Modelagem comonádica de stencils com execução paralela

Lucas Ieks Minicz[1] **, Daniel Cordeiro**[1] **, Emilio Francesquini**[2]

[1]Escola de Artes, Ciências e Humanidades, Universidade de São Paulo
EACH–USP, São Paulo, Brasil

{lieks,daniel.cordeiro}@usp.br

[2]Centro de Matemática, Computação e Cognição, Universidade Federal do ABC
CMCC–UFABC, Santo André, Brasil

e.francesquini@ufabc.edu.br

Resumo. *Stencils resolvem um tipo de problema muito comum em computação paralela. Comônadas, por sua vez, são uma forma conveniente de escrever código paralelizável, mas implementações de stencils usando comônadas são incomuns fora de exemplos. Neste trabalho descrevemos um protótipo que implementa stencils usando comônadas para investigar os limites dessa estrutura quando se tenta usar bibliotecas existentes para paralelismo, e investigamos formas de solucioná-los. Também observamos que o compilador consegue otimizar nosso código bem, mas precisa de algumas anotações adicionais para isso. Concluímos que o uso de comônadas é muito apropriado para implementar stencils.*

Abstract. *Stencils solve a very common kind of inherently parallel problem, and comonads are a convenient way of writing parallelizable code. However, comonad-based stencil implementations are rare outside of examples. We have developed a prototype that implements stencils using comonads in order to investigate the limits of this structure when using existent parallelism libraries, and investigate ways to solve them. We also observed that the compiler can optimize our code fairly well, even if some additional annotations are necessary. We conclude that comonads are very appropriate for implementing stencils.*

1. Introdução

Um *stencil* é um mapa que leva elementos em uma matriz n-dimensional a um novo valor baseando-se apenas no elemento e sua vizinhança, como por exemplo um filtro Gaussiano de imagem. Stencils são utilizados em diversas aplicações práticas como dinâmica dos fluidos computacional, processamento de sinais (convoluções) e resolução de equações diferenciais ordinárias (EDOs).

Em linguagens funcionais, comônadas definem estruturas que permitem efetuar computação utilizando contextos. Por exemplo, para extrair a média móvel de uma *stream* de dados podemos utilizar comônadas e criar uma função que devolva a média para apenas um dos pontos da *stream*. A aplicação em toda a *stream* é feita de forma automática e, como cada aplicação independe das outras, é inerentemente paralelizável [Uustalu and Vene 2008].

Combinar as duas ideias é, portanto, um passo lógico; porém, como mostra a Seção 2, apesar de haver diversos métodos para aplicar *stencils*, comônadas não são geralmente utilizadas em sua forma pura (apesar de algumas construções similares serem utilizadas).

Se a estrutura de comônada não for estritamente seguida, é razoavelmente direto escrever um código que modela *stencils* usando princípios similares aos de uma comônada. Contudo, tais adaptações não necessariamente mantêm as propriedades úteis da comônada. Por outro lado, a escolha em usar uma comônada seguindo a sua definição matemática exata torna mais difícil descrever paralelismo usando uma estrutura de código específica como, por exemplo, uma *Domain Specific Language* embutida que gera código em tempo de execução. Além disso, é necessário tomar precauções para que o compilador seja capaz de otimizar o código final para executar de forma suficientemente eficiente.

Este trabalho investiga se é possível paralelizar comônadas diretamente, mantendo desempenho aceitável, ou implementar algo similar a comônadas, que mantenha suas propriedades úteis para a implementação de *stencils*, mas que também possa ser facilmente paralelizado pelo compilador. Para isso, desenvolvemos um protótipo na linguagem de programação Haskell para verificar experimentalmente o desempenho dessa estrutura de código e explorar estruturas similares que talvez se mostrem úteis.

Na Seção 2, apresentamos o conceito de comônada e as formas mais comuns de implementar *stencils*. Na Seção 3 explicamos a estrutura e considerações de projeto da nossa implementação. Na Seção 4 fazemos um *benchmark* comparando o desempenho de uma implementação convencional de *stencils* com uma comonádica, e na Seção 5 apresentamos a conclusão.

2. Comônadas e stencils

Para este trabalho, escolhemos a linguagem Haskell, que é uma linguagem de programação puramente funcional, amplamente utilizada na academia [Jones 2003]. Haskell também possui outras implementações de *stencils* e comônadas, e o compilador GHC, que tem um otimizador considerado o estado da arte.

Uma comônada é uma estrutura algébrica que pode ser vista como uma estrutura de dados que representa uma computação que requer um "contexto" para a sua execução [Uustalu and Vene 2008]. Sua definição mais usual é a de uma estrutura de dados que encapa um valor mais seu contexto, e possui duas operações, `extract`, que extrai o valor da comônada, e `extend`, que aplica uma função à comônada e devolve uma nova comônada com o valor de retorno (ver Listagem 1).

A utilidade de uma comônada é que a função `extend` não necessariamente precisa aplicar a função somente uma vez. `extend` pode aplicar a função para todos os elementos do contexto e devolver uma comônada com o contexto inteiro modificado (Listagem 2). O fato de uma função ser passada também permite que se controle através da comônada quando e como a função vai ser executada. É possível, portanto, implementar uma comônada que só executa quando `extract` é chamado, ou ainda, que calcula apenas o que for necessário para prover o seu resultado (algo que é facilitado pela linguagem Haskell possuir avaliação preguiçosa). Finalmente, também é possível que a comônada execute as várias aplicações da função em paralelo de maneira transparente ao seu utilizador.

Listagem 1. Uma definição da interface de uma comônada, em Haskell

```haskell
-- O tipo `w a' é uma comônada se tiver definições para as funções
↪  abaixo
class Comonad w where
  -- w a -> a é uma função recebendo `w a' e devolvendo `a'
  extract :: w a -> a
  -- uma função recebendo uma função `w a -> a' e devolvendo outra
  ↪  função `w a -> w a'
  extend :: (w a -> a) -> (w a -> w a)
```

Listagem 2. A definição de uma comônada para uma lista

```haskell
-- Listas são comônadas onde
instance Comonad [] where
  -- extract devolve o primeiro elemento da lista
  extract l = head l
  -- extend aplica a função a todas as sublistas da lista
  extend f [] = []
  extend f l = f l : extend f (tail l)
```

Há várias formas de implementar *stencils*; as duas mais comuns são fazer cada operação no arranjo inteiro (ver Listagem 3), ou fazer todas as operações elemento-a-elemento (ver Listagem 4) [Roth et al. 1997]. A eficiência de cada uma delas depende da situação e da arquitetura de execução. Por exemplo, em arquiteturas de paralelismo maciço, como GPGPUs e supercomputadores, executar as operações elemento a elemento em geral permite mais otimizações [Schaefer and Fey 2011].

As bibliotecas de *stencils* que usam o modelo elemento-a-elemento costumam abstrair o laço e a soma de coordenadas, além das verificações de condições de borda (Listagem 5). Isso é similar ao modelo de comônadas (Listagem 6), a principal diferença sendo que as comônadas definem uma interface com propriedades específicas, enquanto que as bibliotecas de *stencils* não necessariamente seguem todas a mesma interface.

Haskell possui uma implementação padrão para comônadas [Kmett 2021], que inicialmente usamos para este trabalho. Entretanto, devido ao parâmetro `r` da classe `Matrix` definida a seguir, não pudemos continuar utilizando-a. Nenhuma das bibliotecas de stencils mencionadas a seguir usa comônadas.

Existem várias implementações de *stencils* em Haskell, como nas bibliotecas de [Kuleshevich 2021] e [Chalmers 2020], que usam uma abordagem similar a comônadas, mas definem um novo tipo `Stencil` mais uma função para aplicar o *stencil*, ao invés de

Listagem 3. Filtro passa-baixa no arranjo inteiro

```haskell
-- <+> é adição elemento a elemento e `shift' desloca as coordenadas do
↪  array
res = (shift arr 1 <+> arr <+> shift arr (-1)) / 3
```

Listagem 4. Filtro passa-baixa elemento-a-elemento

```
-- `!' indexa um arranjo, e [a | x <- [0..n]] computa `a' para cada x
↪  de 0 a n
res = Array [(arr ! (x-1) + arr ! x + arr ! (x+1)) / 3
            | x <- [0..length arr]]
```

Listagem 5. Filtro passa-baixa com bibilioteca de *stencils*

```
-- \get -> ... é uma lambda
-- note que `get' nesse caso é uma função
res = runStencil arr $ \get ->
  (get (-1) + get 0 + get 1) / 3
```

usar funções diretamente (como comônadas fazem). No trabalho de [Lippmeier and Keller 2011], a interface é ainda mais limitada, sendo específica para convoluções. Em [Lesniak 2010], vetores monádicos são usados, o que ajuda no desempenho mas impede os *stencils* de serem usados em contextos que não sejam monádicos. Em [Orchard and Mycroft 2011], macros são usadas para estender a linguagem com uma sublinguagem específica para *stencils*, o que permite assegurar certas propriedades da execução estaticamente, mas obriga o programador a aprender uma linguagem diferente para esse fim.

3. O protótipo

Nosso protótipo de *stencils* implementa uma pequena biblioteca de processamento de matrizes n-dimensionais, de forma convencional, e a comônada é implementada usando essa biblioteca como base.

A biblioteca possui três implementações diferentes: uma usando `Data.Vector`, que é a implementação de vetores padrão em Haskell e que executa os *stencils* sequencialmente na CPU, e duas usando a biblioteca *Repa* [Lippmeier et al. 2012], que implementa vetores com execução em paralelo, uma executando os *stencils* serialmente (para comparar com a versão paralela sem viés por conta das estruturas de dados serem diferentes) e a outra em paralelo.

A escolha da implementação é completamente transparente aos *stencils* em si. Ela pode ser alterada trocando qual tipo de matriz é passada ao *stencil*. Por exemplo, uma implementação funcional do *Conway's Game of Life* usando somente as funções básicas de matrizes n-dimensionais poderia ser feita como mostrado na Listagem 7, e então executada de diversas formas, como mostrado na Listagem 8.

Listagem 6. Filtro passa-baixa com comônadas

```
-- `stencil' é a função devolvida por `extend'
stencil = extend $ \arr ->
  (arr ! -1 + arr ! 0 + arr ! 1) / 3
res = stencil arr
```

Listagem 7. Game of Life

```
-- `gameOfLife' funciona para qualquer `m' que seja uma matriz
↪  bidimensional
gameOfLife :: Matrix r DIM2 m => m r DIM2 Bool -> m (MResult m) DIM2
↪  Bool
gameOfLife img = mnew (msize img) $ \(Z :. x :. y) ->
  -- n é o número de células vivas ao redor desta
  let n = sum [ if mget img (ix2 (x + dx) (y + dy)) then 1 else 0
              | dx <- [-1..1], dy <- [-1..1], (dx, dy) /= (0, 0) ] in
    -- a célula está viva se tem 3 vizinhos ou se já estava viva e tem
    ↪  2 vizinhos
    n == 3 || (mget img (ix2 x y) && n == 2)
```

Listagem 8. Exemplos de uso

```
img1 :: MatrixVector () DIM2 Bool
gameOfLife img1 -- usa Data.Vector para armazenar a imagem
img2 :: MatrixArray U DIM2 Bool
gameOfLife img2 -- usa Repa em modo serial para executar o stencil
img3 :: MatrixParallel U DIM2 Bool
gameOfLife img3 -- usa Repa em modo paralelo para executar o stencil
```

`gameOfLife` é paramétrico à classe `Matrix`, então qualquer tipo que implemente essa interface pode ser usado, desde que seja bidimensional (que é o que `DIM2` significa). Assim o tipo de execução é determinado somente pelo tipo da matriz passada ao *stencil*.

A implementação do *stencil* em si usa a função `mnew`, que chama uma função para cada elemento da matriz, passando as coordenadas do elemento em questão, e usa o resultado dessa função para construir a matriz resultante. A função interna usa `mget` para acessar o estado da célula sendo alterada e das células ao redor para calcular o novo valor da célula, segundo as regras do autômato celular.

Uma implementação usando comônadas poderia ser feita como mostrado na Listagem 9, que fora os tipos, é bem similar, mas é mais sucinta, principalmente por não precisar manter a posição atual, o que é feito de maneira transparente pela comônada. Ela pode ser utilizada da mesma forma.

O tipo `FocusedMatrix` implementa uma comônada como uma matriz com um foco. Como a posição na matriz é implícita na comônada, não é necessário receber as

Listagem 9. Game of Life com comônadas

```
gameOfLife :: Matrix r DIM2 m =>
  FocusedMatrix m r DIM2 Bool -> FocusedMatrix m (MResult m) DIM2 Bool
gameOfLife = extend $ \img ->
  let n = sum [ if get img (ix2 x y) then 1 else 0
              | x <- [-1..1], y <- [-1..1], (x, y) /= (0, 0) ] in
    n == 3 || (extract img && n == 2)
```

Listagem 10. Definição da classe Matrix (detalhes omitidos por clareza)

```
class (Functor (m r sh), Shape sh) => Matrix m r sh a where
  type MResult m :: *
  mget :: m r sh a -> sh -> a
  mnew :: sh -> (sh -> a) -> m (MResult m) sh a
  minside :: m r sh a -> sh -> Bool
  msize :: m r sh a -> sh
  mzipWith :: (a -> b -> c) -> m r sh a -> m s sh b -> m (MResult m) sh
   ↪  c
```

Listagem 11. A comônada

```
data (Matrix m r sh a, Shape sh) => FocusedMatrix m r sh a =
  FocusedMatrix { unfocus :: m r sh a, focusCoordinates :: sh }
...
get :: (Matrix m r sh a, Shape sh) => FocusedMatrix m r sh a -> sh -> a
get (FocusedMatrix mat fp) p = mget mat (addDim fp p)
...
```

coordenadas explicitamente ou somar a elas para calcular coordenadas relativas. A função `extract` extrai o valor no foco da comônada.

Apesar da mesma interface da classe `Comonad` padrão de Haskell ter sido seguida, a classe em si não foi utilizada, pela necessidade de passar o parâmetro `r`. `r` pode ser diferente na entrada e saída de cada função, e a classe `Comonad` requer que o tipo comônada seja sempre o mesmo. Esse parâmetro é essencial para conseguir bom desempenho com a *Repa*.

3.1. Implementação

O código é estruturado em dois módulos: *MatrixComonad* implementa os tipos de matrizes e a comônada, e *Main* implementa os testes. O tipo central, e o que permite a genericidade, é a classe `Matrix`, que é definida aproximadamente como na Listagem 10.

O parâmetro `sh` especifica o número de dimensões da matriz, e o parâmetro `r` é um tipo que pode ser usado internamente pela implementação para representar informação em tempo de compilação. O tipo `a` é o tipo do elemento. O tipo associado `MResult m` representa o `r` devolvido pelas funções de `Matrix`. As funções `mget` e `mnew` já foram mencionadas, e `minside` e `msize` servem para verificar se um ponto está dentro da matriz ou não, e para recuperar o tamanho da matriz, respectivamente. `mzipWith` aplica uma função binária a duas matrizes, elemento a elemento.

A comônada é implementada pelo tipo `FocusedMatrix`, que é definido na Listagem 11. Ele é definido em termos de `Matrix`, como um *wrapper* que soma o ponto focal ao ponto que ele recebe antes de chamar a função equivalente da classe `Matrix`.

A classe `Shape` se refere ao tipo dos índices sobre a matriz. Um escalar é representado por `Z`. Mais dimensões podem ser adicionadas escrevendo `Z:.Int`, `Z:.Int:.Int` e assim por diante. Também existem tipos pré-definidos como `DIM0`, `DIM1` e `DIM2` por conveniência. Ela pertence à biblioteca *Repa* e está sendo usada aqui sem modificação.

3.2. Especialização

Uma otimização de compilador essencial é a especialização para cada tipo de matriz. A implementação padrão de uma função genérica em Haskell envolve passar dicionários contendo as implementações das funções da classe em tempo de execução, mas é muito mais eficiente (e permite outros tipos de otimização) se o compilador já souber qual é o tipo que está sendo passado de antemão.

Em muitos casos (especialmente em funções no mesmo módulo), o compilador faz essa otimização automaticamente, mas se não fizer, o impacto no desempenhho pode ser grande. Por isso, inserimos *pragmas* (`SPECIALIZE`) nas funções mais chamadas (`pixel` e `inside`) para garantir que o compilador não vai chamar a versão genérica (e portanto mais lenta) da função.

4. Resultados experimentais e análise

Executamos como *benchmark* o algoritmo *autolight*[1], que é moderadamente pesado computacionalmente e usa várias operações comuns em processamento de imagem, e 128 iterações do *Game of Life* como exemplo de *stencil* iterativo, como pode ser visto na Tabela 1. A imagem de teste do *autolight* tem tamanho de 900x1277, e a imagem de teste do *Game of Life* tem tamanho de 512x512. Os benchmarks foram executados pela biblioteca *Criterion* [O'Sullivan 2014] com dois *threads*, em um Intel® Core™ i7-7500U @ 2.70GHz, 2 *cores*/4 *threads*, Linux kernel 5.10.60, GHC 8.10.6.

Tabela 1. Os resultados dos *benchmarks*. (NC) representa uma função não comonádica (base de comparação) e (C) representa a versão comonádica. A razão entre o tempo da implementação comonádica e a base de comparação é mostrada entre parênteses.

benchmark	Data.Vector	Repa (serial)	Repa (paralelo)
autolight (NC)	75,86s (1)	58,90s (1)	37,19s (1)
autolight (C)	71,74s (0,94)	54,38s (0,92)	35,05s (0,94)
game of life (NC)	8,70s (1)	3,59s (1)	2,07s (1)
game of life (C)	8,33s (0,96)	3,73s (1,04)	2,24s (1,08)

A *Repa* serial internamente usa vetores, mas ainda assim é mais rápida que *Data.Vector*. Isso provavelmente se deve ao fato de a *Repa* usar fusão de operações, enquanto que o *Data.Vector* faz cada operação individualmente, sem mais otimizações. A *Repa* paralela é mais rápida que a serial, como esperado. Note que a implementação do algoritmo em si não muda; só a implementação da comônada. Os ganhos de desempenho são gratuitos do ponto de vista do usuário da biblioteca.

Um fenômeno curioso é o de que na maioria dos testes (mas não todos), a versão comonádica é ligeiramente mais rápida do que a versão não comonádica; inicialmente assumimos que fosse uma flutuação na medição, mas com testes maiores vimos que não é isso. Não sabemos a causa exatamente, mas uma hipótese é a de que o compilador otimiza esse código de comônadas um pouco melhor. O fato de *Game of Life* com a *Repa* ser consistentemente mais rápido na versão não comonádica (o contrário dos outros casos) corrobora essa hipótese.

[1] O código utilizado nos experimentos está disponível publicamente em `https://github.com/min4builder/autolight/tree/wic2021`.

5. Conclusão

Comônadas tornam o código de *stencils* mais simples de ler e entender, além de serem, de modo geral, uma boa estrutura para implementar *stencils*. Implementá-las de forma eficiente não é muito diferente de implementar outras técnicas funcionais, mas dependendo do tipo de otimização que for desejada, pode ser necessário alterar os tipos em relação à definição padrão de comônada.

A avaliação experimental do protótipo proposto mostra que o uso de comônodas na implementação de *stencils* não só permite uma representação conveniente do problema, como também resulta em desempenho tão bom quanto outras interfaces.

Como trabalho futuro, pretende-se investigar melhor as características de desempenho de implementações comonádicas e experimentar outras formas de paralelismo.

Referências

[Chalmers 2020] Chalmers, C. (2020). dense: Mutable and immutable dense multidimensional arrays. Disponível em: `https://hackage.haskell.org/package/dense`. Acesso em: 06 de ago. de 2021.

[Jones 2003] Jones, S. P. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003 edition.

[Kmett 2021] Kmett, E. A. (2021). comonad: Comonads. Disponível em: `https://hackage.haskell.org/package/comonad`. Acesso em: 06 de ago. de 2021.

[Kuleshevich 2021] Kuleshevich, A. (2021). massiv: Massiv is an Array Library. Disponível em: `https://hackage.haskell.org/package/massiv`. Acesso em: 06 de ago. de 2021.

[Lesniak 2010] Lesniak, M. (2010). Pastha: Parallelizing stencil calculations in haskell. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, page 5–14, New York, NY, USA. ACM.

[Lippmeier et al. 2012] Lippmeier, B., Chakravarty, M., Keller, G., and Peyton Jones, S. (2012). Guiding parallel array fusion with index types. In *Haskell Symposium, Copenhagen*. ACM.

[Lippmeier and Keller 2011] Lippmeier, B. and Keller, G. (2011). Efficient parallel stencil convolution in haskell. In *Sigplan Notices - SIGPLAN*, volume 46, pages 59–70.

[Orchard and Mycroft 2011] Orchard, D. and Mycroft, A. (2011). Efficient and correct stencil computation via pattern matching and static typing. *Electronic Proceedings in Theoretical Computer Science*, 66:25.

[O'Sullivan 2014] O'Sullivan, B. (2014). criterion: a Haskell microbenchmarking library. Disponível em: `http://www.serpentine.com/criterion/`. Acesso em: 06 de ago. de 2021.

[Roth et al. 1997] Roth, G., Mellor-crummey, J., Kennedy, K., and Brickner, R. G. (1997). Compiling stencils in high performance fortran. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–20. ACM Press.

[Schaefer and Fey 2011] Schaefer, A. and Fey, D. (2011). High performance stencil code algorithms for GPGPUs. *Procedia Computer Science*, 4:2027–2036.

[Uustalu and Vene 2008] Uustalu, T. and Vene, V. (2008). Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263–284.