Como obter o inverso de um tipo

Daniel Marshall

Escola de Computação, Universidade de Kent, Canterbury, Reino Unido

Dominic Orchard

Escola de Computação, Universidade de Kent, Canterbury, Reino Unido

Departamento de Ciência e Tecnologia da Computação, Universidade de Cambridge, Reino Unido

Abstrato

Na programação funcional, os tipos regulares são um subconjunto de tipos de dados algébricos formados por produtos e somas com suas respectivas unidades. Pode-se ver tipos regulares como formando um semi-anel comutativo, mas onde os axiomas usuais são isomorfismos em vez de igualdades. Nesta pérola, mostramos que tipos regulares em uma configuração *linear* permitem uma noção útil de *inverso multiplicativo,* permitindo-nos “dividir” um tipo por outro. Nossa aventura começa com uma exploração das propriedades e aplicações dessa construção, visitando vários tópicos da literatura, incluindo cálculo de programas, polinômios de Laurent e derivadas de tipos de dados. Exemplos são dados usando a extensão de tipos lineares de Haskell para demonstrar as ideias. Em seguida, passamos pelo espelho para descobrir o que pode ser possível em ambientes mais ricos; a linguagem funcional Granule oferece funções lineares que incorporam efeitos colaterais locais, o que nos permite demonstrar uma estrutura algébrica adicional. Por fim, discutimos se dualidades na lógica linear podem permitir a noção relacionada de um *inverso aditivo.*

2012 ACM Classificação de assuntos Teoria da computação ÿ Teoria dos tipos

Palavras-chave e frases tipos lineares, tipos regulares, álgebra de programação, derivadas

Identificador de objeto digital 10.4230/LIPIcs.ECOOP.2022.5

Software **de material suplementar** *(artefato aprovado pela avaliação de artefato ECOOP 2022):* https://doi.org/10.4230/DARTS.8.2.1

Financiamento Este trabalho é financiado por um EPSRC Doctoral Training Award (Marshall) e uma concessão EPSRC EP/T013516/1 (Verificando o uso de dados semelhantes a recursos em programas por meio de tipos).

Agradecimentos Obrigado a Nicolas Wu e Harley Eades III por seus valiosos comentários e discussões sobre rascunhos anteriores, e também aos revisores anônimos por seus valiosos comentários.

1 Prólogo: Consumindo com Inversos

Os tipos de dados algébricos são o pão com manteiga tanto da teoria quanto da prática da programação funcional. A visão algébrica dá origem a vastas possibilidades para manipular tipos e para “calcular” programas a partir de sua estrutura de tipos na tradição de Bird-Meertens [7, 8, 37].

Tipos regulares são um subconjunto de tipos algébricos formados a partir de produtos ×, somas +, unidade 1 e tipos vazios 0, e pontos fixos, dando origem a expressões de tipo polinomial [37, 40] e uma estrutura algébrica semelhante a um semi-anel comutativo. A parte multiplicativa é por produtos e o tipo de unidade, e a parte aditiva por somas e o tipo vazio. Entretanto, as leis dos semianéis são relaxadas a isomorfismos, por exemplo, $a \times (b \times c) \ddot{y} = (a \times b) \times c$ é testemunhada por uma bijeção entre as duas formas de associar uma tripla expressa em pares. A operação de cardinalidade $\ddot{y}|$ (mapear um tipo para seu tamanho) é então um homomorfismo de semi-anel (um functor) da estrutura de tipos para números naturais, por exemplo, $|a \times b| = |a||b|$. Isso fornece uma técnica útil para entender quando diferentes expressões de tipo são isomórficas, verificando se suas cardinalidades são iguais, um fato aproveitado por muitos estudantes por décadas. Na teoria da categoria, podemos modelar tipos regulares como uma *categoria de semiring (comutativa)* (ou *categoria de rig* [11]) ou uma *categoria bimonoidal (simétrica)* [26], com regras de semiring como isomorfismos naturais.

De qualquer maneira, uma rosa com qualquer outro nome ainda é doce.

36ª Conferência Europeia sobre Programação Orientada a Objetos (ECOOP 2022).
Editores: Karim Ali e Jan Vitek; Artigo nº 5; pp. 5:1–5:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Alemanha

5:2 Como obter o inverso de um tipo

Dados os ricos análogos algébricos e modelos para tipos, pode-se então (talvez à toa) se perguntar: se os tipos de produto são multiplicativos, *existe uma noção de inverso multiplicativo para tipos que nos permite dividir um tipo por outro?* Mostramos que esta questão tem uma resposta bastante clara e simples para *tipos lineares:*

ÿ **Definição 1.** O inverso multiplicativo do tipo *ÿ* é o tipo de função *que consome ÿ :*

ÿ ÿ1 ÿ *ÿ* ÿ 1

onde ÿ é o tipo de função linear, de funções que usam seus argumentos exatamente uma vez.

Os tipos lineares são um cenário ideal para capturar a ideia de *consumo ,* pois os tipos lineares tratam os valores como recursos que devem ser usados exatamente uma vez – eles nunca podem ser descartados (sem enfraquecimento) ou duplicados ou compartilhados (sem contração) [20, 51, 55] . Portanto, uma função *ÿ* ÿ 1 deve consumir seu argumento ao invés de simplesmente retornar um valor do tipo unitário.

Tipos regulares lineares têm produtos ÿ (“conjunção multiplicativa”) onde cada componente do par é usado exatamente uma vez e somas ÿ (“disjunção aditiva”) que se comportam como tipos normais de soma, embora qualquer componente que nos seja dado deve, é claro, ser usado linearmente . Tipos regulares lineares novamente formam uma estrutura de semi-anel comutativo, com igualdades como isomorfismos. *ÿ* ÿ 1, podemos consultar

Dado *ÿ* ÿ1 ÿ imediatamente o significado padrão do inverso multiplicativo em relação à sua teoria equacional: um grupo estende um monóide *(X,* •, e) tal que para cada *x* ÿ *X* existe um elemento *inverso* denotado *x* = *e* e ÿ1 ÿ X, para o qual *x* • *x* • ÿ1 *x* ÿ1 *x* = e. Se o uso da notação ÿÿ1 e a terminologia dos inversos for garantido, então poderíamos razoavelmente esperar *ÿ* ÿ *ÿ* ÿ= 1. Dada a definição acima de inverso, então uma direção desse isomorfismo, de *ÿ* ÿ1 ÿ *ÿ* a 1, é habitado via aplicação de função:

ÿ(u, x). *x : (ÿ* ÿ1 ÿ *ÿ)* ÿ 1 (1)

onde o primeiro componente do par consome o segundo componente. Da mesma forma podemos construir a versão simétrica *(ÿ* ÿ *ÿ* ÿ1) ÿ 1 invertendo primeiro as componentes do par.

Estes são os únicos habitantes de seus tipos: o inverso multiplicativo de *ÿ deve* consumir o valor de *ÿ* no outro componente do par devido às restrições de linearidade. De fato, partindo do objetivo de definir um inverso multiplicativo para um tipo linear tal que ÿ ÿ *ÿ* 1 se mantenha, por “currying” devemos ter um mapa inverso para ÿ1 da propriedade *ÿ ÿ* Portanto, *ÿ* ÿ 1 é a escolha mais natural para um ÿ1 para *ÿ* ÿ 1. ÿ primeiro precisam ser mapeados para *ÿ* ÿ 1 para consumir o valor do **,** como qualquer outro inverso faria tipo *ÿ* .

Chamamos a equação (1) e sua contraparte simétrica *de leis inversas* relaxadas seguindo a terminologia da teoria das categorias: estruturas *estritas* têm igualdades, *fortes* têm isomorfismos e *laxistas* têm apenas morfismos em uma direção. O resto da estrutura de semi-anel comutativo de tipos regulares lineares é “forte” como associatividade, comutatividade, etc. são isomorfismos. No entanto, para inversas, a direção oposta de 1 a *ÿ* ÿ *ÿ* não existe em geral: para qualquer *ÿ* ÿ1 não podemos necessariamente formar um par *ÿ* ÿ *ÿ* pois não temos um algoritmo para construir um valor *ÿ* arbitrário nem seu consumidor. Mesmo que possamos habitar 1 ÿ *(ÿ* ÿ *ÿ)* para um *ÿ* particular (por exemplo, se houver um valor padrão para um tipo e uma forma padrão de consumir seus valores) isso só forma um isomorfismo *ÿ* é que a divisão ÿ1 ÿ *ÿ* ÿ= 1 na configuração limitada de tipos de 1 elemento. O ponto crucial *perde* informação: *ÿ* ÿ1 ÿ *ÿ* ÿ 1 consome conhecimento do valor *ÿ* original .

Em uma configuração não linear (“cartesiana”), ainda poderíamos definir inversos de maneira ÿ1 ÿ *ÿ* ÿ 1, e a lei inversa frouxa *ÿ* como *ÿ* ÿ1 semelhante × *ÿ* ÿ 1 seria habitado de maneira semelhante pela aplicação de funções. No entanto, a não linearidade torna essa definição mais fraca e menos natural, pois ((ÿ ÿ 1) × *ÿ)* ÿ 1 também é habitado pela função *ÿ(u, x).1* que descarta ambos os argumentos e retorna a unidade valor. Poderíamos facilmente construir este termo de

tipo *(ÿ* ÿ1 × *ÿ)* ÿ 1 independentemente da definição que escolhemos para *ÿ*ÿ1 . É somente trabalhando na contexto de tipos lineares que nossa noção de inverso recebe significado, como uma função *ÿ* ÿ *ÿ* ÿ1 ÿ deve ÿ1 ÿ 1 tanto o termo *ÿ* quanto seu inverso, para o qual *ÿ ÿ* ÿ 1 é o ajuste natural.usar

Esta definição de *ÿ* ÿ1 permite-nos ver os tipos como uma estrutura algébrica que é *quase* um semicampo; um semicampo se assemelha a um semi-anel, exceto que cada elemento diferente de zero tem um inverso multiplicativo. A terminologia de *assimetria* também pode ser aplicada aqui; por exemplo, um *monóide enviesado* é aquele em que a associatividade e as propriedades unitárias são morfismos em vez de isomorfismos ou igualdades [48, 50]. Assim, nossa construção pode ser descrita como um "inverso de distorção multiplicativa". No futuro, usamos apenas "inverso" para abreviar.

Roteiro

Nesta pérola, exploramos várias aplicações e consequências desta ideia. Começamos programando com inversas em Haskell por meio da extensão de tipos lineares do Glasgow Haskell Compiler1 (baseado no trabalho de Bernardy et al. [6]) e passamos a considerar equações decorrentes de funções sobre inversas e outras implicações algébricas.

Um resultado interessante que descobrimos é que enquanto os tipos regulares produzem expressões de tipo polinomial, os tipos inversos produzem um análogo da generalização matemática dos *polinômios* de *Laurent* (Seção 4); estes diferem dos polinômios comuns porque podem ter termos de grau negativo, fornecendo uma noção mais geral de expoente para tipos regulares. Mas os inversos acabam tendo aplicações além do meramente teórico; mostramos que as inversas permitem que a noção de derivadas para tipos regulares (à la McBride [37]) seja generalizada, fornecendo a capacidade de obter a derivada de um tipo *em relação a outro tipo* (Seção 5). Isso produz uma maneira de gerar tipos de dados com n-buracos (buracos de *n* elementos contíguos) que aplicamos ao idioma de programação comum de cálculos de estêncil.

Na segunda metade, consideramos as possibilidades de desenvolver ainda mais a estrutura algébrica dos tipos inversos, trabalhando em configurações mais ricas e expressivas. Acontece que o inverso multiplicativo se torna uma involução (Seção 6) se pudermos expressar *sequencialmente funções realizáveis* [32] que realizam efeitos colaterais locais que não são observáveis externamente. Demonstramos isso usando canais de sessão linear à la Lindley e Morris [30] que permitem inversos para fazer mais computacionalmente. Mostramos exemplos na linguagem funcional moderna Granule, que tem tipos lineares em seu núcleo [42]. Essa involução também produz uma construção semelhante à conhecida mônada de continuação, que discutimos brevemente.

Por fim, mostramos que também é possível definir um inverso aditivo (Seção 7). No entanto, isso requer trabalhar em uma configuração diferente, onde os produtos são dados pela lógica linear & em vez de ÿ e, da mesma forma, as somas são ` em vez de ÿ. Assim, embora possamos desenvolver a teoria para cada tipo de inversa separadamente, ainda não existe nenhuma teoria de tipo em que os dois possam coexistir.

Um objetivo colateral dessa pérola é popularizar as habilidades crescentes das linguagens funcionais modernas para expressar tipos lineares. Para aqueles não familiarizados com tipos lineares e que desejam ir além das intuições aqui, o Apêndice A fornece algumas regras de digitação e sintaxe padrão. Também fornecemos um artefato2 incluindo todos os exemplos de código fornecidos em Haskell e Granule, para ajudar na compreensão e permitir mais experimentações.

[1] Disponível a partir do GHC 9.0.1, https://www.haskell.org/ghc/download_ghc_9_0_1.html, lançado em fevereiro de 2021.

[2] https://doi.org/10.5281/zenodo.6275280

2 Programação com inversos

Estamos em um momento emocionante para nossa comunidade. Finalmente, mais de 30 anos após sua concepção em lógica [20], os tipos lineares estão começando a ganhar espaço nas principais linguagens de programação funcional. Uma dessas linguagens é Haskell, via extensão de tipos lineares do GHC [6] que usa um sistema de tipo graduado [42] baseado na anotação de tipos de função a %r -> b com sua "multiplicidade" r (que também pode ser entendida como um *coefeito,* ou *efeito de consumo* [43]), que descreve quantas vezes o argumento é usado. Em Haskell, isso pode ser 1 representando comportamento *linear* ou Muitos representando comportamento *irrestrito ,* incluindo a possibilidade de 0 usos. Assim, podemos descrever inversos e uma versão improvisada da lei inversa frouxa da seguinte forma:3

```
1 tipo Inverso a = a %1 -> ()                    -- recall () é o tipo de unidade 1 de Haskell
2
3 divida :: a %1 -> Inverse a %1 -> ()
4 divide xu = ux
```

A nomeação de divide é para evocar a intuição usual associada a grupos onde *a/ b* = a•be desde ÿ1 que esta função "age" o consumo do primeiro input pelo segundo.

Existem outras linguagens de tipagem linear nas quais também é possível aplicar prontamente nossa noção de inversos, por exemplo, ATS [47], Alms [49] e Quill [39]. Por meio de algumas traduções, também podemos representar inversos em idiomas com sistemas de tipos graduados mais expressivos, como Granule [42] e Idris 2 [10], que podem descrever a linearidade, bem como outros tipos de dados engenhosos. Nós nos concentramos em Haskell por enquanto, mas as ideias são as mesmas, não importa o idioma.

No cenário concreto de uma língua real, podemos agora dar um exemplo de habitante de tipo inverso. Estes são normalmente definidos por alguma correspondência de padrão em todas as entradas possíveis, onde o ato de correspondência de padrão no valor de entrada consome a entrada à medida que inspeciona seu valor. Por exemplo, um inverso ao tipo booleano de Haskell é dado por:

```
1 boolDrop :: Bool inverso 2 boolDrop
True = () 3 boolDrop False = ()
```

A biblioteca de base linear para Haskell fornece uma classe de tipo para aqueles tipos que são "consumíveis ": habitantes do inverso do tipo a. A instância de Consumable para o tipo booleano é a função boolDrop definida explicitamente acima.

```
1 classe Consumível a onde
2         consumir :: a %1-> ()
3
4 instâncias Consumable Bool onde
5       consumir Verdadeiro = ()
6       consumir Falso = ()
```

Vários tipos embutidos como Int têm uma implementação "linearmente insegura" que simplesmente descarta o argumento em vez de, digamos, consumir um inteiro de máquina combinando no caso 0 e, de outra forma, consumindo recursivamente o inteiro decrementado em 1, o que seria seguro, mas lento ! Essa operação de enfraquecimento explícito também pode ser gerada algoritmicamente a partir de um tipo regular, seguindo um mecanismo de derivação genérico [23].

---

[3] Observe que primeiro precisamos habilitar a extensão de tipos lineares, usando o pragma {-# LANGUAGE LinearTypes #-}; isso ficará implícito em todos os trechos de Haskell em toda a pérola.

Um aspecto fundamental dessa disciplina de digitação é que não queremos que certos tipos sejam consumíveis sem efeitos colaterais; por exemplo, manipuladores de arquivos, soquetes, canais ou qualquer outro dado que atue como um proxy para um recurso para o qual existe algum protocolo de interação. Na Seção 6, vemos habitantes mais interessantes de tipos inversos em um cenário mais expressivo.

Podemos considerar as propriedades algébricas dos inversos e entendê-los através das lentes dos tipos regulares lineares usando esta definição, tendo em mente que nossos inversos são frouxos e, portanto, as propriedades serão mantidas apenas em uma direção. Por exemplo, considere a seguinte propriedade, que é uma aplicação simples da distributividade da multiplicação sobre a adição.

$$(\ddot{y} \ddot{y} 1) \ddot{y} \ddot{y}^{\ddot{y}1} \quad \ddot{y} = ((\ddot{y} \ddot{y} \ddot{y}^{\ddot{y}1}) \ddot{y} \ddot{y}^{\ddot{y}1}) \quad \ddot{y} \; 1 \; \ddot{y} \; \ddot{y}^{\ddot{y}1} \tag{2}$$

Podemos entender *ÿ* ÿ1 como a versão linear do tipo de dados Haskell Maybe tradicional (chamado de *opção* em ML) e, assim, recuperar a seguinte definição de função em Haskell correspondente à (in)equação acima, nos dando uma maneira de distribuir um inverso em um valor Maybe .

```
1 MaybeNeg :: (Talvez a) %1 -> Inverse a %1 -> Maybe (Inverse a) 2 MaybeNeg Nada u =
Apenas u 3 MaybeNeg (Apenas n) u =
LetUnit (divide nu) Nada
4
5 letUnit :: () %1 -> a %1 -> a -- Resumos 'let () = t1 em t2' - necessário desde 6 letUnit () x = x
                                -- let bindings são atualmente sempre não lineares.
```

No segundo caso de MaybeNeg, não podemos simplesmente retornar Nothing , pois u e n são digitados linearmente ; devemos primeiro aplicar u a n (via divisão) para consumir ambos os valores. Em seguida, queremos let () = divide nu in Nothing, mas let-bindings lineares ainda não foram implementados (a partir do GHC 9.2.2, lançado em março de 2022), então abstraímos esse padrão como a função letUnit .

3 Cálculo com inversos

Os tipos regulares vêm equipados com várias equações governando suas operações que podem ser usadas para raciocinar sobre programas funcionais [18] e até mesmo derivar implementações a partir de especificações equacionais (o formalismo *de Bird-Meertens)* [7, 8, 19]. Consideramos aqui equações análogas para calcular com inversas. Exploramos essas equações sob a perspectiva do cálculo ÿ linear com tipos regulares, ilustrando alguns pontos usando Haskell por conveniência. Pode-se traduzir livremente entre os dois.

Em uma configuração de tipos lineares, muitas das equações usuais que regem os produtos não estão disponíveis para nós, porque a “tupla” que combina as funções regulares $f : A \ddot{y} B$ e $g : A \ddot{y} C$ em $\ddot{y}f,$ gÿ : $A \ddot{y} (B \times C)$ viola a linearidade copiando um valor do tipo A, e as projeções ÿ1 : $A \times B \ddot{y} A$ e ÿ2 : $A \times B \ddot{y} B$ violam a linearidade ao descartar um componente de um par. Podemos, no entanto, trabalhar com ÿ como um bifuntor (que eleva $f : A \ddot{y} B$ e $g : C \ddot{y} D$ para $f \ddot{y} g : A \ddot{y} C \ddot{y} B \ddot{y} D$) e coupling $[h,$ k] : $A \ddot{y} B \ddot{y} C$ (para $h : A \ddot{y} C$ e $k : B \ddot{y} C$) ainda está disponível. Assim temos equações para (bi)funtorialidade de ÿ e ÿ:

$$id \ddot{y} id = id \quad (f \ddot{y} g) \ddot{y} (h \ddot{y} k) = (f \ddot{y} h) \ddot{y} (g \ddot{y} k)$$

$$id \ddot{y} id = id \quad (f \ddot{y} g) \ddot{y} (h \ddot{y} k) = (f \ddot{y} h) \ddot{y} (g \ddot{y} k)$$

e equações interagindo cotuplicação, injeções e o bifuntor ÿ, por exemplo, para citar alguns:

$$[f, g] \ddot{y} \text{inl} = f \qquad [f, g] \ddot{y} \text{inr} = g \qquad [h \ddot{y} \text{inl}, h \ddot{y} \text{inr}] = h$$

Por brevidade, omitimos o resto, pois não são o objeto aqui. O Apêndice A.1 fornece as equações restantes (que são o subconjunto daquelas de Gibbons [18] que são permitidas em uma configuração linear). Existem várias outras equações decorrentes dos isomorfismos de tipos regulares (Seção 1), por exemplo, para os isomorfismos testemunhando associatividade com *ÿ : (A* ÿ B) ÿ *C* ÿ *A* ÿ *(B* ÿ C) e *ÿi* é seu inverso, então nós tem equações *ÿ* ÿ *ÿi = ÿi* ÿ *ÿ = id.*

Inverso como um functor

Algumas equações surgem do simples fato de que o inverso multiplicativo é um funtor contravariante e, portanto, temos uma função “mapa” contravariante por meio da composição da função:

```
1 comap :: (b %1 -> a) %1 -> Inverso a %1 -> Inverso b 2 comap fg = \x -> g
(fx)
```

A ação de um funtor em um morfismo é comumente escrita usando o mesmo símbolo de sua ação em objetos (tipos), assim como visto acima para ÿ e ÿ. No entanto, escrever comap aplicado a *f* como $f^{ÿ1}$ dá a impressão errada: não estamos representando o inverso de uma função, mas ÿ1 para levantar uma função para trabalhar em inversos. Portanto, escrevemos *f* comap *f* para evitar confusão.4

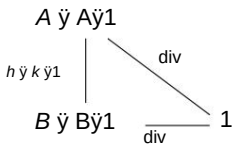
Portanto, temos as equações de funcionalidade para inversas:

idÿ1 = *id* gÿ1 ÿ $f^{ÿ1}$ = $(f$ ÿ g$)^{ÿ1}$

Seja div : *A* ÿ Aÿ1 ÿ 1 a lei inversa lax como uma função no cálculo ÿ linear (a forma não curável da função divide :: a -> Inverse a -> () que definimos anteriormente para Haskell).
Dadas duas funções *h : A* ÿ *B, k : B* ÿ A, temos então a seguinte propriedade de naturalidade:

div ÿ *(h* ÿ *k* ÿ1)= div

que pode ser visto mais claramente em um diagrama como:



ou seja, transformar um valor e seu inverso antes do consumo é o mesmo que apenas consumirmos o valor original. Esta propriedade decorre das definições. Revisitamos essa lei na Seção 6, uma vez que introduzimos funções de consumo que podem ter algum efeito colateral (seguro por linearidade).

Estrutura monoidal de inversas

O functor inverso (contravariante) também possui estrutura de functor monoidal adicional, que podemos escrever em Haskell simplesmente como:

```
1 munit :: () %1 -> Inverse () 2 munit () = (\
() -> ())
3
4 mmult :: (Inverse a) %1 -> (Inverse b) %1 -> Inverse (a, b) 5 mmult fg = \(a, b) -> letUnit
(fa) (gb)
```

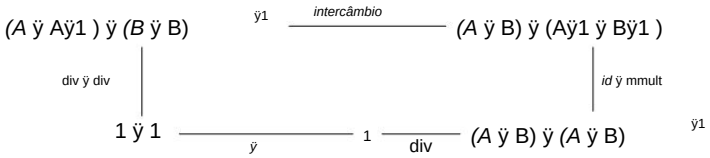
4 Consideramos usar a notação *ÿ* ÿ1 para tipos também, mas achei muito feio para colocar em todos os lugares.

ou seja, munit consome um valor unitário por correspondência de padrão e depois retorna unidade (a função de identidade polimórfica padrão teria funcionado igualmente bem) e mmult retorna uma função que consome um par usando f para consumir o primeiro componente e g para consumir o segundo. Os axiomas usuais de um funtor monoidal (relaxado) (associatividade e leis unitárias) [34] são válidos, interagindo com a estrutura monoidal de ÿ; detalhamos esses axiomas no Apêndice A.1.

Essa estrutura de funtor monoidal em inversos nos dá a ideia simples de que podemos combinar vários inversos em um inverso composto; em outras palavras, um par de consumidores pode se tornar um consumidor de pares. Isso satisfaz a seguinte equação:

ÿ ÿ (div ÿ div) = div ÿ *(id* ÿ mmult) ÿ *intercâmbio*

onde *ÿ :* 1ÿ1 ÿ 1 colapsa unidades e *troca :* (AÿB)ÿ(C *ÿD)* ÿ (AÿC)ÿ(B *ÿD)* é derivado de isomorfismos de associatividade e comutatividade. Esta regra pode ser vista mais claramente como um diagrama:



ou seja, podemos realizar duas eliminações inversas ou podemos reorganizar, combinar as inversas em uma e então aplicar uma única eliminação.

A ideia de combinar produtos de inversos leva naturalmente a noções de *exponenciação ,* mas com expoentes negativos, que exploramos a seguir.

4 Exponenciação com Inversas

No semi-anel padrão de tipos regulares (lineares) discutidos na Seção 1, os construtores de tipo 0, 1, ÿ e ÿ geram polinômios sobre alguma metavariável de tipo onde termos com expoentes = 1. Para *n* ÿ 0, isso nos dá $ÿ^{n}$ são representados pelo produto n-wise de *ÿ* onde $ÿ^{0}$ o usual leis de expoentes positivos até isomorfismo (usando associatividade e comutatividade):

ÿa, b.(a ÿ 0 ÿ *b* ÿ 0) *ÿ ÿa.(a* ÿ ^{a} b ÿ *ÿ* ÿ= $ÿ^{a+b}$ (exp+)

0) *ÿ* ^{a} ÿ *ÿ* ^{a} ÿ= *(ÿ* ÿ *ÿ) a* (expÿ)

A introdução do inverso multiplicativo nos permite generalizá-los para expoentes negativos e, assim, gerar *polinômios de Laurent* sobre tipos, que diferem dos polinômios comuns porque podem ter termos de grau negativo [28].

Definimos exponenciação sobre um tipo *ÿ* para expoentes negativos como

ÿa.(a ÿ 0) $ÿ^{ÿ}$ *ÿa* ÿ *(ÿ* ^{ÿ1}) *um*

Por exemplo, $ÿ^{ÿ2}$ = *(ÿ* ^{ÿ1}) capturando um par de inversos. 2 ÿ1 ÿ1 ÿ *ÿ* = *ÿ*

É claro que a primeira lei exponencial (equação exp+) se generaliza no caso de ambos como produto de coeficientes são negativos, pois definimos *ÿ* da ^{ÿn} *n* inverso *ÿ* ^{ÿ1} em muito o mesma forma que $ÿ^{n}$ representa um produto de *n* valores do tipo ÿ , ou seja,

ÿa, b.(a ÿ 0 ÿ *b* ÿ 0) *ÿ* ^{ÿa} ÿ *ÿ* ^{ÿb} ÿ= *ÿ* ÿ(a+b) (exp-)

Isso é um isomorfismo, pois equivale a reassociar produtos.

5:8 Como obter o inverso de um tipo

No caso de apenas um coeficiente ser negativo, nossa noção de inverso satisfaz uma generalização da lei de exponenciação como uma propriedade frouxa:

$$\ddot{y}a, \text{b.}(a \ddot{y} 0 \ddot{y} b \ddot{y} 0) \ddot{y} \quad {}^{a} \ddot{y} \ddot{y} \quad {}^{\ddot{y}b}{}_{\ddot{y}} \quad {}^{a\ddot{y}b \ddot{y}} \qquad \text{(exp±)}$$

A lei inversa frouxa $\ddot{y} \ddot{y} \ddot{y}$ ${}^{\ddot{y}1}$ $\ddot{y}$ 1 é então uma especialização do anterior com $a = b = 1$ desde = 1. Como outro ${}^{0\ \ddot{y}}$ exemplo, considere $a = 5$ e $b = 3$; então podemos eliminar/consumir três elementos de um quíntuplo para obter pares $(a \ddot{y} b = 2)$. Isso levanta uma questão combinatória interessante sobre o número de funções que habitam esse tipo (testemunhando essa propriedade frouxa).

ÿ **Proposição 1.** *Acontece que o número de possíveis testemunhas da lei inversa frouxa* ${}^{\ddot{y}\ a}$ $\ddot{y}b\ a\ddot{y}b\ \ddot{y}\ \ddot{y}\ \ddot{y}$ *se* $a \ddot{y} 0$, $b \ddot{y} 0$ *e* $a \ddot{y} b$ *é simplesmente* $\dfrac{a}{b} \times (a \ddot{y} b)! = b! \dfrac{a!}{}$.

A intuição para isso é que, se tivermos algum número a de valores e algum número b de inversos, e temos mais valores do que inversos, devemos aplicar cada inverso a um valor, e a única escolha que podemos fazer é quais valores vamos consumir.
Combinatorialmente existem $b! \dfrac{a}{(a\ddot{y}b)!} = \dfrac{a!}{}$ maneiras de escolher b dos elementos a, deixando $(a \ddot{y} b)$ elementos restantes no final que podemos permutar em qualquer ordem (daí o fator $(a \ddot{y} b)!$ que cancela). Se $b \ddot{y} a$, então devemos consumir todos os valores e, inversamente, estamos simplesmente escolhendo quais inversos *de* a usaremos para fazer isso, dando um resultado de $\overline{b!}\ a!$.

Até agora no artigo, examinamos o cálculo ÿ linear e os tipos lineares em Haskell, onde os inversos não têm conteúdo computacional significativo além de consumir um valor linear.
No entanto, mais adiante na Seção 6, trabalharemos em um cenário com inversos que incorporam efeitos colaterais locais. Observe que, em tal configuração, a ordem em que aplicamos os inversos pode ser importante. Assim, consideramos também o resultado que obtemos ao levar isso em consideração.

ÿ **Proposição 2.** *Se os reordenamentos forem considerados distintos, então o número de testemunhas possíveis se* $a \ddot{y} 0$, $b \ddot{y}$ ${}^{a}\ \ddot{y}\ \ddot{y}\ {}^{\ddot{y}b}{}_{\ddot{y}}\ {}^{a\ddot{y}b\ \ddot{y}}$ $\ddot{y}$ 0 *e* $a \ddot{y} b$ *é um* fatorial (a!). *da lei de exponenciação frouxa*

Isso é derivado por $P(a, \text{b}) \times (a\ddot{y}b)!$, ou seja, $\times(a\ddot{y}b)! = \text{um}!$ já que novamente podemos “consumir” $(a \ddot{y} b)!$ elementos por meio de seus inversos em qualquer ordem, dando o número de permutações $P(a, \text{b})$ de comprimento b retirado de a, e há $(a \ddot{y} b)$ elementos restantes deixados depois para permutar. Aqui examinamos a prova na íntegra, pois ela é instrutiva sobre como habitar a lei em ambos os casos.

Prova. Provamos que o número de possíveis testemunhas da lei de exponenciação lax é dado acima por indução em b (e lembre-se que assumimos $a \ddot{y} b$).

- $(b = 0)$ Para o caso base, estamos então considerando $|\ddot{y}\ a\ \ddot{y}\ \ddot{y}\ {}^{0\ \ddot{y}}\ {}^{a\ddot{y}0\ \ddot{y}}| = |\ddot{y}\ a\ {}^{\ddot{y}\ \ddot{y}}\ \textit{um}|$ ${}^{0\ \ddot{y}} = 1$. Os habitantes possíveis de $\ddot{y}$ produto de $\ddot{y}$ ${}^{a\ \ddot{y}\ \ddot{y}}\ {}^{a}$ desde então são apenas permutações do a-wise e assim $|\ddot{y}\ aa| = a!$, dando o resultado aqui. $(b = 1)$ Em seguida, consideramos
- outro caso em que $b = 1$, pois isso é instrutivo (embora não seja necessário). Aqui, precisamos descobrir quantas maneiras existem de habitar $\ddot{y}$ aÿÿ ie, quantas maneiras de “cortar” um $\ddot{y}$ de um produto a-wise de $\ddot{y}$. ${}^{\ddot{y}1\quad \ddot{y}\quad a\ddot{y}1\ \ddot{y}}$,

  Podemos construir *muitos* termos como testemunhas para este tipo escolhendo qualquer um dos valores $\ddot{y}$ ÿ1 para consumir com o inverso $\ddot{y}$:

  $\ddot{y}((a1, \text{a2}, \ldots, \text{an}), \text{u}) = \textbf{let}\ () = u\ \text{a1}\ \textbf{in}\ (\text{a2}, \ldots, \text{an}) : (\ddot{y}\ \text{a}\ \ddot{y}\ (\ddot{y}\ \ddot{y}\ 1))\ \ddot{y}\ \ddot{y}$ ${}^{\text{a-1}}$
  $\ddot{y}((a1, \text{a2}, \ldots, \text{an}), \text{u}) = \textbf{let}\ () = u\ \text{a2}\ \textbf{in}\ (\text{a1}, \text{a3}, \ldots, an) : (\ddot{y}\ \text{a}\ \ddot{y}\ (\ddot{y}\ \ddot{y}\ 1))\ \ddot{y}\ \ddot{y}$ ${}^{\text{a-1}}$
  . . .
  $\ddot{y}((a1, \text{a2}, \ldots, \text{an}), \text{u}) = \textbf{let}\ () = u\ an\ \textbf{in}\ (\text{a1}, \text{a2}, \ldots, \text{an}\ddot{y}1) : (\ddot{y}\ \text{a}\ \ddot{y}\ (\ddot{y}\ \ddot{y}\ 1))\ \ddot{y}\ \ddot{y}$ ${}^{\text{a-1}}$

Seja qual for a testemunha que escolhermos, um inverso foi aplicado, então *um* ÿ 1 dos elementos originais permanece. Estes podem ser permutados em qualquer ordem, dando um total de *a* × *(a* ÿ 1)! = um! testemunhas.

(b = *n* + 1) (Passo indutivo) Precisamos mostrar que |*ÿ* a ÿ *ÿ* ÿ(n+1) ÿ= *ÿ* ÿ(n+1) ÿ *ÿ* aÿ(n+1)| = a!. Nós sabemos *ÿ* ÿ1 ÿ *ÿ* ÿn pois podemos simplesmente "dividir" um dos *n* + 1 inversos do produto restante de *n* inversos. Similarmente ao caso base *(b* = 0), primeiro aplicamos este inverso a qualquer um dos elementos de *ÿ* dos elementos originais *a* ; há *uma* escolha possível aqui. Isso deixa *um* - 1 restantes, e *n* inversos. Portanto, podemos raciocinar da seguinte forma:

ÿ a ÿ *ÿ* = ÿ(n+1) | ÿ *ÿ* aÿ(n+1)|

|(ÿ a ÿ *ÿ* ÿ1) ÿ *ÿ* ÿn ÿ *ÿ* aÿ(n+1)| (dividir um inverso, como acima) *(uma*

= *a* × |*ÿ* = *a* aÿ1 *ÿn* ÿ *ÿ* ÿ *ÿ* (aÿ1)ÿn | maneira de aplicar um inverso)

× *(a* ÿ 1)! = um! (indução com *a* ÿ 1)

ÿ

Observe que *a* ÿ *(n*+ 1) implica (aÿ1) ÿ n, o que é necessário para aplicar indutivamente a proposição. | (com *a* ÿ b), Outra maneira de ver isso intuitivamente é a seguinte. Para |*ÿ* a *ÿÿ* encontrar *ÿb* ÿ aÿb ÿ para uma testemunha, simplesmente precisamos organizar os elementos *a* em qualquer ordem, de modo que o primeiro *b* seja atribuído aos *b* inversos e os restantes *a* ÿ *b* não sejam usados. Há um! maneiras de organizar os elementos *a ,* dando o resultado. Novamente, no caso de *b* ÿ *a ,* consideramos apenas os inversos de *b ,* atribuindo inversos aos elementos, dando um total de b! pedidos. ÿ

Depois desse *divertimento* nos habitantes da lei de exponenciação negativa (exp±) e sua cardinalidade, consideramos a segunda lei de exponenciação (às vezes chamada de *poder de um produto),* que para tipos regulares é o seguinte isomorfismo, por comutatividade e associatividade:

ÿa.(a ÿ 0) *ÿ* a a ÿ *ÿ* ÿ= *(ÿ* ÿ *ÿ) a* (expÿ)

Para uma versão desta lei com expoentes negativos, um caso especial com *a* = ÿ1 já é fornecido pela estrutura do funtor monoidal da Seção 3 com mmult : *ÿ* que combina ÿ1 *ÿÿ* ÿ1 ÿ *(ÿ ÿÿ)* ÿ1 inversos. Compondo isso com a lei de expoentes negativos duplos expÿ : *ÿ* ÿa *ÿb* ÿ= *ÿ* ÿ(a+b) produz a generalização para todos os expoentes negativos: ÿ *ÿ*

ÿa.(a ÿ 0) *ÿ* ÿa ÿ *ÿ* ÿa ÿ *(ÿ* ÿ *ÿ) ÿa* (expÿÿ)

Por exemplo, se *a* = 2, então isso é derivado como:

ÿ2 *ÿ* ÿ *ÿ* ÿ2 ÿ+ÿ ÿ= *(ÿ* ÿ1 ÿ *ÿ* ÿ1) ÿ *(ÿ* ÿ1 ÿ *ÿ* ÿ1) mmultÿmmult ÿ *(ÿ* ÿ *ÿ ÿ)* ÿ1 ÿ *(ÿ* ÿ *ÿ ÿ)* ÿ1 expÿÿ= *(ÿ* ÿ *ÿ)* ÿ2

onde o isomorfismo à esquerda aplica associatividade *ÿ* e comutatividade ÿ.

No geral, os expoentes negativos generalizam a história dos "inversos como consumidores". Dado um produto de dois tipos exponenciais, um positivo e outro negativo, então "acionar" os inversos envolve consumir um certo número de elementos de um produto, deixando-nos com o restante. Isso captura a noção de "projeção", onde alguma parte de um tipo é descartada e outra parte é retida. Esse padrão é relevante a seguir, onde ter inversos e expoentes negativos para tipos regulares nos permite definir a derivada de um tipo *em relação a outro tipo.*

5 Diferenciando com Inversos

Com nossa noção de inverso multiplicativo em mãos, podemos aplicar outras ideias da matemática que dependem da presença da divisão.

5:10 Como obter o inverso de um tipo

Primeiro, vamos relembrar a notável característica dos tipos regulares (com variáveis de tipo adicionadas) de que é possível calcular sua *derivada* aplicando as leis do cálculo de Newton-Leibniz [29]. Isso produz um tipo de dados complementar de "contextos de um buraco" para o tipo original, uma devido a McBride [37]. Por exemplo, para o tipo paramétrico $\ddot{y}$ o ^{4} bela ideia (4-tuplas com mesmo tipo) sua derivada em relação a $\ddot{y}$ é $4\ddot{y}$ a intuição ^{3}, equivalente a $\ddot{y}$ $3\,3\,3 + \ddot{y} + \ddot{y}$ elementos todos de 3 . O + $\ddot{y}$ por trás disso é que existem quatro maneiras distintas nas quais você pode pegar o tipo de dados original (4-tuplas) e remover um único elemento (criando um buraco), deixando o contexto circundante (um triplo dos três elementos restantes). Podemos visualizar essas quatro possibilidades (cada uma do tipo $\ddot{y}$ ^{3}) da seguinte forma, onde ÿ representa o "buraco" e onde *x, y, z, w :* ÿ:

(-, *y, z,* w) *(x,* ÿ*, z,* w) *(x, y,* ÿ*,* w) *(x, y, z,* ÿ)

Escrevemos esta derivada como ^{4}), ou seja, a derivada parcial em relação a $\ddot{y}$ mantendo outras ÿÿ(variáveis ÿ constantes (incluindo variáveis de recursão ou outros parâmetros de tipo).

Essa abordagem pode ser aplicada a tipos regulares recursivos. Por exemplo, a técnica de McBride calcula ÿÿ(List ÿ) = $\ddot{y}\ddot{y}(\mu X.1 + \ddot{y} \times X)$ = (List ÿ) × (List ÿ) representando a ideia de que uma lista com um único buraco é equivalente a um par de listas – o prefixo dos elementos antes do furo e o sufixo dos elementos depois do furo. O tipo de dados dos zíperes da lista (à la Huet [22]) é então dado por $\ddot{y}$ × ÿÿ(Lista ÿ) onde temos um valor $\ddot{y}$ que "preenche" a posição do buraco, pareado com seu contexto. É possível estender esta noção ainda mais para considerar derivados de tipos de dados gerais que atuam como contêineres [1] ou mesmo dados que estão em processo de transformação de um tipo para outro [38], mas aqui vamos nos concentrar nos mais simples casos.

Um tipo de dados de contextos com *dois* buracos pode ser obtido por diferenciação repetida, ie, $\ddot{y}\ddot{y}(\ddot{y}\ddot{y}(f(\ddot{y}))),$ e assim podemos computar contextos com *n* buracos tomando a n-ésima derivada. Esses buracos são todos independentes; eles podem aparecer em qualquer lugar no tipo. Por exemplo, ÿÿ(ÿÿ(List ÿ)) = (List ÿ×(List ÿ×List ÿ))+ ((List ÿ×List ÿ)×List ÿ), representando duas maneiras de adicionar outro buraco a uma lista que já tem um buraco: ou temos outro buraco na sublista esquerda ou na sublista direita.

Embora as derivadas acima sejam baseadas em tipos regulares padrão que possuem a estrutura da lógica intuicionista, os tipos regulares lineares formam um semi-anel exatamente da mesma maneira sintática. Assim, a noção de derivação de um tipo se aplica igualmente bem na configuração linear. Consideramos derivadas de tipos regulares lineares daqui para frente e mostramos que as inversas nos permitem definir o que significa obter uma derivada *em relação a outro tipo.*

Primeiro, permaneçamos no terreno firme da análise real. Lembre-se do cálculo que podemos obter a derivada de uma função *f* em relação a outra função *g* pelo seguinte método:

$$\ddot{y}g(\ddot{y})(f(\ddot{y})) = \frac{\ddot{y}\ddot{y}(f(\ddot{y}))}{\ddot{y}\ddot{y}(g(\ddot{y}))} \tag{3}$$

Por exemplo, tomando $f(\ddot{y}) = \ddot{y}$ ^{4} e $g(\ddot{y}) = \ddot{y}$ ^{2}, então:

$$\ddot{y}(\ddot{y}2)\ddot{y}^4 = \frac{\ddot{y}\ddot{y}(\ddot{y}^4)}{\ddot{y}\ddot{y}(\ddot{y}2)} = \frac{3\,4\ddot{y}}{2\ddot{y}}\,2 = 2\ddot{y} \tag{4}$$

Dando a isso uma interpretação em tipos regulares, em vez de R, relembre $\ddot{y}$ em relação a $\ddot{y}$ ^{4} é uma tupla 4 (uma quádruplo) e $\ddot{y}$ que é o ^{2} é uma 2-tupla (um par). Diferenciando $\ddot{y}$ ^{4} ^{2} rende $2\ddot{y}$ ^{2} tipo de dados que captura os dois contextos possíveis obtidos removendo um par do tipo original, deixando dois elementos no contexto restante. Chamamos o par removido aqui de *2 furos,* pois ele captura dois furos contíguos (adjacentes). Observe que isso é diferente do caso, descrito anteriormente, de diferenciação em relação a $\ddot{y}$ duas vezes; isso dava dois orifícios independentes que não precisavam necessariamente ser contíguos. O tipo resultante aqui $2\ddot{y}$ pode ser interpretado como o tipo de contextos de 2 furos, ilustrado como: ^{2} $^{2 = \ddot{y}}$ $+ \ddot{y}$ ^{2}

(ÿ1, ÿ2, *z,* w) *e* *(x, y,* ÿ1, ÿ2) (5)

onde -1 e ÿ2 correspondem aos dois componentes sucessivos do 2-hole. Não podemos ter o 2-buraco dividindo os dois elementos restantes como *(x,* ÿ1, ÿ2, y); o tipo de dados de contextos de 2 furos visualiza todo o tipo em termos de pares.

A derivada calculada em (4) estava em R e *então* interpretamos o resultado como um tipo. em Isso, no entanto, não generaliza bem. Considere a derivada de *ÿ* 3 relação a *ÿ* 2:

$$\ddot{y}(\ddot{y}2)\ddot{y} \quad 3 = \frac{\ddot{y}\ddot{y}(\ddot{y}^{3})}{\ddot{y}\ddot{y}(\ddot{y}2)} = \frac{2\,3\ddot{y}}{2\ddot{y}} = \frac{3}{2}\ddot{y}$$

Simplificamos o resultado seguindo os axiomas dos campos aqui, mas ficamos com o pesado que $\frac{3}{2}$ não podemos traduzir significativamente no reino dos tipos. Usando nossa abordagem de em vez disso, os inversos multiplicativos produzem um resultado aplicável de forma mais geral para tipos regulares.

ÿ **Definição 2.** A derivada de um tipo regular em relação a outro tipo regular é:

$$\ddot{y}g(\ddot{y})f(\ddot{y}) = \ddot{y}\ddot{y}f(\ddot{y})\ \ddot{y}\ (\ddot{y}\ddot{y}g(\ddot{y}))\ddot{y}1 \qquad (6)$$

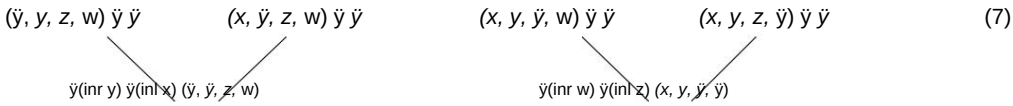
Essa construção produz a derivada usual do numerador, emparelhada com um consumidor da derivada do denominador.

Voltando ao exemplo de derivação da equação *ÿ* (4), em vez disso,4 em relação a *ÿ* 2 mostrado em aplicamos a abordagem inversa da Definição 2, que produz:

$$\ddot{y}\ddot{y}2\ (\ddot{y}^{4}) = 4\ddot{y}^{3}\ \ddot{y}\ (2\ddot{y}) \quad \ddot{y}1$$

Este é o tipo $4\ddot{y}^{3}$ de contextos de 1 furo para 4 tuplas (os quatro possíveis triplos resultantes da remoção de um elemento) emparelhado com um inverso que pode ser usado para consumir um valor *ÿ* adicional para criar um 2 furos. Este inverso consome valores *2ÿ* , ou seja, *ÿ ÿ ÿ*, onde o valor *ÿ* é marcado com um bit extra de informação para explicar ao inverso qual elemento está sendo removido para criar o 2-hole. Podemos ver isso como os quatro 1-buracos possíveis com um inverso *ÿ : (2ÿ)*

ÿ1 que pode ser acionado para recuperar os 2 furos para 4 tuplas como na equação (5):



(7)

com as injeções usuais inl : *A* ÿ *A* ÿ *B* e inr : *B* ÿ *A* ÿ B.

Juntamos tudo isso em Haskell para definir uma noção de contextos de 1 furo para 4 tuplas (QuadContexts abaixo), que emparelhamos com consumidores para criar 2 furos (QuadTwoContexts abaixo):

```
1 -- Representa 4a^3 (as quatro maneiras possíveis de remover um elemento de uma tupla 4). 2 dados
QuadContexts a =
3         Mk1 aaa -- contexto: -, y, z, w
4       | Mk2 aaa -- contexto: x, -, z, w
5       | Mk3 aaa -- contexto: x, y, -, w
6       | Mk4 aaa -- contexto: x, y, z, -
7
8 dados QuadTwoContexts a = Mk (QuadContexts a) (Inverso (Ou aa))
```

Podemos então usar os inversos, como na ilustração acima, para mapear de um buraco 2 e um contexto de volta ao *ÿ* original 4 tipo, implementando a ilustração da equação (7):

5:12 Como obter o inverso de um tipo

```
fromContext :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
-- Nos dois primeiros casos, colocamos o buraco 2 no início da tupla 4.
fromContext (h1, h2) (Mk (Mk1 yzw) inv) =
    letUnit (inv (Right y)) (h1, h2, z, w) -- consome y então preenche 2 buracos

fromContext (h1, h2) (Mk (Mk2 xzw) inv) =
    letUnit (inv (Esquerda x)) (h1, h2, z, w) -- consome x então preenche 2 buracos

-- Nos dois segundos casos, colocamos o buraco 2 no final da tupla 4.
fromContext (h1, h2) (Mk (Mk3 xyw) inv) =
    letUnit (inv (Direita w)) (x, y, h1, h2) -- consome w então preenche 2 buracos

fromContext (h1, h2) (Mk (Mk4 xyz) inv) =
    letUnit (inv (Esquerda z)) (x, y, h1, h2) -- consome z então preenche 2 buracos
```

A intuição é que o inverso *(2ÿ)* (inv acima) é usado para consumir um elemento da 4-tupla que se sobrepõe ao buraco, com o construtor de Both delineando de qual posição estamos consumindo.

Essa técnica se torna mais útil quando queremos contextos de 2 buracos em um tipo que não contém um número par de elementos – ou mais geralmente quando queremos contextos de n buracos de um tipo de dados cujo número de elementos não é exatamente divisível por n. Por exemplo, agora podemos calcular o tipo de 5 tuplas com 2 furos como:

$$\ddot{y}(\ddot{y}2)\ddot{y}^{5} \; 4 = 5\ddot{y} \quad \ddot{y}\,(2\ddot{y}\;\ddot{y}\;1)$$

A interpretação usual no domínio real teria gerado interpretação em $\frac{5}{3}\ddot{y}^{2}$ para o qual não temos tipos regulares. Em vez disso, podemos usar a abordagem inversa para produzir contextos de 2 buracos para 5 tuplas. O equivalente resultante de fromContext então tem que capturar um buraco final que se sobrepõe ao precedente, para lidar com o fato de que 5 não é fatorado por 2.

Uma possibilidade ainda mais interessante se apresenta, no entanto. Observe que o exemplo acima de contextos de 2 furos para 4 tuplas considera o contexto também dividido em pares contíguos e, portanto, não podemos ter o contexto *(x,* ÿ1, ÿ2, w) com o 2 furo “em o meio". No entanto, essa interpretação certamente deve ser possível usando a abordagem inversa, pois há informações suficientes disponíveis: no domínio de R, a divisão (multiplicação por um inverso) é uma operação não injetora (destrói informações), enquanto com tipos regulares, o inverso preserva a estrutura do tipo original até aplicarmos a divisão. De fato, podemos definir a seguinte maneira alternativa de mapear o tipo de dados QuadTwoContexts de volta para uma tupla de 4:

```
fromContext' :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
fromContext' (h1, h2) (Mk (Mk1 yzw) inv) =
    letUnit (inv (Esquerda h1)) (h2, y, z, w) -- primeiro buraco fora da 4-tupla!

fromContext' (h1, h2) (Mk (Mk2 xzw) inv) =
    letUnit (inv (Esquerda x)) (h1, h2, z, w) -- 2 furos no início da 4 tupla

fromContext' (h1, h2) (Mk (Mk3 xyw) inv) =
    letUnit (inv (Esquerda y)) (x, h1, h2, w) -- 2 furos no meio da 4 tupla

fromContext' (h1, h2) (Mk (Mk4 xyz) inv) =
    letUnit (inv (Esquerda z)) (x, y, h1, h2) -- 2 furos no final da 4 tupla
```

Em comparação com fromContext, isso “desloca” o 2-hole através de posições sucessivas do contexto, não exigindo que o contexto circundante seja dividido em pares. Em vez disso, fromContext' usa o inverso para consumir o valor extra sempre sob o componente esquerdo do buraco, deixando os quatro elementos restantes da seguinte forma:

h1(h2, y, z, w) *(xh* ÿ *1,* h2, *z,* w) *(x,* ÿ *y* h1, h2, w) *(x, y,* ÿ *z* h1, h2)

com o elemento consumido mostrado aqui tachado. Observe em particular o primeiro caso (Mk1) em que o elemento excluído está “fora” da 4-tupla; se tivermos um buraco 2 onde o segundo buraco está na posição mais à esquerda de uma tupla 4, então o primeiro buraco refere-se aos dados (h1) fora da tupla 4 à esquerda. Podemos entender esses dados como sendo um “valor de limite”, o que sugere um aplicativo para tipos de dados de contexto n-hole: *cálculos de estêncil.*

n-buracos e cálculos de estêncil

Normalmente aplicado a matrizes, uma computação de estêncil percorre cada posição em uma matriz, lendo uma pequena vizinhança de elementos na posição “atual” e ao redor dela para calcular o elemento correspondente de uma nova matriz. Isso é usado, por exemplo, para processamento de imagens (por exemplo, desfoque gaussiano), autômatos celulares e o método de diferenças finitas para resolver PDEs [25]. A ideia pode ser generalizada para outros tipos além de arrays, por exemplo, árvores, grafos e malhas triangulares [41]. A noção acima de uma derivada generalizada em relação a outro tipo e sua interpretação como uma espécie de zíper em n-buracos captura exatamente essa estrutura. No exemplo acima de), o 2-^{2ÿ} ÿ *ÿÿ2 (ÿ* ^{4}hole descreve uma vizinhança de dois elementos, olhando neste caso “à esquerda” (em outras palavras, a vizinhança compreende o elemento atual e um à sua esquerda na interpretação acima). No caso Mk1 acima, o valor h1 é o valor limite quando o segundo furo é posicionado no ponto mais à esquerda. Isso é padrão para cálculos de estêncil: precisamos de um “halo” de valores de limite para calcular na borda de uma estrutura de dados.

Como um exemplo mais concreto, considere o operador Laplace discreto sobre arrays unidimensionais . Matematicamente, podemos descrever a operação como pegar uma matriz *A* e calcular os elementos na posição *i* na matriz de saída *B* da seguinte forma:

Bi = Aiÿ1 ÿ *2Ai* + Ai+1

(Observe que isso ignora o que fazer nos limites da matriz nas posições A-1 e An+1).
Isso pode ser estruturado usando uma computação local em 3 furos, por exemplo, em Haskell:

```
1 laplace :: (Float, Float, Float) %1 -> Float 2 laplace (a,
b, c) = a - 2*b + c
```

Podemos então capturar o tipo de dados de contextos para a travessia global correspondente de uma estrutura de dados (digamos, listas) calculando os contextos de 3 furos, por exemplo,

ÿÿ3 (Lista ÿ) = ÿÿ(Lista ÿ) ÿ (3ÿ ^{2} ÿ 1)

o que nos dá o tipo de dados usual de contextos de 1 buraco mais uma maneira de consumir 2 elementos, gerando uma lacuna para 3 buracos. Uma função recursiva pode então navegar “à direita” através desta estrutura de dados, aplicando laplace a cada 3 furos para calcular os valores de uma lista de saída, dando uma iteração da computação discreta do estêncil Laplace. A definição real dessa operação é menos relevante para tipos inversos, então a omitimos aqui, mas a incluímos no artefato de código que a acompanha para esta pérola. O inverso é necessário para mapear essa estrutura de dados do zíper de volta ao formulário de lista original, por meio de uma operação semelhante a fromContext'.

Assim, nossa noção de inverso multiplicativo nos deu uma maneira de generalizar derivadas para n-buracos, que podem ser usadas para capturar a “janela deslizante” de uma computação de estêncil por meio de qualquer tipo de dados.

5:14 Como obter o inverso de um tipo

6 Comunicando-se com os Inversos

Até agora, os habitantes dos inversos $ÿ^{ÿ1}$ têm sido bastante mundanos, consumindo suas entradas por correspondência de padrões. Agora nos voltamos para um cenário mais rico em que alguns tipos têm um inverso habitado por funções que podem realizar algum tipo de efeito colateral local. Para isso, usamos a linguagem funcional Granule, que combina tipos lineares e indexados com tipos modais graduados [42], embora vamos aproveitar principalmente apenas tipos lineares aqui. Consideramos inversos mais ricos primeiro por meio da questão de saber se nossa noção de inversos é uma *involução.*

Uma função é uma involução se for sua própria inversa, ou seja, f(f(x)) = x. Na álgebra abstrata, os inversos em grupos e corpos são automaticamente involuções.5 Na configuração de Granule, o inverso de um tipo também é uma involução com isomorfismo $(ÿ^{ÿ1})^{ÿ1} \cong ÿ$, o que pode ser surpreendente, visto que até agora nosso inverso tem sido frouxo.

Uma direção do isomorfismo de involução $ÿ ÿ (ÿ^{ÿ1})^{ÿ1}$ é fácil através da aplicação de funções. Damos a definição abaixo no Granule, cuja sintaxe é muito parecida com a do Haskell:

```
1 tipo Inverso a = a ÿ ()
2
3 -- ou seja, o tipo se expande para a ÿ ((a ÿ ()) ÿ ()) 4 invol : ÿ
{a : Tipo} . a ÿ Inverso (Inverso a) 5 invol x = ÿf ÿ fx
```

Como se pode ver, as diferenças entre Granule e Haskell são bastante mínimas: as setas do Granule são lineares por padrão (fãs de pirulitos ÿ terão apenas que apertar os olhos nas amostras de código!) Enquanto na extensão de tipos lineares de Haskell a multiplicidade linear deve ser explicitamente escrita. O restante da tradução de Haskell para Granule reside principalmente em quantificar explicitamente nossos tipos e usar : em vez do :: de Haskell para nossas declarações de tipo. Se o leitor quiser acompanhar usando o Granule para esta seção, recomendamos a versão mais recente.6

Útil para esta pérola, o Granule implementa um mecanismo para derivar algoritmicamente uma operação de enfraquecimento para tipos regulares [23]; isso pode ser acessado escrevendo drop @t para algum tipo t, então temos, por exemplo, drop @Bool : Bool ÿ (), fornecendo um inverso.

Esta direção do isomorfismo de involução também está bem definida no cálculo ÿ linear e em Haskell. A direção oposta $(ÿ ÿ ÿ)^{ÿ1}$ é muito mais desafiadora: na verdade ela não é habitada se nos restringirmos ao ÿ-cálculo linear ou mesmo tradicional

Haskell não linear, mas em vez disso é uma *função realizável sequencialmente* [32, 33].

Uma função sequencialmente realizável é aquela que tem um comportamento externo puro, mas se baseia em uma noção de efeitos colaterais locais; estes estão inteiramente contidos no corpo da função. Tradicionalmente, as funções realizáveis sequencialmente só podem ser expressas na família de linguagens ML (que permite efeitos colaterais irrestritos), mas não em Haskell. No entanto, no contexto da digitação linear, agora podemos reintroduzir com segurança alguns efeitos colaterais por meio de referências lineares ou canais lineares. Mostramos a última abordagem, aproveitando os canais lineares tipados por sessão do Granule [35] inspirados no cálculo GV [54]. A mesma abordagem funcionaria igualmente bem em outras implementações de cálculos semelhantes, como Fst [31] ou FreeST 2 [3].

Originário de Gay e Vasconcelos [17], posteriormente desenvolvido por Wadler [54], para o qual usamos a formulação de Lindley e Morris [30], o cálculo GV estende o cálculo ÿ linear com um tipo de canais parametrizados por tipos de sessão [57], que captam o protocolo de interação permitido no canal. Granule fornece um tipo análogo de canais lineares LChan : Protocolo ÿ Tipo indexado por protocolos, fornecido pelos construtores:

---

5 Para um grupo $(X, \bullet, e)$ então as propriedades dos inversos rendem $(x^{ÿ1})^{ÿ1} \bullet x^{ÿ1} = e$ o que implica $(x^{ÿ1})^{ÿ1} \bullet x^{ÿ1} \bullet x = x$; assim, pelas mesmas propriedades temos $(x^{ÿ1})^{ÿ1} = x$.

6 Exemplos foram testados em https://github.com/granule-project/granule/releases/tag/v0.8.1.0

Enviar: Digite ÿ Protocolo ÿ Protocolo Fim: Protocolo
Recv: Tipo ÿ Protocolo ÿ Protocolo

As seguintes primitivas são fornecidas para usar esses canais (síncronos):

```
enviar : ÿ {a : Tipo, s : Protocolo} . LChan (Enviar como) ÿ a ÿ LChan s
recv : LChan (Recv como) ÿ (a, LChan s)
fechar : LChan Fim ÿ ()
forkLinear : ÿ {s : Protocolo} . (LChan s ÿ ()) ÿ LChan (Dual s)
```

Este é um subconjunto das primitivas disponíveis. Podemos ver que send recebe um canal de entrada com o protocolo Send as e um valor de entrada a que é enviado pelo canal para gerar um canal que pode ser usado de acordo com o protocolo s. A função recv é dual, pegando um canal que pode seguir o protocolo Recv a s, retornando um par do valor a recebido e um novo canal que pode se comportar como s. A primitiva de fechamento consome um canal que está no final de um protocolo. Por fim, forkLinear gera um processo a partir da função de parâmetro, que é aplicado a um canal recém-criado, retornando um canal com o protocolo "dual" para se comunicar com o novo processo. Aqui, Dual é uma função de nível de tipo definida:

```
Dual (Enviar como) = Recv a (Dual s)
Dual (Recv as) = Enviar um (Dual s)
Extremidade Dupla = Extremidade
```

Curiosamente, forkLinear é um combinador relacionando inverso e dualidade: (LChan s)^{ÿ1} ^{ÿ} LChan (s ÿ), ou seja, uma função que consome um canal com comportamento s produz um canal com comportamento dual (denotado pela notação padrão ÿ como na lógica linear e no cálculo GV).

Agora temos maquinário adequado para definir a involução na direção *(ÿ* ^{ÿ1} *)*^{ÿ1} $^{ÿ ÿ :}$

```
1 involOp : ÿ {a : Tipo} . Inverso (Inverso a) ÿ a 2 involOp k =
3     deixe r = forkLinear (ÿs ÿ k (ÿx ÿ deixe c = enviar sx no fechamento c)); (x, c') = recv r; () =
4         fechar c'
5
6     em x
```

Assim k tem tipo (a ÿ ()) ÿ (), para o qual é passada a função ÿx ÿ let c = send sx in close c ; isso envia a entrada x : a no canal c que é então fechado. Esse canal é fornecido por forkLinear e, portanto, k é aplicado em um processo que leva uma extremidade do canal para "esgueirar-se" do valor do tipo a. Fora disso, recv espera receber da extremidade dupla do canal retornado por forkLinear. O canal restante é fechado e x é retornado. Um efeito colateral local é executado dentro do involOp que não é observável externamente, mas não teria sido possível construir a função necessária sem realizar esse efeito.

Em linguagens como ML, uma definição equivalente alternativa pode ser dada usando mutável referências que também se assemelham ao combinador **F** de Longley [32].

Por fim, as funções invol e involOp formam um isomorfismo, testemunhando *ÿ* ÿ= *(ÿ* A ^{ÿ1} *)*^{ÿ1} . prova disso pode ser demonstrada por meio de cálculos sobre as definições, com detalhes dados no Apêndice B. Remetemos o leitor a Lindley et al. [30, 31] para obter detalhes de como adicionar canais lineares digitados por sessão ao cálculo ÿ linear (e Sistema linear F) mantém a segurança de tipo.

Tipos de sessão baseados em canais lineares também podem ser representados em Haskell, mas não nos permitem demonstrar um isomorfismo completo na mesma extensão. Aqui ilustramos a direção mais desafiadora da involução usando a biblioteca Priority Sesh,[7] um pacote recente para

[7] Disponível em https://github.com/wenkokke/priority-sesh

5:16 Como obter o inverso de um tipo

comunicação tipada por sessão em Linear Haskell que é ela própria inspirada no cálculo GV [27]. No entanto, observe que as duas direções não podem formar uma involução aqui, uma vez que tudo deve ser agrupado na mônada IO linear para que esses tipos de sessão sejam usados; não podemos limitar os efeitos colaterais à função, como é possível no Granule.

```
digite Inverse' a = a %1 -> Linear.IO ()

involOp :: Inverse' (Inverse' a) %1 -> Linear.IO a involOp k = do (s, r) <- new void $ forkIO $ k (\z
        -> send sz)

        recv r
```

mônada de continuação

Um tipo duplo inverso *(ÿ =ÿ(ÿ)ÿ* 1) ÿ 1 também é uma especialização da conhecida *mônada de continuação* [52], cujo tipo de retorno é o tipo de unidade (o tipo de dados Haskell geralmente é escrito data Cont ra = Cont ((a -> r) -> r) então este é Cont () a aqui). A definição de invol fornece a operação de retorno da mônada e o operador “bind” é a definição usual para a mônada de continuação:

```
retorno : ÿ {a : Tipo} . a ÿ Inverse (Inverse a) return = invol


ligação: ÿ {ab: tipo}
                . Inverso (Inverso a) ÿ (a ÿ Inverso (Inverso b)) ÿ Inverso (Inverso b) bind mk = ÿc ÿ m (ÿa ÿ kac)
```

Uma maneira padrão de entender o uso da mônada de continuação é ver que suas setas de Kleisli (funções do tipo a ÿ Inverse (Inverse b)) caracterizam programas de estilo de passagem de continuação (CPS) que podem então ser compostos sequencialmente. Isso pode ser visto por meio de uma pequena manipulação algébrica:

a ÿ Inverso (Inverso b) ÿ a ÿ ((b ÿ ()) ÿ ()) ÿ= (b ÿ ()) ÿ (a ÿ ())

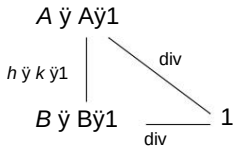
Assim, podemos ver que as setas de Kleisli são funções transformadas por CPS de *“a* ÿ b”. Sob nossa interpretação, essas são as mesmas funções que mapeiam consumidores de *b* para consumidores de a, e a mônada “inversa dupla” nos dá uma composição sequencial para esses inversos.

A maneira usual de “avaliar” uma computação de mônada de continuação é terminar com um valor do tipo Cont rr (ou seja, (r -> r) -> r) ao qual a identidade é aplicada para retornar o valor “final” do tipo r. Isso requer que o tipo r de toda a computação de Cont r seja pré-determinado com base no valor que queremos ser capazes de extrair de uma computação de mônada de continuação. Para Inverse (Inverse a) só podemos aplicar a identidade quando a = (), ou seja, apenas em casos triviais. No entanto, nossa mônada “inversa dupla” é realmente mais poderosa graças à linearidade e à capacidade de realização sequencial: podemos extrair o valor de qualquer cálculo Inverso (Inverso a) para qualquer a aplicando a função de involução sequencialmente realizável involOp para extrair o valor a . Isso acaba sendo mais flexível do que a mônada de continuação, pois não precisamos pré-determinar o tipo de “resultado” da continuação (que para o inverso duplo é fixado como unidade de qualquer maneira) e podemos extrair o valor dentro da computação em qualquer ponto.

Assim, ver a mônada de continuação através das lentes dos inversos lógico-lineares produz uma composição monádica de estilo de passagem de continuação mais flexível. A restrição crucial, porém, é que as continuações devem ser usadas *linearmente.*

Calculando com inversos que se comunicam

Lembre-se da propriedade de naturalidade discutida na Seção 2:



No Granule com efeitos colaterais locais devido a canais, esta equação só é válida se *k* ÿ *h* = *id.* Considere o seguinte código onde divNat captura o caminho inferior esquerdo do diagrama:

```
1 divNat : ÿ {ab : Tipo} . (a ÿ b) ÿ (b ÿ a) ÿ (a, Inverse a) ÿ () 2 divNat hk (x, y) = divide (hx) (comap
ky)
3
4 exemplo : ÿ {ab : Tipo} . (a ÿ b) ÿ (b ÿ a) ÿ a ÿ a 5 exemplo hka = let r =
forkLinear (ÿs ÿ divNat
6       hk (a, ÿy ÿ let c = enviar sy in close c));
7             (a', c') = recv r; () =
8             fechar c'
9       em um'
```

A função de exemplo aplica divNat dentro de um processo bifurcado onde o inverso envia o resultado no canal que é recebido do lado de fora. exemplo hk é apenas a função de identidade se k . h = *id,* por exemplo, exemplo (ÿxÿ x + 1) (ÿxÿ x - 1) 42 resulta em 42. Assim, podemos ver o poder dos efeitos colaterais locais; aqui os inversos podem fazer mais do que apenas consumir.

7 Inversos Aditivos

Como demonstramos, existe uma definição razoável de inversos para tipos de produto no domínio da lógica linear. Pode-se, portanto, perguntar se a definição de inversos para tipos de soma (ou seja, *subtração)* também é viável. Da mesma forma que definir um inverso multiplicativo quase nos dá um semicampo de tipos porque algumas das identidades são frouxas, ser capaz de definir um inverso aditivo da mesma forma nos daria algo próximo de um anel de tipos. A resposta sobre se podemos fazer isso, no entanto, é um tanto confusa.

Os tipos regulares lineares que temos usado têm tipos produto como a “conjunção multiplicativa” ÿ da lógica linear e tipos soma como “disjunção aditiva” ÿ. Infelizmente, não podemos definir uma inversa aditiva sensata para este operador. Para fazer isso, precisaríamos ter *A* ÿ *ÿA* ÿ= 0. No entanto, definir um mapa *A* ÿ *ÿA* ÿ 0 é impossível, independentemente do valor de *ÿA,* porque se *A* é não vazio, então *A* ÿ *B* também deve ser não vazio o que significa que não podemos construir um termo do tipo 0. Além disso, definir um mapa 0 ÿ *A* ÿ *ÿA* é impossível a menos que *A* = *ÿA* = 0, caso contrário, teríamos que ser capazes de construir algum valor do tipo *A* ou do tipo *ÿA* do nada. Consequentemente, tipos regulares lineares não podem formar um *corpo* (com inversos multiplicativos e aditivos).

Acontece que a razão pela qual não é possível definir um inverso aditivo no contexto da lógica intuicionista padrão ou usando a operação ÿ oferecida por tipos regulares lineares é um corolário de um resultado conhecido como *lema de Crolard* [5]. Este lema afirma que a subtração *A* \ *B* não pode ser definida para uniões disjuntas na categoria de conjuntos, a menos que *A* ou *B* seja o conjunto vazio. De fato, este resultado também se aplica a qualquer operação como ÿ que permite uma escolha livre entre *A* e B, então qualquer definição de subtração em relação a ÿ deve ser trivial.

5:18 Como obter o inverso de um tipo

No entanto, definir produtos como conjunção multiplicativa e somas como disjunção aditiva como em tipos regulares não é a única interpretação possível que podemos usar para esses conceitos. Podemos facilmente ter tipos de produto que seguem as regras do operador &, pronunciado “com” e descrevendo “conjunção aditiva”, e da mesma forma podemos ter tipos de soma baseados no operador ` , pronunciado “par”, que descreve “multiplicativo disjunção". Isso torna possível discutir uma configuração intimamente relacionada que chamaremos de *tipos co-regulares* (uma vez que essas operações se comportam de maneira dual àquelas usadas para definir *tipos regulares),* com sintaxe de tipo:

ÿ ::= ÿ & ÿ ÿ | *ÿ ` ÿ* ÿ | ÿ | ÿ

onde ÿ e ÿ são as unidades para & e ` respectivamente. A intuição para & é que a&b nos permite selecionar um de *a* ou *b* e usá-lo, em vez de ter acesso a ambos ao mesmo tempo, mas ter que usar cada um, como com *a* ÿ b. O operador ` é mais difícil de entender intuitivamente, mas uma interpretação é que *a ` b* dá dois processos *a* e *b* que acontecem em paralelo, e temos a escolha de como intercalar os dois processos [4]. O importante a ter em mente é que & é dual para ÿ e ` é dual para ÿ.[8]

Se considerarmos os tipos soma ` co-regulares que não permitem uma escolha livre entre *A* e *B* em vez dos tipos soma ÿ regulares, então a perspectiva para definir um inverso aditivo é menos sombria: é possível definir uma operação inversa para disjunção multiplicativa .
A lógica linear cointuicionista [5] oferece uma operação chamada *subtração linear,* denotada por *A \ B* e lida *“A exclui* B”, que atua como o adjunto esquerdo de `. Intuitivamente, podemos entender a implicação linear da seguinte maneira:

A ÿ *B* ÿ *C* se e somente se *A* ÿ *B* ÿ *C*

que surge da noção categórica de adjunções: (ÿ ÿ B) é adjunta à esquerda a *(B* ÿ ÿ).
Duplamente, a subtração linear pode ser entendida da seguinte forma:

A ÿ *B ` C* se e somente se *A \ B* ÿ *C*

Em outras palavras, se *A* nos dá *B ` C* então A *excluindo* a possibilidade de *B* nos dá C, e inversamente se *A* excluindo *B* nos dá *C* então de *A* podemos obter *B ` C*.

Neste cenário dual, devemos agora encontrar uma definição de subtração de uma unidade adequada que atue como o inverso aditivo que desejamos. Como a subtração linear é dual para a implicação linear, assim como podemos definir a implicação em termos do conectivo ` (ou seja, *A* ÿ *B* ÿ Aÿ ` B), podemos representar a subtração de maneira semelhante usando o conectivo ÿ, como *B \ A* ÿ *B* ÿ Aÿ.

Usando esta representação de subtração linear, por dualidade podemos mostrar que um inverso não trivial para disjunção multiplicativa ` existe no contexto da teoria de tipo linear. Duplamente para nossa definição de um inverso para conjunção multiplicativa, podemos definir um inverso para ÿ ÿ \ ÿ disjunção multiplicativa como *ÿÿ* ÿ *ÿ* ÿ 1, o , via subtração linear como discutido acima.

Da mesma forma que a identidade ÿ *ÿ* inverso aditivo *ÿÿ* também satisfaz um lax
lax *ÿ* lei inversa, mas na direção oposta:

ÿ ÿ *ÿ ` ÿÿ* (8)

Através da identidade entre ÿ e ` e entre \ e ÿ então *ÿ `ÿÿ* ÿ= *ÿ* tinha um sistema ÿ (ÿÿÿ ÿ). Se nós de tipos envolvendo tipos regulares e co-regulares com um operador de dualidade acima ÿ (ÿ ÿ *ÿ* ÿ). lei dada construtivamente pelo termo *ÿb.(ÿx.(b,* x)) : ÿ ÿ *ÿ* lax (8) seria

[8] A lógica linear clássica possui um operador de dualidade involutivo, escrito (ÿ) ÿ, onde *(A* & B) ÿ *B* ÿ. e *= A* ÿ ÿ *B* ÿ *= A* ÿ *(A`B)*

As aplicações dessa identidade são menos aparentes, pois não podemos construir uma testemunha para ela da mesma forma que a lei inversa frouxa da Seção 2, dadas as restrições de ter que escolher entre trabalhar com tipos regulares ou co-regulares. Se não fôssemos limitados por essa limitação, poderíamos ter uma estrutura algébrica com todas as quatro operações matemáticas comuns, com ÿ atuando como uma identidade aditiva e 1 atuando como uma identidade multiplicativa.9 Lógicas intuicionistas e cointuicionistas podem ser combinadas em uma única estrutura, conhecida como lógica bi-intuicionista, e o trabalho para dar sentido a isso através das lentes da teoria dos tipos e da teoria das categorias está em andamento [15]; isso poderia fornecer uma maneira de combinar tipos regulares e co-regulares em um único sistema.

Dadas as definições acima, podemos mostrar uma involução frouxa em uma direção para inversos aditivos. Entre a conjunção multiplicativa e a disjunção existe uma distribuição: *(A* ÿ *(B* ` C)) ÿ ((A ÿ B) ` C) que não é um isomorfismo, mas é uma implicação válida nesse sentido [13]. Usando esta distribuição fraca, para todo *ÿ* esta involução frouxa é definida:

ÿ(ÿÿ) ÿ= (ÿ \ (ÿ \ *ÿ)) ÿ*= ÿ
ÿ (ÿ ÿ *ÿ* ÿ) ÿ ÿ= ÿ ÿ
(1 ` *ÿ)* ÿ (ÿ ÿ 1)
` *ÿ* ÿ= ÿ ` *ÿ*

ÿ= *ÿ*

Curiosamente, esse tipo de dicotomia entre disjunção aditiva e multiplicativa também pode ser vista para a conjunção. O inverso multiplicativo que definimos para tipos regulares lineares não se comporta bem se tentarmos aplicá-lo ao operador & (conjunção aditiva). *Não podemos* definir um mapa *A* & *(A* ÿ 1) ÿ 1, porque só podemos usar um dos dois componentes do & à esquerda, portanto não podemos aplicar o inverso ao valor *A.* Além disso, da mesma forma que ÿ, não podemos em geral definir um mapa na outra direção, pois precisaríamos ser capazes de construir um valor de um tipo arbitrário *A* a partir do nada.

No final, a lógica linear ainda não pode nos dar um campo de tipos – ela só pode nos dar um anel ou meio campo (um semicampo), mas ambos ao mesmo tempo estão além do nosso orçamento atual. A interpretação do semicampo, no entanto, tem as intuições mais próximas dos conceitos familiares na programação funcional, e os tipos regulares lineares são certamente mais frequentemente encontrados do que os co-regulares no cenário de programação atual, daí nosso foco neles nesta pérola.

8 Discussão: Pensando com Inversos

Ao nos aproximarmos do final de nossa jornada, comentamos sobre algumas perspectivas e abordagens alternativas e algumas conexões com trabalhos relacionados.

Curry-Howard com inversos

Do ponto de vista lógico, a propriedade inversa (frouxa) que discutimos fornece uma noção natural de eliminação inversa e introdução em uma lógica de dedução natural para um correspondente de Curry-Howard ao inverso de um tipo:

9 Ainda não podemos, entretanto, formar um corpo mesmo se usarmos ÿ e ` como nossas operações; eles não obedecem à distributividade, *A* ÿ ÿ ÿÿ= ÿ (observe, por exemplo, que ÿ ÿ ÿ ÿ= ÿ) e, de fato, ambos os tipos de inverso obedecem apenas a leis inversas frouxas, de modo que os isomorfismos necessários para um corpo não existem.

5:20 Como obter o inverso de um tipo

$$\frac{\ddot{y}\ \ddot{y}\ p\ \ddot{y}\ \ddot{y}\ p\ \ddot{y},\ \ddot{y}\ddot{y}}{\ddot{y}\ 1}^{\ddot{y}1}{}_{\ddot{y}1}\ E \qquad \frac{\ddot{y},\ p\ \ddot{y}\ 1}{\ddot{y}\ \ddot{y}\ p\ddot{y}1}^{\ddot{y}1}{}_{EU}$$

ou seja, a eliminação é apenas um modus ponens especializado e um p inverso $\ddot{y}1$ é introduzido por uma prova (linear) começando com p e concluindo 1 – a subprova consome a suposição p.

Dualidade

Pode-se considerar tentar usar a noção de lógica linear clássica de "dualidade" $\ddot{y}$ [20] para fornecer inversos multiplicativos, mas ela não se comporta de acordo: $1 \ddot{y} 1 = 1 \ddot{y} \ddot{y} = \ddot{y}$ mas, em vez disso, $\ddot{y}1 = \ddot{y}$ 1 ÿ 1 dualidade 1. No entanto, existem várias aplicações interessantes de linear e gostaríamos de clássica relacionando chamada por valor e chamada por nome [53, 45]. Eles tiram vantagem de várias propriedades da dualidade, algumas das quais nossos inversos realmente compartilham.

Tipos negativos e fracionários

Apesar da manipulação algébrica de tipos de dados produzir uma rica fonte de ideias, os inversos parecem não ter tido muita consideração. Um tópico notável, porém, é devido a James e Sabry, que consideram tipos *negativos* e *fracionários* no contexto de computações reversíveis onde um $1/b$ recíproco *"impõe restrições em [seu] contexto"* agindo como uma variável lógica [24]. Eles apresentam um cálculo reversível admitindo isomorfismos $\ddot{y} : 1 \ddot{y} (1/b)\times b$ para todos os tipos: com direção da esquerda para a direita produzindo uma nova variável lógica $\ddot{y}$ habitando b junto com seu dual, e o inverso $\ddot{y}$ correspondendo à unificação de variáveis lógicas. Mais tarde eles interpretam esta semântica categórica computacionalmente [12], definindo uma semântica operacional sólida para tais tipos, em que um tipo negativo representa um efeito computacional que "reverte o fluxo de execução" e um tipo fracionário representa aquele que "coleta de lixo" valores ou lança exceções .
Isso difere da nossa abordagem, mas certamente tem um pouco do mesmo sabor. No entanto, não podemos construir um par de a $b^{\ddot{y}1}$ $\ddot{y}$ b fora do ar para qualquer b.

Cardinalidades

Conforme lembrado na introdução, a operação de cardinalidade em tipos é um homomorfismo de semi-anel de tipos regulares para números naturais (por exemplo, $|a \times b| = |a||b|$). Então, qual é a cardinalidade de um tipo inverso? Em uma configuração cartesiana, uma função $\ddot{y}$ ÿ 1 simplesmente tem cardinalidade1, pois $|\ddot{y} \ddot{y} 1| = |1| |\ddot{y}| = 1$. Em uma configuração linear sem efeitos colaterais (ou seja, canais lineares), podemos recuperar um resultado semelhante. Como um exemplo simples, considere o tipo booleano. A cardinalidade de Bool é 2, pois possui dois elementos: True e False. A cardinalidade de Inverse Bool, por outro lado, é 1; o tipo tem exatamente um habitante: a função boolDrop mostrada na Seção 2.

Poderíamos, no entanto, considerar uma noção diferente de cardinalidade que permitiria $|\ddot{y}\ a\ |\times|\ddot{y}^{b}| = |\ddot{y}^{a+b}|$ em geral; esta afirmação já vale para a ÿ 0 ÿ b ÿ 0, mas agora consideramos casos onde os tipos não são necessariamente isomórficos. Em particular, podemos examinar a noção de *cardinalidades fracionárias* [44, 46] atribuindo uma cardinalidade generalizada de $(1/|\ddot{y}|)$ m Se para $\ddot{y}$ $\ddot{y}m$.

especializarmos isso, podemos fazer $a = 1$ e b = ÿ1 e ver que $|\ddot{y} \ddot{y} \ddot{y}^{\ddot{y}1}| = |\ddot{y}| \times |\ddot{y}^{\ddot{y}1}| = n \times (1/n) = 1 = |1|$; os dois tipos têm a mesma cardinalidade fracionária, embora tenhamos apenas um mapa lax de $\ddot{y} \ddot{y} \ddot{y}$ a 1. Claro, isso não corresponde à ideia padrão de cardinalidade em tipos, pois é claro que $\ddot{y} \ddot{y} \ddot{y}$ tem pelo menos menos tantos habitantes quanto $\ddot{y}$. Deixamos uma interpretação para isso como algo para outros ponderarem.

série taylor

Como vimos, não é possível formar um corpo a partir de tipos regulares (lineares ou não), pois a operação aditiva que permite uma inversa não é a mesma adição que se comporta como o ÿ lógico que normalmente desejaríamos. Mas acontece que, se suspendermos nossa descrença e assumirmos que os tipos formam um corpo, alguns resultados da análise real podem ser aplicados com sucesso surpreendente: as aproximações da série de Taylor podem produzir soluções para tipos recursivos.

Considere a definição recursiva de listas sobre elementos do tipo ÿ:

Lista *ÿ* = 1 ÿ *(ÿ* ÿ Lista ÿ)

Através de algum rearranjo algébrico injustificado, obtemos a Lista *ÿ* = calcula a expansão ~~1 em~~ que podemos (1ÿÿ) de Taylor produzindo a solução familiar de ponto fixo mínimo da Lista ÿ:

Lista *ÿ* = 1 ÿ *ÿ* ÿ *ÿ* $\frac{2}{ÿ}$ $\frac{3}{ÿ}$ ÿ *ÿ* ÿ . . .

ou seja, uma lista está vazia, ou tem um elemento, ou tem dois elementos, etc.

Isso é bastante surpreendente; devemos aplicar as equações de um corpo para obter uma derivação para esse resultado, usando inversas às quais não temos acesso no reino dos tipos regulares e, no entanto, acabamos com um resultado que faz sentido usando apenas operações regulares. Não está claro se existe uma interpretação de nossos inversos que possa fornecer uma base significativa para essas manipulações intermediárias, mas certamente seria interessante investigar.

Alguém pode se perguntar se é coincidência que esse resultado seja verdadeiro para listas em particular, mas esse não é o caso. Fiore e Leinster [16] mostram que, para todos os números complexos *t* e polinômios p, q1 e q2 com coeficientes não negativos (com algumas restrições), então se *t = p(t)* implica q1(t) = q2(t) o mesmo resultado também se aplica ao isomorfismo em qualquer outro semi-anel (bem como para números complexos), que inclui tipos regulares.

Isso foi aplicado com grande efeito para o exemplo de árvores binárias finitas para demonstrar o famoso resultado “sete árvores em uma” [9], mostrando que há uma bijeção particularmente elementar (envolvendo distinções de caso apenas até uma profundidade fixa) entre o conjunto *T* de árvores binárias finitas e o conjunto *T* de 7-tuplas dessas árvores. É mais difícil encontrar soluções para tipos mais complexos por meio desse tipo de raciocínio equacional, principalmente devido ao teorema de Abel-Ruffini [2] que afirma que não há solução em radicais para equações polinomiais gerais de grau cinco ou superior.

9 Epílogo

Resumo

Tomando *ÿ ÿ* $\frac{ÿ1}{ÿ}$ produz uma noção útil de inverso multiplicativo para tipos regulares lineares.
Vimos que isso produz leis de exponenciação (frouxas) na presença de coeficientes negativos:

ÿ *ÿ* $ÿ^{1}$ ÿ 1 *ÿ* $ÿ^{um\ ÿ}$ ÿ *ÿ* $ÿb^{ÿ}$ $aÿb^{ÿ}$ $ÿa^{ÿ}$ $ÿb^{ÿ}$ ÿ(a+b) ÿ= *ÿ* ÿ

$ÿa^{ÿ}$ ÿ *ÿ* $ÿ^{ÿa}$ ÿ *(ÿ* ÿ *ÿ*) *ÿa* *ÿ* ÿ *(ÿ* $ÿ^{ÿ1}$) $ÿ^{ÿ1}$

para todo *a* ÿ *0, b* ÿ 0. A primeira identidade lax é generalizada pela segunda (Seção 4). O quarto é induzido por ÿÿ1 ser um funtor monoidal (Seção 3). A última identidade lax torna-se an quando funções sequencialmente realizáveis são permitidas isomorfismo *ÿ* ÿ= *(ÿ* $ÿ^{ÿ1}$) $ÿ^{ÿ1}$ (Seção 6).

5:22 Como obter o inverso de um tipo

barbatana

As características algébricas dos tipos de dados foram estudadas e aproveitadas desde o surgimento da programação funcional; nós os chamamos de tipos de dados “algébricos”, afinal. No cenário linear, a ideia de consumo como um inverso multiplicativo frouxo nos deu uma nova perspectiva sobre a caracterização algébrica de tipos lineares regulares. Agora que a digitação linear está se tornando mais comum, por exemplo, em Haskell [6], e com ideias intimamente relacionadas surgindo em linguagens como Clean e Rust (o conceito de singularidade que é, em certo sentido, dual à linearidade [21, 14, 36], e sistemas mais sofisticados de rastreamento de propriedade e empréstimo [56]), agora é o momento ideal para começar a levar nossa compreensão algébrica dos tipos de dados para o próximo nível. Esta pérola tem sido uma demonstração de como um truque estranho pode levar a uma viagem por muitas áreas interessantes e diversas do nosso campo. Esperamos que isso tenha alimentado seu entusiasmo em investigar a ideia de levar o inverso de um tipo ainda mais longe.

Referências

1 Michael Abbott, Thorsten Altenkirch, Conor Mcbride e Neil Ghani. *ÿ* para dados: Differenti estrutura de dados. *fundam. Inf.,* 65(1–2):1–28, janeiro de 2005.

2 Niels Henrik Abel. Memorize as equações algébricas, ou seja, ao demonstrar a impossibilidade de resolução da equação geral de cinco graus. 1:28–33, 1824. doi:10.1017/CBO9781139245807.004.

3 Bernardo Almeida, Andreia Mordido, Peter Thiemann e Vasco T. Vasconcelos. polimórfico tipos de sessão sem contexto, 2021. arXiv:2106.06658.

4 Federico Aschieri e Francesco A. Genco. Par significa paralelo: Provas de lógica linear multiplicativa como programas funcionais concorrentes. *Proc. Programa ACM. Lang.,* 4(POPL), dezembro de 2019. doi:10.1145/3371086.

5 Gianluigi Bellin, Massimiliano Carrara, Daniele Chiffi e Alessandro Menti. Interpretações pragmáticas e dialógicas do bi-intuicionismo. Parte I. *Lógica e Filosofia Lógica,* 23(4):449–480, 2014.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones e Arnaud Spiwack. Haskell linear: linearidade prática em uma linguagem polimórfica de ordem superior. *Proceedings of the ACM on Programming Languages,* 2(POPL):1–29, 2017.

7 Richard Bird e Oege de Moor. *Álgebra de Programação.* Prentice-Hall, Inc., EUA, 1997.

8 Richard S. Bird. Identidades algébricas para cálculo de programas. *The Computer Journal,* 32(2):122–126, 1989.

9 Andreas Blass. Sete árvores em uma. *Journal of Pure and Applied Algebra,* 103(1):1–21, 1995.

10 Edwin Brady. Idris 2: Teoria de tipos quantitativos na prática. Em Anders Møller e Manu Sridharan, editores, *35ª Conferência Europeia sobre Programação Orientada a Objetos (ECOOP 2021),* volume 194 de *Leibniz International Proceedings in Informatics (LIPIcs),* páginas 9:1– 9:26, Dagstuhl, Alemanha, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ECOOP.2021.9.

11 Jacques Carette e Amr Sabry. Computação com semirings e grupóides de rig fracos. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632,* pages 123–148, Berlin, Heidelberg, 2016. Springer-Verlag.

12 Chao-Hong Chen e Amr Sabry. Uma interpretação computacional de categorias fechadas compactas: Programação reversível com tipos negativos e fracionários. *Proc. Programa ACM. Lang.,* 5(POPL), janeiro de 2021. doi:10.1145/3434290.

13 JRB Cockett e RAG Seely. Categorias fracamente distributivas. *Jornal dos Puros e Aplicados Algebra,* 114(2):133–173, 1997. doi:10.1016/0022-4049(95)00160-3.

14 Edsko de Vries, Rinus Plasmeijer e David M Abrahamson. Digitação de exclusividade simplificada. No *Simpósio sobre Implementação e Aplicação de Linguagens Funcionais,* páginas 201–218. Springer, 2007. doi:10.1007/978-3-540-85373-2_12.

15 Harley Eades III e Gianluigi Bellin. Uma lógica adjunta cointuicionista, 2017. arXiv:1708.05896.

16 Marcelo Fiore e Tom Leinster. Objetos de categorias como números complexos. *Avanços Mathematics,* 190(2):264–277, 2005. doi:10.1016/j.aim.2004.01.002.

17 Simon J. Gay e Vasco Thudichum Vasconcelos. Teoria de tipo linear para tipos de sessão assíncrona. *J. Função. Program.,* 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

18 Jeremy Gibbons. Cálculo de programas funcionais. Em *Métodos Algébricos e Coalgébricos na Matemática da Construção de Programas,* páginas 151–203. Springer, 2002.

19 Jeremy Gibbons. A escola de Squiggol - Uma história do formalismo de Bird-Meertens. Em *Métodos Formais. FM 2019 International Workshops - Porto, Portugal, 7-11 de outubro de 2019, Revised Selected Papers, Parte II,* páginas 35–53, 2019. doi:10.1007/978-3-030-54997-8_2.

20 Jean-Yves Girard. Lógica linear. *Ciência teórica da computação,* 50(1):1–101, 1987.

21 Dana Harington. Lógica da unicidade. *Ciência teórica da computação,* 354(1):24–41, 2006.

22 Gérard Huet. O zíper. *Journal of Functional Programming,* 7(5):549–554, 1997.

23 Jack Hughes, Michael Vollmer e Dominic Orchard. Derivando leis distributivas para tipos lineares graduados. Em *TLLA/ Linearidade,* 2020.

24 Roshan P James e Amr Sabry. As Duas Dualidades da Computação: Negativa e Fracionária Tipos. Relatório técnico, Universidade de Indiana, 2012.

25 Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf e Katherine Yelick. Otimizações implícitas e explícitas para cálculos de estêncil. Em *Proceedings of the 2006 Workshop on Memory System Performance and Correctness,* páginas 51–60, 2006.

26G Maxwell Kelly. Teoremas de coerência para álgebras laxas e para leis distributivas. Em Seminário *de categoria ,* páginas 281–375. Springer, 1974.

27 Wen Kokke e Ornela Dardha. Tipos de sessão sem deadlock em Haskell linear. Em *Processos do 14º Simpósio Internacional ACM SIGPLAN sobre Haskell,* páginas 1–13, 2021.

28 Serge Lang. *Álgebra.* Springer, Nova York, NY, 2002.

29 Gottfried Wilhelm Leibniz. Nova methodus pro maximis et minimis, itemque tangentibus, qua nec irracionais quantifica moratur. *Acta eruditorum,* 1684.

30 Sam Lindley e J Garrett Morris. Uma semântica para proposições como sessões. No Simpósio *Europeu de Linguagens e Sistemas de Programação,* páginas 560–584. Primavera, 2015.

31 Sam Lindley e J Garrett Morris. Tipos de sessão funcionais leves. *Tipos Comportamentais: da teoria às ferramentas. River Publishers,* páginas 265–286, 2017.

32 John Longley. Quando um programa funcional não é um programa funcional? Em *ACM SIGPLAN Avisos,* volume 34(9), páginas 1–7. ACM, 1999.

33 John Longley. Os funcionais sequencialmente realizáveis. *Ana. Puro Aplic. Log.,* 117(1-3):1–93, 2002. doi:10.1016/S0168-0072(01)00110-5.

34 Saunders Mac Lane. *Categorias para o trabalho matemático,* volume 5. Springer Science & Business Media, 2013.

35 Daniel Marshall e Dominic Orchard. Replique, reutilize, repita: capturando comunicação não linear por meio de tipos de sessão e tipos modais graduados. *Proceedings of PLACES 2022, Electronic Proceedings in Theoretical Computer Science,* 356:1–11, março de 2022. doi:10.4204/ eptcs.356.1.

36 Daniel Marshall, Michael Vollmer e Dominic Orchard. Linearidade e Singularidade: Uma Entente Cordiale. Em Ilya Sergey, editor, *Programming Languages and Systems,* páginas 346–375, Cham, 2022. Springer International Publishing.

37 Conor McBride. A derivada de um tipo regular é seu tipo de contextos de um buraco. *não publicado manuscrito,* páginas 74–88, 2001.

38 Conor McBride. Palhaços à minha esquerda, curingas à direita (pérola): Dissecando estruturas de dados. *SIGPLAN Not.,* 43(1):287–295, janeiro de 2008. doi:10.1145/1328897.1328474.

39 J. Garrett Morris. O melhor dos dois mundos: programação funcional linear sem concessões. Em Jacques Garrigue, Gabriele Keller e Eijiro Sumii, editores, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japão, 18-22 de setembro de 2016,* páginas 448–461. ACM, 2016. doi:10.1145/2951913.2951925.

40 Peter Morris, Thorsten Altenkirch e Conor McBride. Explorando os tipos de árvores regulares. Em *International Workshop on Types for Proofs and Programs,* páginas 252–267. Springer, 2004.

41 Dominic Orchard. Programação de cálculos contextuais. Relatório técnico, Universidade de Cambridge, Laboratório de Computação, 2014.

42 Dominic Orchard, Vilem-Benjamin Liepelt e Harley Eades III. Raciocínio quantitativo de programas com tipos modais graduados. *Proceedings of the ACM on Programming Languages,* 3 (ICFP): 1–30, 2019.

43 Tomas Petricek, Dominic A. Orchard e Alan Mycroft. Coefeitos: um cálculo de computação dependente do contexto. Em *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, Gotemburgo, Suécia, 1 a 3 de setembro de 2014,* páginas 123–135, 2014. doi:10.1145/2628136.2628160.

44 James Propp. Medida de Euler como cardinalidade generalizada. *arXiv: Combinatória,* 2002.

45 Ben Rudiak-Gould, Alan Mycroft e Simon Peyton Jones. Haskell não é ML. No *Simpósio Europeu de Programação,* páginas 38–53. Springer, 2006.

46 Stephen H. Schanuel. Conjuntos negativos têm característica e dimensão de Euler. In Aurelio Carboni, Maria Cristina Pedicchio e Guiseppe Rosolini, editores, *Category Theory,* páginas 379–385, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

47 Rui Shi e Hongwei Xi. Um sistema de tipo linear para programação multicore em ATS. *Science of Computer Programming,* 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.

48 Kornel Szlachányi. Categorias skew-monoidais e bialgebróides. *Advances in Mathematics,* 231(3-4):1694–1730, 2012.

49 Jesse A. Tov e Riccardo Pucella. Tipos afins práticos. Em *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, EUA, janeiro 26-28, 2011,* páginas 447–458, 2011. doi: 10.1145/1926385.1926436.

50 Tarmo Uustalu, Niccolò Veltri e Noam Zeilberger. O cálculo seqüencial de monoidal de inclinação categorias. *Notas Eletrônicas em Ciência da Computação Teórica,* 341:345–370, 2018.

51Philip Wadler. Tipos lineares podem mudar o mundo! Em *Conceitos e métodos de programação,* volume 3(4), página 5. Citeseer, 1990.

52Philip Wadler. A essência da programação funcional. Em *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* páginas 1–14, 1992.

53Philip Wadler. Call-by-value é dual para call-by-name. Em *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming,* páginas 189–201, 2003.

54Philip Wadler. Proposições como sessões. *Journal of Functional Programming,* 24(2-3):384–418, 2014.

55David Walker. Sistemas do tipo subestrutural. *Tópicos Avançados em Tipos e Programação Idiomas,* páginas 3–44, 2005.

56 Aaron Weiss, Olek Gierczak, Daniel Patterson e Amal Ahmed. Óxido: A essência da ferrugem, 2021. arXiv:1903.00982.

57 Nobuko Yoshida e Vasco T Vasconcelos. Primitivas de linguagem e disciplina de tipo para programação baseada em comunicação estruturada revisitada: Dois sistemas para comunicação de sessão de ordem superior . *Electronic Notes in Theoretical Computer Science,* 171(4):73–93, 2007.

A Tipos Lineares Regulares

A sintaxe de tipo para tipos regulares lineares (conforme discutido na Seção 1) é a seguinte.

$$\ddot{y} ::= \ddot{y} \ddot{y} \ddot{y} \quad {}^{\ddot{y}} \mid \ddot{y} \ddot{y} \ddot{y} \quad {}^{\ddot{y}} \mid 1 \mid 0 \mid X \mid \mu X.\ddot{y}$$

onde X varia sobre variáveis de recursão. Nós nos concentramos principalmente no subconjunto não recursivo (apenas as quatro primeiras construções acima), embora os tipos recursivos apareçam na Seção 5 e os incluamos aqui para coerência com a descrição usual de tipos regulares na literatura. Por toda parte, usamos $\ddot{y}$ e $\ddot{y}$ para variar os tipos e também A, B, C, D.

As regras de digitação para tipos regulares lineares são as seguintes, que incluem seus formadores de termo padrão.

$$\frac{}{x : A \ddot{y} x : A}\ \text{var} \qquad \frac{\ddot{y}\ \ddot{y}\ t : A \ddot{y} \ddot{y}\ t \qquad \ddot{y} : B}{\ddot{y}\ \ddot{y}\ (t, t\ddot{y}) : A \ddot{y} B}\ \ddot{y}\text{I}\ \ddot{y}, \qquad \frac{\ddot{y}\ \ddot{y}\ t : A \ddot{y} B\ \ddot{y}, u : A, v : B \ddot{y}\ t\ \ddot{y}, \ddot{y}\ \ddot{y}\ \textbf{let}\ (u, \text{v}) \ddot{y} : \qquad \ddot{y} : C}{= t\ \textbf{em}\ t}\ C\ddot{y}E\ \_$$

$$\frac{}{\ddot{y}\ \ddot{y} : 1}\ 1\text{I} \qquad \frac{\ddot{y}\ \ddot{y}\ t : 1\ \ddot{y}\ \ddot{y}\ t \qquad \ddot{y} : C}{\ddot{y}, \ddot{y}\ \ddot{y}\ \textbf{let}\ () = t\ \textbf{em}\ t \qquad \ddot{y} : C}\ 1\text{E} \qquad \frac{\ddot{y}\ \ddot{y}\ t : A}{\ddot{y}\ \ddot{y}\ \text{inl}\ t : A \ddot{y} B}\ \ddot{y}\text{IL} \qquad \frac{\ddot{y}\ \ddot{y}\ t : B}{\ddot{y}\ \ddot{y}\ \text{inr}\ t : A \ddot{y} B}\ \ddot{y}\text{IR}$$

$$\frac{\ddot{y}\ \ddot{y}\ t : A \ddot{y} B \ddot{y}, u : A \ddot{y}\ t \qquad \ddot{y} : C \ddot{y}, v : B \ddot{y}\ t \mid \text{inr}\ v \ddot{y} \text{“} :}{\ddot{y}, \ddot{y}\ \ddot{y}\ \textbf{caso}\ t\ \text{de inl}\ u \ddot{y}\ t \qquad \ddot{y}\ t \qquad \text{“} : C}\ C\ddot{y}E\ \_$$

Observe que acima não incluímos o espaço de função linear $\ddot{y}\ \ddot{y}\ \ddot{y}$ apenas a sintaxe de tipos $\ddot{y}$ desde que consideramos regulares na Seção 1, mas as funções lineares são usadas ao longo do artigo. Suas regras de introdução e eliminação são:

$$\frac{\ddot{y}, x : A \ddot{y}\ t : B \ddot{y}\ \ddot{y}}{\ddot{y}x.t : A \ddot{y} B} \qquad \frac{\ddot{y}\ \ddot{y}\ t : A \ddot{y} B \ddot{y}\ \ddot{y}\ t \qquad \ddot{y} : Um}{\ddot{y}, \ddot{y}\ \ddot{y}\ t\ t\ddot{y} : B}$$

Conforme declarado na Seção 1, os tipos regulares (lineares) se comportam *como* um semi-anel comutativo, ou seja, ÿ e ÿ são ambos comutativos e associativos com 1 e 0 como suas unidades correspondentes, com distributividade, mas todos até o isomorfismo.

$$A \ddot{y} (B \ddot{y} C) = \ddot{y} (A \ddot{y} B) \ddot{y} C \qquad A \ddot{y} (B \ddot{y} C) \ddot{y} = (A \ddot{y} B) \ddot{y} C \qquad A \ddot{y} (B \ddot{y} C) \ddot{y} = (A \ddot{y} B) \ddot{y} (A \ddot{y} C)$$
$$A \ddot{y} B \ddot{y} = B \ddot{y} A\ 1 \qquad A \ddot{y} B \ddot{y} = B \ddot{y} A \qquad (B \ddot{y} C) \ddot{y} A \ddot{y} = (B \ddot{y} A) \ddot{y} (C \ddot{y} A)$$
$$\ddot{y} A = \ddot{y} A \qquad A \ddot{y} 0 \ddot{y} = A \qquad A \ddot{y} 0 \ddot{y} = 0$$

Todos os isomorfismos acima são testemunhados por pares de funções mutuamente inversas.

Tipos regulares também permitem uma noção de expoente (positivo), com $\ddot{y}\ a$ definidoindutivamente como:

$$\ddot{y}\ 0 = 1 \qquad a+1\ a = \ddot{y}\ \ddot{y}\ \ddot{y}\ \ddot{y}$$

As leis usuais de expoentes positivos sustentam o isomorfismo via associatividade e comutatividade (e remoção de unidades no caso do isomorfismo mais à esquerda), para todo $a \ddot{y} 0, b \ddot{y} 0$:

$$\ddot{y}\ 1 \ddot{y} = \ddot{y} \qquad \text{um}\ \ddot{y} \ddot{y} = \ddot{y}\ b\ a+b \ddot{y}\ \ddot{y} \qquad (\ddot{y}\ a)\ b\ ab \ddot{y} = \ddot{y} \qquad (\ddot{y}\ \ddot{y}\ \ddot{y})\ a \qquad \ddot{y} = \ddot{y}\ a\ \ddot{y}\ \ddot{y}\ a$$

A.1 Equações

Este cálculo tem equações para (bi)funcionalidade de ÿ e ÿ:

$$id \ddot{y} id = id \quad (f \ddot{y} g) \ddot{y} (h \ddot{y} k) = (f \ddot{y} h) \ddot{y} (g \ddot{y} k) \qquad id \ddot{y} id = id \quad (f \ddot{y} g) \ddot{y} (h \ddot{y} k) = (f \ddot{y} h) \ddot{y} (g \ddot{y} k)$$

As seguintes equações são sobre a interação de cotuplicação, injeções e o ÿ bifuntor, que são subconjuntos daqueles de Gibbons [18] que são deriváveis para a parte do coproduto de tipos regulares lineares:

$$[f, g] \ddot{y} \text{inl} = f \quad [f, g] \ddot{y} \text{inr} = g \qquad [h \ddot{y} \text{inl}, h \ddot{y} \text{inr}] = h \quad [\text{inl}, \text{inr}] = id \qquad [f, g] \ddot{y} (h \ddot{y} k) = [f \ddot{y} h, g \ddot{y} k]\ h \ddot{y} [f, g] = [h \ddot{y} f, h \ddot{y} g]$$

Os axiomas usuais para um funtor monoidal (relaxado) valem para o funtor inverso (contravariante). Esses axiomas são os seguintes:

$$\text{mmult} \ddot{y} (\text{munit} \ddot{y} id) \ddot{y} \ddot{y}i = \ddot{y}\ \ddot{y}1 \qquad :A\ \ddot{y}1\ \ddot{y}\ (1 \ddot{y} A) \ddot{y}\ \ddot{y}1$$
$$\text{mmult} \ddot{y} (id \ddot{y} \text{munit}) \ddot{y} \ddot{y}i = \ddot{y}\ \text{mmult}\ \ddot{y}1 \qquad :A\ \ddot{y}1\ \ddot{y}\ (A \ddot{y} 1) \ddot{y} 1 \ddot{y}$$
$$\ddot{y} (\text{mmult} \ddot{y} id) \ddot{y} \ddot{y}i = \ddot{y}\ \ddot{y}1\ \text{mmult}\ \ddot{y}(id \ddot{y} \text{mmult}) :A \ddot{y} 1 \ddot{y} (B \ddot{y} 1 \ddot{y} C \ddot{y} 1\ )\ddot{y}((A \ddot{y} B) \ddot{y} C)\ \ddot{y}1$$

onde $\ddot{y}$ é a associatividade e $\ddot{y} : 1 \ddot{y} A \ddot{y} A$ e $\ddot{y} : A \ddot{y} 1 \ddot{y} A$ (e seus inversos $\ddot{y}i$ e $\ddot{y}i$) testemunham as propriedades unitárias de ÿ.

5:26 Como obter o inverso de um tipo

B A involução é um isomorfismo

Mostramos que para todo ÿ ÿ1 ÿ= , então *ÿ* ÿ1 é uma involução sequencialmente realizável até o isomorfismo, ou seja, *(ÿ* ÿ1 com *ÿ* ÿ, ((ÿ ÿ 1) ÿ 1) implementada como *ÿx.ÿf.fx* (consulte a Seção 6) e a direção inversa como segue, usando a sintaxe do cálculo GV (conforme formulado por [30]) em vez de Grânulo conforme mostrado na Seção 6, do tipo ((ÿ ÿ 1) ÿ 1) ÿ *ÿ :*

(ÿk.let (x, c) = *recv(fork(ÿc.k(ÿx.send c* x))) **in let** () = wait *c* **in** x)

Para provar que *ÿ* é uma involução até o isomorfismo, precisamos mostrar que as funções *i : ÿ* ÿ ((ÿ ÿ 1) ÿ 1 e *j :* ((ÿ ÿ 1) ÿ 1) ÿ *ÿ* são mutuamente inversas, ou em outras palavras que j(i(t)) = *t : ÿ* e i(j(h)) = *h : (ÿ* ÿ 1) ÿ 1.

Aproveitamos a teoria da igualdade ÿÿ de GV com base em sua semântica operacional dada por [30], que é a mesma semântica operacional para canais implementada no Granule [42].

Mostramos ambas as direções separadamente, como segue:

j(i(t)) = *(ÿk.let (x,* c) = *recv(fork(ÿc.k(ÿx.send c* x))) **in let** () = wait *c* **in** *x)(ÿf.f* t)

= **(let** *(x,* c) = recv(fork(ÿc.(ÿf.f *t)(ÿx.send c* x))) **in let** () = wait *c* **in** x) = **(let** *(x,* c) =

recv (fork(ÿc.(ÿx.send c x)t)) **in let** () = wait *c* **in** x) = **(let** *(x,* c) =

recv(fork(ÿc.send *c* t)) **in let** () = espera *c* **em** x)

Verificando a digitação da expressão interna, temos:

ÿc.send *ct : Chan(!ÿ.end!)* ÿ Chan(end!)

fork(ÿc.send c t) : *Chan(?ÿ.end?)*

recv(fork(ÿc.send *c* t)) : *ÿ* ÿ Chan (fim?)

Então temos a ligação *(x,* c): *ÿ* ÿ Chan(end?). Aplicando a semântica de configuração global do GV [30, Figura 4], obtemos o seguinte:

let *(x,* c) = recv(fork(ÿc.send *c* t)) **in let** () = wait *c* **in** x)

(Levantar+Fork) ÿ (ÿc)(let *(x,* c) = recv c **in let** () = esperar *c* **in** x) | (enviar c t)

(Lift+Send) ÿ (ÿc)(let *(x,* c) = *(t,* c) **in let** () = espera *c* **in** x) | c (LiftV) ÿ (ÿc)

(let () = espera c **em** t) | c (Lift+Wait) ÿ **let** ()

= () **em** *t* (LiftV) ÿ *t*

Assim, j(i(t)) = *t : ÿ* conforme necessário.

Na direção oposta do isomorfismo temos então, *h : (ÿ* ÿ1)ÿ1 com

i(j(h)) = *(ÿx.ÿf.f* **x)(let** *(x,* c) = *recv(fork(ÿc.h(ÿx.send c* x))) **in let** () = wait *c* **in** x)

= *(ÿf.f* **(let** *(x,* c) = *recv(fork(ÿc.h(ÿx.send c* x))) **in let** () = wait *c* **in** x)) = **let** *(x,* c) =

recv(fork(ÿc.h(ÿx.send c x))) **in let** () = wait *c* **in** *(ÿf.f* x)

Similarmente ao primeiro caso, temos então a digitação interna:

ÿx.send *cx : ÿ* ÿ Chan(end!)

h(ÿx.send c x) : Chan(end!)

fork(h(ÿx.send c x)) : *Chan(?ÿ.end?)*

recv(fork(h(ÿx.send *c* x))) : *ÿ* ÿ Chan (fim?)

Então, novamente, *(x,* c) : *ÿ* ÿ Chan(end?).

Aplicando a semântica de configuração global do GV [30, Figura 4], obtemos:

let *(x,* c) = *recv(fork(ÿc.h(ÿx.send c* x))) **in let** () = wait *c* **in** *(ÿf.f* x)

(Levantar+Fork) ÿ (ÿc) **(let** *(x,* c) = recv c **in let** () = esperar *c* **in** *(ÿf.f* x) | *h* (ÿx.send c x))

Lembre-se que *h : (ÿ* ÿ 1) ÿ 1 portanto sabemos que *h* deve necessariamente usar a entrada portanto após alguma algum *t :* ÿ : () e algum configuração *C* no **,** redução em *h* (ÿx.send c x) obtemos o parâmetro, aplicando-o a ÿ = enviar c *t* **em** *h* ÿ caso de avaliar para isso some **deixar** *c* ponto teve alguns outros efeitos de comunicação. Note que *c :* Chan(end!)ÿ não está nas variáveis livres de *h* ÿ digitação da sessão nos diz que ela não é usada, ou seja, *c* ÿ já que a

Então obtemos a sequência de redução contínua:

(LiftVÿ) ÿÿ (ÿc) **(let** *(x,* c) = recv c **in let()** = wait *c in(ÿf.f* x) | **let** *c* = c **in** *h* ÿ = enviar c *t* **em** *h*ÿ | C)
(Lift+Send) ÿ (ÿc) **(let** *(x,* c) = *(t,* c) **in let()** = wait *c in(ÿf.f* x) | **let** *c* ÿ ÿ | C)
(LiftV) ÿ (ÿc) **(let** () = espera c **in** *(ÿf.f* t) | **let** *c* | *C* ÿ = c **em** *h* ÿ | C)
(Lift+Wait) ÿ *(ÿf.f* t) | *h* ÿ

O resultado é um termo que se comporta como o h original; aplicar o termo *t* de dentro *de h* para a continuação de *f* resulta em uma configuração *C* e tem alguma redução restante para fazer como *h* ÿ. Assim i(j(h)) = *h : (ÿ* ÿ 1) ÿ 1 conforme necessário.