) \\
 &\equiv \{ \text{extensionalidade (2.5)} \} \\
 k &= ap \cdot (f \times \text{idéntidade}) \\
 &\equiv \{ \text{propriedade universal (2.83)} \} \\
 F &= k \\
 &\equiv \{ \text{expandir } k \} \\
 f &= \underline{\underline{uncurry}} f
 \end{aligned}$$

Concluimos que $\underline{\underline{\text{uncurry}}}$ é o inverso da transposição, que é, de Curry. Vamos usar a abreviatura \hat{f} para $\underline{\underline{\text{uncurry}}} f$, em que a igualdade é acima escrita

$\hat{f} = f$. Ele também pode ser verificado que $f = \hat{f}$ também detém, instanciar k acima por \hat{f} .

$$\hat{T} = ap \cdot (f \times \text{idéntidade})$$

$$\begin{aligned}
 &\equiv \{ \text{cancelamento (2.84)} \} \\
 \hat{\hat{f}} &= f
 \end{aligned}$$

2

assim $\underline{\underline{\text{uncurry}}}$ - ou seja, $(\hat{_})$ e Curry - ou seja, $(\)$ - são inversas entre si,

$$g = f \Leftrightarrow \hat{g} = f \tag{2.92}$$

levando a isomorfismo

$$UMA \rightarrow C_B \sim = UMA \times B \rightarrow C$$

que também pode ser escrito como

$$\begin{array}{ccc}
 & \xrightarrow{\text{uncurry}} & * \\
 (C_{BA} & \sim = & C_{UMA \times B} \\
 & \xrightarrow{\text{Curry}} &
 \end{array} \tag{2.93}$$

decorado com as testemunhas correspondentes.¹³

¹³ Escrita f/\hat{f} ou Curry f (resp. $\underline{\underline{\text{uncurry}}} f$) é uma questão de gosto: o último são mais na tradição de programação funcional e ajuda quando as funções têm de ser chamado; o ex-economizar tinta nas expressões algébricas e cálculos.

48 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Isomorfismo (2.93) é o cerne da teoria e prática da programação funcional. Assemelha-se claramente uma igualdade bem conhecida sobre exponenciais numéricos, $b^{c \times a} = (b^a)^c$. Além disso, outros fatos conhecidos sobre exponenciais numéricos, por exemplo $\text{uma } b^{a+c} = \text{uma } b \times \text{uma } c \text{ ou } (b \times c)_a = b_{\text{uma}} \times c_{\text{uma}}$ também achar o seu homólogo em exponenciais funcionais. A contrapartida do primeiro,

$$\text{UMA } b^{a+c} \sim = \text{UMA } b \times \text{UMA } c \quad (2.94)$$

decorre da singularidade do *ou* combinação: cada par de funções $(f, g) \in \text{UMA } b \times \text{UMA } c$ leva a uma função singular $[f, g] \in \text{UMA } b + c$ e vice-versa, em cada função $\text{UMA } b + c$ é o *ou* de alguma função no $\text{UMA } b$ e de outro em $\text{UMA } c$.

O exponenciais função homólogo do segundo fato sobre exponenciais numéricos acima é

$$(B \times C)_{\text{uma}} \sim = B_{\text{uma}} \times C_{\text{uma}} \quad (2.95)$$

Isso pode ser justificada por um argumento similar sobre a singularidade do *Dividido* combinator $\langle f, g \rangle$.

E quanto a outros fatos válido para exponenciais numéricos, como $\text{uma}_0 = 1$ e $1_a = 1$? O leitor é convidado a voltar para a seção 2.11 e recordar o que 0 e 1 significa como tipos de dados: o vazio (void) e tipos de dados únicos, respectivamente. Nossa contrapartida $\text{uma}_0 = 1$ então é

$$\text{UMA}_0 \sim = 1 \quad (2.96)$$

Onde UMA_0 indica o conjunto de todas as funções do conjunto vazio para alguns UMA . O que (2.96) significa? Ele simplesmente nos diz que não é apenas uma função de tal conjunto - o mapeamento função vazia “não” valor. Este fato confirma a nossa escolha de notação uma vez (compare com $\text{uma}_0 = 1$ num contexto numérico).

Em seguida, podemos nos perguntar sobre os fatos

$$1_{\text{UMA}} \sim = 1 \quad (2.97)$$

$$\text{UMA}_1 \sim = \text{UMA} \quad (2.98)$$

que são os homólogos funcionais de exponenciação $1_a = 1$ e $\text{uma}_1 = \text{uma}$.

Fato (2.97) é válida: isso significa que há apenas mapeamento uma função UMA a um conjunto singleton $\{c\}$ - a função constante c . Não há espaço para outra função em 1_{UMA} porque só c está disponível como valor de saída. Nossa denotação padrão para uma função tão único é dada por (2.58).

2.15. GLUING FUNCTIONS WHICH DONOT COMPOSE-EXPONENTIALS 49

Fato (2,98) também é válida: todas as funções em UMA_1 são (valor único) funções constantes e há tantas funções constantes, de tal conjunto como existem elementos em UMA . Essas funções são muitas vezes chamado (sumário) "pontos" por causa do 1-to-1 mapeamento entre UMA_1 e os elementos (pontos) nas UMA .

Exercício 2.24. Relacionar o isomorfismo envolvendo tipo elementar genérico 2

$$UMA \times UMA \sim = UMA_2 \quad (2,99)$$

para a expressão $| f \rightarrow (f \text{ É verdade}, f \text{ False})$ escrito em HASKELL sintaxe.

2

Exercício 2.25. Considere as testemunhas de isomorfismo (2,95)

$$\begin{array}{ccc} (B \times C) UMA & \xrightarrow{\text{desemparelhar}} & + \\ \text{k.k} & \sim = & B UMA \times C UMA \\ & \xrightarrow{\text{par}} & \end{array}$$

definido por:

$$\begin{aligned} \text{par } (f, g) &= \langle f, g \rangle \\ \text{desemparelhar } k &= (\pi_1 \cdot k, \pi_2 \cdot k) \end{aligned}$$

Mostre que $\text{par} \cdot \text{desemparelhar} = \text{idéntidade}$ e $\text{desemparelhar} \cdot \text{par} = \text{idéntidade aguarde}$.

2

Exercício 2.26. Mostre que a seguinte igualdade

$$\overline{fa} = f \cdot \langle \underline{uma}, \underline{\text{idéntidade}} \rangle \quad (2,100)$$

detém.

2

Exercício 2.27. Considerando $\alpha = [i_1, Eu_2, (a)]$ inferir o principal (mais geral) TIPO DE α

e representá-lo em um diagrama; (B) α é um isomorfismo bem conhecido - dizer qual inferindo seu tipo.

Exercício 2.28. Provar a igualdade $g = g \cdot \pi_2$ sabendo que

$$\overline{\pi_2} = \underline{\text{idéntidade}}$$

(2.101)

detém.

2.16 produtos finitária e coprodutos

Na seção 2.8, foi sugerido que o produto poderia ser considerado como a abstração trás **primitivas-estruturação de dados, tais como struct em C ou registro em P Ascal**.

De modo semelhante, os co-produtos foram sugeridas na secção 2.9 homólogos como sumário de uniões de C ou P Ascal registros de variante. Para um nite fi **UMA**, exponencial **B** **UMA** Pode ser realizado como um **ordem** em qualquer um desses idiomas.

Essas analogias são capturados na tabela 2.1.

Da mesma forma C struct areia União s podem conter infinitamente muitas entradas, como pode P Ascal (variante) registos, produtos **UMA** × **B** estende-se ao produto fi nitary **UMA** ₁ × ... × **UMA** _n, para $n \in \mathbb{N}$, também denotado por $\prod_{n=1}^n \text{UMA}_{Ei}$, para que, como muitos projecções π_{Ei} estão associadas como o número n de factores envolvidos. Claro, *splits* tornar-se n -ary bem

$$\langle f_1, \dots, f_n \rangle : \text{UMA}_1 \times \dots \times \text{UMA}_n \leftarrow B$$

para $f_{Ei} : \text{UMA}_{Ei} \leftarrow B$, $i = 1, n$.

Duplicamente, coproduct **A** + **B** é extensível à soma fi nitary **UMA** ₁ + ... + **UMA** _n, para $n \in \mathbb{N}$, também denotado por $\Sigma_n \text{UMA}_i$, para que, como muitos injecções Ei são atribuídos como o número n de termos envolvido. Similarmente, *eithers* tornar-se n -ary

$$[f_1, \dots, f_n : \text{UMA}_1 + \dots + \text{UMA}_n \rightarrow B$$

para $f_{Ei} : B \leftarrow \text{UMA}_{Ei}$, $i = 1, n$.

notação Abstract	P Ascal	C / C ++	Descrição
$UMA \times B$	registro P: A; S: B final;	struct { Um primeiro; B segundo; };	registros
$A + B$	registro tag caso: número inteiro de x = 1: (P: A); 2: (S: B) fim;	struct { int tag; /* 1,2 */ union { Um IFA; B IFB; } dados; };	registros variantes
B_{UMA}	matriz [A] de B	BAJ	Arrays
$1 + UMA$	UMA	UMA *...	ponteiros

Tabela 2.1: Notação Resumo versus os dados-estruturas de linguagem de programação.

Tipo de dados n

Avançar depois 2, podemos pensar 3 como representando a classe abstrata de todos os tipos de dados que contêm exatamente três elementos. Generalizando, podemos pensar n como representando a classe abstrata de todos os tipos de dados que contêm exatamente n elementos. Claro, segmento inicial n será nesta classe abstrata. (Chamar (2,17), por exemplo: tanto dia da semana e 7 são abstratamente representado por 7.) Assim sendo,

$$n \sim = 1 \pm \dots \pm 1 \{ \dots \}_n$$

e

$$\underbrace{UMA \times \{ \dots \}}_n \sim = UMA_n \quad (2,102)$$

$$\underbrace{UMA \pm \{ \dots \}}_n \sim = n \times UMA \quad (2,103)$$

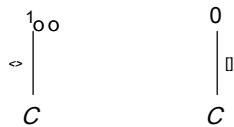
aguarde.

Exercício 2.29. Com base na Tabela 2.1, codificar undistr (2.51) em C ou PASCAL. Compare seu código com o HASKELL pointfree e pontuais equivalentes.

2.17 tipos de dados inicial e terminal

Todas as propriedades estudadas para binário *splits* e binário *either*s estender-se ao caso fi nitary. Para a situação particular $n = 1$, nós teremos $\langle f \rangle = [f] f = e$

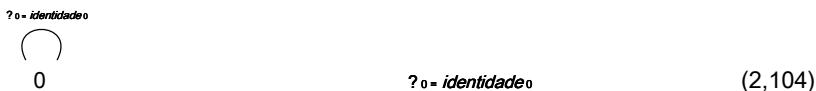
$\pi_1 = Eu_1 = \text{identidade}$, claro. Para a situação particular $n = 0$, fi produtos nitary “degenerada” para 1 e co-produtos fi nitary “degenerada” para 0. Assim diagramas (2,23) e (2,38) são reduzidos a



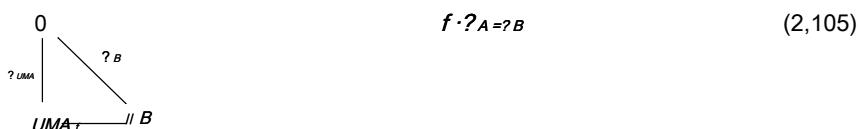
A notação padrão para o vazio *Dividido* $\langle \rangle$ é $!c$, em que o índice C pode ser omitido se implícito no contexto. By the way, esta é precisamente a única função no 1 c , recordação (2,97) e (2,58). Dualmente, a notação padrão para o vazio *ou* $[\]$ é $?c$, em que o índice C também pode ser omitido. By the way, esta é precisamente a única função no C_0 , Sensibilidade (2,96).

Em resumo, podemos pensar 0 e 1 como, em certo sentido, os “extremos” de todo o espectro tipo de dados. Por esta razão, eles são chamados *inicial* e *terminal*, respectivamente. Concluímos este assunto com a apresentação de suas principais propriedades que, como já dissemos, são instâncias de propriedades já dissemos para produtos e co-produtos.

tipo de dados inicial de reflexão:



fusão tipo de dados inicial:



tipo de dados do terminal de reflexão:

$$\begin{array}{ccc} !_1 = \text{idéntidade}_1 & & \\ \text{1} & & \\ & & !_1 = \text{idéntidade}_1 \\ & & (2,106) \end{array}$$

fusão tipo de dados de terminal:

$$\begin{array}{ccc} 1 \circ & & !_A \cdot f = !_B \\ |_!_A & & \\ & & B \\ & & f \\ 1 \circ & & \end{array} \quad (2,107)$$

Exercício 2.30. particularizar o lei cambial (2.49) para produtos vazio e co-produtos vazios, ou seja 1 e 0.

2

2,18 somas e produtos em HASKELL

Concluímos deste capítulo com uma análise da principal disponível primitivo em HASKELL para a criação de tipos de dados: o dados declaração. Suponha que nós declaramos

dados CostumerId = P Int | Int CC

o que significa dizer que, por alguma empresa, um cliente é identificado quer pelo seu número de passaporte ou pelo seu número de cartão de crédito, se houver. O que é que este pedaço de sintaxe precisamente significa?

Se inquirir o HASKELL intérprete sobre o que ele sabe sobre CostumerId,
a resposta irá conter as seguintes informações:

```
Principal>: i CostumerId
- - Tipo de dados construtor
CostumerId

- - construtores: P :: Int -> CostumerId
CC :: Int -> CostumerId
```

54 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

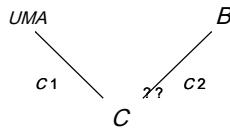
Em geral, vamos UMA e B ser de dois tipos de dados conhecidos. via declaração

$$\text{dados de } C = C1 \ A \mid C2 \ B \quad (2,108)$$

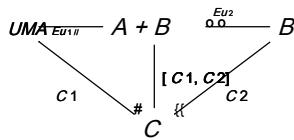
obtém-se fromH ASKELL um novo tipo de dados C equipado com construtores C

$$\xrightarrow{\alpha_0^{C1}} UMA$$

e $C \xrightarrow{\alpha_0^{C2}} B$, na verdade, os únicos disponíveis para a construção de valores de C :



Este diagrama leva a um exemplo óbvio de diagrama co-produto (2,38),

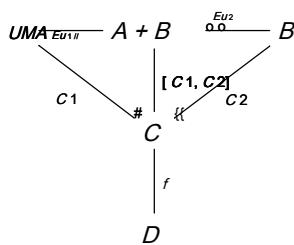


descreve que um dados declaração em H ASKELL significa que o *ou* de seus construtores.

Porque não há outros meios para construir C dados, segue-se que C é isomorfo a $A + B$. Assim [C 1, C 2] tem uma inversa, digamos inv , que é tal que

$inv \cdot [C1, C2] = \text{idéntidade}$. Como calculamos inv ? Vamos primeiro pensar na situação genérica de uma função D

$$\xrightarrow{\alpha_0^f} C \quad \text{observa que tipo de dados } C:$$



Esta é uma oportunidade para + - fusão (2,42), segundo o qual obtemos

$$f \cdot [C1, C2] = [f \cdot C1, f \cdot C2]$$

Portanto, a observação será completamente descrito desde que explicar como f comporta-se em relação à $C1$ - cf. $f \cdot C1$ - e com respeito ao $C2$ - cf. $f \cdot C2$.

Isto é o que está por trás da típica *indutivo* estrutura de ponto a ponto f , que será feita de duas e apenas duas cláusulas:

$$\begin{aligned} F: C \rightarrow D \\ f(C1\ a) = . \\ .. f(C2\ b) = . . . \end{aligned}$$

Vamos usar isso em calcular o inverso inv do $[C1, C2]$:

$$\begin{aligned} inv \cdot [C1, C2] &= \text{idéntidade} \\ &\equiv \{ \text{por } + - \text{ fusão (2.42)} \} \\ [inv \cdot C1, inv \cdot C2] &= \text{idéntidade} \\ &\equiv \{ \text{por } + - \text{ re flexão (2.41)} \} \\ [inv \cdot C1, inv \cdot C2] &= [Eu1, Eu2] \\ &\equiv \{ \text{ou igualdade estrutural (2.66)} \} \\ inv \cdot C1 = Eu1 \wedge inv \cdot C2 = Eu2 & \end{aligned}$$

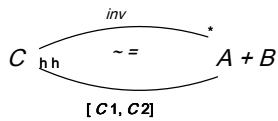
Assim sendo:

$$\begin{aligned} INV: C \rightarrow Um\ inv + B(C1\ a) \\ = i_1\ um\ inv(C2\ b) = i_2\ b \end{aligned}$$

Em suma, $C1$ é um “renomear” da injeção $Eu1$, $C2$ é um “renomear” da injeção $Eu2$ e C é “rebatizado” réplica $A + B$:

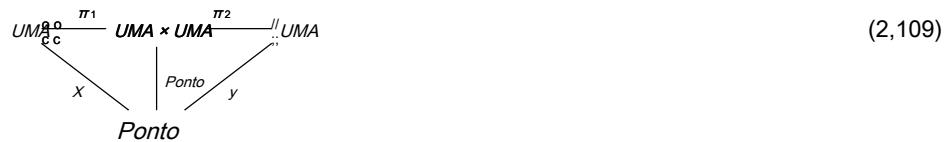
$$C \xrightarrow{\alpha_0} [C1, C2] \longrightarrow A + B$$

$[C1, C2]$ é chamado de *álgebra* de tipo de dados C e o seu inverso inv é chamado de *coálgebra* do C . A álgebra contém os construtores de $C1$ e $C2$ do tipo C , isto é, ela é usada para “construir” C -valores. Na direcção oposta, o co-álgebra inv nos permite “destruir” ou observar valores de C :

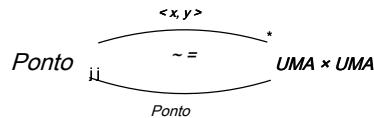


56 Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO

Álgebra / coálgebra também surgem sobre tipos de dados de produtos. Por exemplo, suponhamos que se deseja descrever o tipo de dados *Ponto* habitado por pares (X_0, y_0, X_1, y_1 etc. de coordenadas cartesianas de um determinado tipo, digamos *UMA*. Apesar *UMA* \times *UMA* equipado com projeções π_1, π_2 “É” um tal tipo de dados, pode-se estar interessado em uma réplica adequadamente chamado de *UMA* \times *UMA* nos quais os pontos são construídas explicitamente por alguns construtor (digamos *Ponto*) e observado por seletores dedicados (digamos *X* e *y*):



Este sobe uma álgebra (*Ponto*) e um coálgebra ($\langle x, y \rangle$) para tipo de dados *Ponto*:



em HASKELL se escreve

`Ponto de dados a = Ponto {x ::, uma y :: a}`

mas ser avisado que HASKELL entrega Ponto na forma de avaliar num:

`Ponto :: a -> a -> ponto um`

Finalmente, qual é a -equivalente “ponteiro” em HASKELL? Isto corresponde a $A = 1$ em (2,108) e para o seguinte HASKELL declaração:

`dados de C = C1 () | C2 B`

Note-se que HASKELL permite uma alternativa orientada a programação mais, neste caso, no qual o tipo de unidade () é eliminada:

`dados de C = C1 | C2 B`

A diferença é que aqui C1 indica um habitante *C* (e assim por uma cláusula $f(C1 a) = \dots$ é reescrito para $f C1 = \dots$) enquanto acima C1 indica uma função (constante) $C \xrightarrow{\text{a} \in C1} 1$. Isomorfismo (2,98) ajuda na comparação destes dois situa- alternativa ções.

2,19 exercícios

Exercício 2.31. Deixe UMA e B ser de dois tipos de dados disjuntos, isto é, $UMA \cap B = \emptyset$ detém. Mostram que isomorfismo

$$UMA \cup B \sim = A + B$$

(2,110)

detém. dica: de fina $UMA \cup B$ $\xrightarrow{\text{0.0 } EU} A + B$ Como $i = \text{emb } UMA$, $\text{emb } a$ para $\text{emb } UMA a = a$ e $\text{emb } b = b$, e encontrar o seu inverso. By the way, por que não definimos Eu tão simples quanto Eu é? identidade UMA , identidade B ?

2

Exercício 2.32. Sabendo que uma determinada função xr satisfaz propriedade

$$xr \cdot \langle\langle f, g \rangle, h \rangle = \langle\langle f, h \rangle, g \rangle$$

(2,111)

para todos f , g e h , derivar de (2.111), a definição de xr :

$$xr = \langle \pi_1 \times id, \pi_2 \cdot \pi_1 \rangle$$

(2,112)

2

Exercício 2.33. Deixe distr (ler: 'Distribuir direito') ser o bijection que testemunhas isomorfismo $UMA \times (B + C) \sim = UMA \times B + A \times C$. Preencha o "... No diagrama que se segue de modo a que ele descreve bijeção distr (vermelho: 'Distribuir esquerda') que isomorfismo testemunhas $(B + C) \times UMA \sim = B \times A + C \times UMA$:

$$(B + C) \times UMA \xrightarrow{\text{troca } II} \dots \xrightarrow{\text{distr } II} \dots \xrightarrow{\text{distl}} II B \times A + C \times UMA$$

2

Exercício 2.34. No contexto do exercício 2.33, provar

$$[g, h] \times f = [g \times f, h \times f] \cdot \text{distl}$$

(2,113)

sabendo que

$$f \times [g, h] = [f \times g, f \times h] \cdot \text{distr} \quad (2.114)$$

detém.

2

Exercício 2.35. A lei aritmética $(a + b)(c + d) = (ac + ad) + (+ Bc bd)$ corresponde ao isomorfismo

$$(A + B) \times (C + D) \xrightarrow{\text{kk}} h = [f_{11} \times Eu_1, Eu_1 \times Eu_2, f_{12} \times Eu_1, Eu_2 \times Eu_2] \sim= (UMA \times C + A \times D) + (B \times C + B \times D)$$

De propriedade universal (2.65) inferir o seguinte definição de função h , escrito na sintaxe Haskell:

$h(\text{Esquerda } (\text{Esquerda } (a, c))) = (\text{Esquerda } a, c \text{ Esquerda } h(\text{para } a \text{ esquerda } (\text{direita } (a, d)))) = (\text{Esquerda } \text{um}, \text{direito } d) h(\text{direita } (\text{esquerda } (b, c))) = (b \text{ direito}, \text{esquerdo } c) h(\text{para } a \text{ direita } (\text{Right } (b, d))) = (\text{direito } b, d \text{ direito})$

2

Exercício 2.36. Cada programador C sabe que um struct de ponteiros

$$(A + 1) \times (B + 1)$$

oferece um tipo de dados que representa ao mesmo tempo o produto $UMA \times B$ (struct) e coproducto $A + B$ (União), alternativamente. Expresso em notação pointfree os isomorfismos Eu_1 para Eu_5 do

$$\begin{array}{ccc} (A + 1) \times (B + 1) & \xrightarrow{\alpha\alpha} & ((A + 1) \times B) + ((A + 1) \times 1) \\ & & \downarrow Eu_1 \\ & & (UMA \times B + 1 \times B) + (A \times 1 + 1 \times 1) \\ & & \downarrow Eu_2 \\ & & (UMA \times B B +) + (A + 1) \\ & & \downarrow Eu_3 \\ (UMA \times B + (B + A)) + 1 & \xrightarrow{\beta\beta} & // UMA \times B + (B + (A + 1)) \\ & & \downarrow Eu_4 \\ & & Eu_5 \end{array}$$

que testemunha a observação acima.

2

Exercício 2.37. Prove a seguinte propriedade de condicionais McCarthy:

$$p \rightarrow f \cdot g, h \cdot k = [f, h] \cdot (p \rightarrow Eu_1 \cdot G, I_2 \cdot k) \quad (2,115)$$

2

Exercício 2.38. Supondo que o fato

$$(p? + P?) \cdot p? = (I_1 + Eu_2) \cdot p? \quad (2,116)$$

mostram que condicionais aninhados pode ser simplificada:

$$p \rightarrow (p \rightarrow f, g), (p \rightarrow h, k) = P \rightarrow F, K \quad (2,117)$$

2

Exercício 2.39. Mostre que $(f \cdot ap) g = \overline{f \cdot g}$ detém, cf. (2,87).

2

Exercício 2.40. Considere o maior fim-isomorfismo giro definida da seguinte forma:

$$\begin{array}{ccccccccc} (C_{BA}) & \sim & = C_{UMA \times B} & \sim & = C_{B \times UMA} & \sim & = & (C_A)_B \\ f & \longrightarrow & \hat{f} & \longrightarrow & f. \text{ troca} & \longrightarrow & f. \text{ Swap} & = \text{aleta } f \end{array}$$

Mostre que virar $f \circ y = f \circ x$.

2

Exercício 2.41. Deixe $C \text{ const} // C_{\text{UMA}}$ ser a função de exercício 2.2, ou seja, $\text{const } c = c_{\text{UMA}}$.

Fato que é expresso pelo seguinte diagrama apresentando const ?



Escrevê-lo a nível ponto e descrevê-lo por suas próprias palavras.

2

Exercício 2.42. Mostre que $\pi_2 \cdot f = \pi_2$ vale para todos os f . portanto π_2 é uma função constante

- qual?

2

Exercício 2.43. Estabelecer a diferença entre as duas declarações seguintes em

HASKELL,

dados D = D1 A | D2 BC dados E = E1 A |
E2 (B, C)

para A, B e C Quaisquer três prede fi nidas tipos. Está D e E isomorphic? Se assim for, você pode especificar e codificar o isomorfismo correspondente?

2

2.20 Notas Bibliografia

Algumas décadas atrás John Backus ler, em sua palestra Turing Award, um papel revolucionário [3]. Este artigo proclamou linguagens de programação orientada a comandos convencionais obsoletos por causa de sua eficiência fi INEF decorrente da retenção, a um alto nível, o chamado “acesso gargalo de memória” da computação subjacente

modelo - o conhecido *von Neumann* arquitetura. Alternativamente, o (no momento em que já madura) *programação funcional* estilos foi apresentado por duas razões principais. Em primeiro lugar, devido ao seu potencial para a computação simultânea e paralela. Em segundo lugar - e Backus ênfase foi realmente colocar este -, devido à sua forte base algébrica.

Backus *álgebra de programas (funcionais)* foi providencial para alertar os programadores de computador que linguagens de computador por si só são insuficientes, e que apenas os idiomas que apresentam um *álgebra* para raciocinar sobre os objetos que pretendem descrever será útil no longo prazo.

O impacto da Backus argumento primeiro nas comunidades de ciência e arquitetura de computadores computação foi considerável, em particular, se avaliada em qualidade em vez de quantidade e para além da quase contemporânea *programação estruturada* tendência ¹⁴. Por outro lado, o segundo argumento para a mudança de programação de computador foi em grande parte ignorada, e só o chamado *álgebra de programação* minorias de investigação levada nesta direção. No entanto, os avanços nesta área ao longo das duas últimas décadas são impressionantes e podem ser totalmente apreciado pela leitura de um livro escrito há relativamente pouco tempo por Bird e de Moor [6]. Uma revisão abrangente da literatura volumosa disponível nesta área também podem ser encontradas neste livro.

Embora a necessidade de uma álgebra pointfree de programação era primeiro identificada por Backus, talvez influenciado por A de Iversen PL crescente popularidade nos EUA, nesse momento, a idéia de raciocínio e usar a matemática para transformar programas é muito mais antiga e pode ser rastreada até os tempos de trabalho de McCarthy sobre os fundamentos da programação de computadores [28], do trabalho de Floyd no sentido programa [9] e de Paterson e Hewitt de *comparativa schematology* [39]. Trabalho da chamada *transformação programa* escola já era muito expressivo em meados dos anos 1970, ver referências de instância [7].

A matemática adequados para a integração efectiva dessas linhas relacionadas, mas independentes de pensamento foi fornecido pela abordagem categorial de Manes e Arbib compilado em um livro [27] que tem muito fortemente influenciado a última década de ciência da computação teórica do século 20.

A chamada MPC (“Matemática do Programa de Construção”) comunidade tem sido entre os mais ativos na produção de um corpo integrado de conhecimentos sobre a álgebra da programação que tem encontrado na programação funcional um meio eloquente e paradigmático. A programação funcional tem uma tradição de absor-

¹⁴ Mesmo a linguagem de programação C e UNIX sistema operacional, com a sua implícita avour funcional, podem ser considerados como resultados sutis da tendência “vai funcionais”.

62 *Capítulo 2. UMA INTRODUÇÃO À pointfree PROGRAMAÇÃO*

ing frescas resultados de ciência da computação teórica, álgebra e teoria da categoria. Línguas tais como HASKELL [5] estão competindo para integrar os desenvolvimentos mais recentes e, portanto, são excelentes *prototipagem* veículos em cursos de cálculo programa, como acontece com este livro.

Para relativamente recente trabalho sobre este tema ver, por exemplo [12, 16, 17, 11].