Calculadora de Streams

Daniel Kiyoshi Hashimoto Vouzella de Andrade – 119025937

Julho 2023

1 Por quê Streams?

Streams são sequências infinitas de **coisas**

$$(a,\ b,\ c,\ d,\ e,\ f,\ \dots)$$

Aparecem em Computação:

- Stream de dados
  - Vídeo
  - Áudio
  - ...
- Processamento de Sinais
  - Filtros, conversores, ...

Aparecem em Matemática:

- Séries de Potência/Polinômios

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots$$

- Série de Taylor

$$f(x) = f(a) + \frac{f'(a) \cdot (x-a)^1}{1!} + \frac{f''(a) \cdot (x-a)^2}{2!} + \cdots$$

- Problemas de contagem (sequências aparecem como solução)
  - Fibonacci

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ \dots$$

  - Números Triangulares

$$1,\ 3,\ 6,\ 10,\ \dots,\ \frac{n \cdot (n+1)}{2},\ \dots$$

2 O que são Streams?

Matematicamente, uma Stream de A é um elemento do conjunto:

$$A^\omega := \{S\ ;\ S \in \{0,\ 1,\ 2,\ \dots\} \to A\}$$

ou seja, funções de $\mathbb{N}$ para seus elementos de A.
Mas podemos fingir que são uma 'lista infinita' de As:

$$S := (s_0,\ s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ \dots)$$

2.1 O que podemos fazer com Streams?

Temos algumas operações úteis:

- Indexar: pedir o i-ésimo elemento da stream

  $$S[i] := s_i$$

  Quando $i = 0$ chamamos a operação de head.

- Tail/Derivar/Shift Left: jogar o primeiro elemento **coisa** fora

  $$S' := (s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ \dots)$$

  $$S'[i] := S[i+1]$$

- Cons/Integrar/Shift Right: colocar uma **coisa** no início da Stream

  $$(a : S) := (a,\ s_0,\ s_1,\ s_2,\ s_3,\ \dots)$$

  $$\begin{cases} (a:S)[0] := a \\ (a:S)[i] := S[i-1] \end{cases}$$

2.2 Como definimos Streams?

Podemos usar *Equações Diferenciais* para definir Streams.
Para isso, basta dizer quem é *Head* e *Tail* de S.
Alguns exemplos:

- "Sempre" a:

  $$\begin{cases} S[0] &= a \\ S' &= S \end{cases}$$

- "Sempre" a e b:

  $$\begin{cases} S[0] &= a \\ S'[0] &= b \\ S'' &= S \end{cases}$$

- "Sempre" a e b (versão 2):

$$\begin{cases} S[0] & = a \\ T[0] & = b \\ S' & = T \\ T' & = S \end{cases}$$

- Os pares de S:

$$\begin{cases} par(S)[0] & = S[0] \\ par(S)' & = par(S'') \end{cases}$$

- Os ímpares de S:

$$\begin{cases} impar(S)[0] & = S[1] \\ impar(S)' & = impar(S'') \end{cases}$$

- Zip de S e T:

$$\begin{cases} zip(S,T)[0] & = S[0] \\ zip(S,T)' & = zip(T,S') \end{cases}$$

- Double de 'S':

$$\begin{cases} double(S)[0] & = S[0] \\ double(S)[1] & = S[0] \\ double(S)'' & = double(S') \end{cases}$$

Com essas definições, podemos ir calculando *head*s e *tail*s e começar a montar a lista infinita.
E após algumas tentativas as vezes conseguimos adivinhar a "cara" dessas Streams.
Assim são o início dessas streams:

- "Sempre" a:

$$(a,\ a,\ a,\ \dots)$$

- "Sempre" a e b:

$$(a,\ b,\ a,\ b,\ a,\ b,\ \dots)$$

- Pares de S:

$$(s_0,\ s_2,\ s_4,\ s_6,\ \dots)$$

- Impares de S:

$$(s_1,\ s_3,\ s_5,\ s_7,\ \dots)$$

- Zip de S e T:

$$(s_0,\ t_0,\ s_1,\ t_1,\ s_2,\ t_2,\ \dots)$$

- Double de S:

$$(s_0,\ s_0,\ s_1,\ s_1,\ s_2,\ s_2,\ \dots)$$

2.3 Equivalência de Streams com Autômatos

Um autômato é uma quadrupla $(Q, q_0, out, next)$, onde:

- Q: Conjunto de Estados (possívelmente infinito)
- q_0: O primeiro estado
- out: Função $Q \to A$
- $next$: Função $Q \to Q$

A ideia é que cada estado gera um elemento (usando out) e $next$ diz o resto da Stream.
out está relacionada com *head* e $next$ está relacionada com *tail*.
Usando essa ideia, podemos pegar uma Stream $S \in A^\omega$ e fazer um autômato equivalente

$$(S, s_0, head, tail)$$

onde:

$$\begin{aligned} s_0 &:= S[0] \\ head(T) &:= T[0] \\ tail(T) &:= T' \end{aligned}$$

3 Streams de Números: um caso especial

Daqui para frente vamos usar *Números Racionais* $\mathcal{N}$ no lugar de A. Mas vai funcionar com qualquer outro *Corpo*.

3.1 Operações básicas

3.1.1 Insersão

Podemos levar elementos de $\mathcal{N}$ em elementos de $\mathcal{N}^\omega$:

$$\begin{cases} \langle n \rangle[0] & := n \\ \langle n \rangle' & := \langle 0 \rangle \end{cases}$$

Intuitivamente:

$$\langle n \rangle := (n,\ 0,\ 0,\ 0,\ \dots)$$

3.1.2 Soma

Podemos somar dois $\mathcal{N}^\omega$, S e T, resultando em outro $\mathcal{N}^\omega$:

$$\begin{cases} (S+T)[0] & := S[0] + T[0] \\ (S+T)' & := (S' + T') \end{cases}$$

Intuitivamente:

$$S + T := (s_0 + t_0,\ s_1 + t_1,\ s_2 + t_2,\ \dots)$$

Observe que se "somarmos" com um $n \in \mathcal{N}$:

$$\langle n \rangle + S := (n + s_0,\ s_1,\ s_2,\ \dots)$$

3.1.3 Produto

Podemos multiplicar dois $\mathcal{N}^\omega$, S e T, resultando em outro $\mathcal{N}^\omega$:

$$\begin{cases} (S \times T)[0] & := S[0] \cdot T[0] \\ (S \times T)' & := (\langle S[0] \rangle \times T') + (S' \times T) \end{cases}$$

Esse é menos intuitivo, mas fica parecido com a propriedade distributiva da multiplicação de números:

$$S \times T := (s_0 \cdot t_0,\ s_1 \cdot t_0 + s_0 \cdot t_1,\ s_0 \cdot t_2 + s_1 \cdot t_1 + s_2 \cdot t_0,\ \dots)$$

Observe que se "multiplicarmos" com um $n \in \mathcal{N}$:

$$\langle n \rangle \times S := (n \cdot s_0,\ n \cdot s_1,\ n \cdot s_2,\ \dots)$$

3.2 Semelhança com Polinômios (Stream X)

No geral, a Stream

$$S = (s_0,\ s_1,\ s_2,\ \dots)$$

é muito parecida com o polinômio

$$P(x) = s_0\ x^0 + s_1\ x^1 + s_2\ x^2 + \cdots$$

Temos uma Stream especial, vamos chamar de $X \in \mathcal{N}^\omega$, que façam Streams mais parecidas com polinômios? Queremos algo assim:

$$\begin{aligned}
\langle s_0 \rangle &= (s_0,\ 0,\ 0,\ \dots) \\
\langle s_1 \rangle \times X &= (0,\ s_1,\ 0,\ \dots) \\
\langle s_2 \rangle \times X \times X &= (0,\ 0,\ s_2,\ \dots) \\
&\vdots
\end{aligned}$$

Temos!

$$X := (0,\ 1,\ 0,\ 0,\ 0,\ \dots)$$

Observe que multiplicar por X é equivalente a "empurar" a Stream toda para direita e por um 0 como primeiro elemento:

$$S \times X = (0,\ s_0,\ s_1,\ s_2,\ s_3,\ \dots) = (0 : S)$$

3.3 Recorrência Linear

Podemos usar Streams para resolver Recorrências Lineares.
O exemplo clássico é a recorrência de Fibonacci:

$$\begin{aligned}
F[0] &:= 0 \\
F[1] &:= 1 \\
F[n] &:= F[n-1] + F[n-2]
\end{aligned}$$

Podemos fazer uma versão "recursiva":

$$\begin{aligned}
F &:= (0 + \langle 1 \rangle \times X) + (F \times X) + (F \times X \times X) \\
&:= (\langle 1 \rangle \times X) + (F \times X) + (F \times X \times X) \\
&:= X + (F \times X) + (F \times X \times X)
\end{aligned}$$

No desenho ficaria algo como:

$$\begin{array}{cccccccc}
 & (0, & 1, & 0, & 0, & 0, & \dots) \\
+ & (0, & f_0, & f_1, & f_2, & f_3, & \dots) \\
+ & (0, & 0, & f_0, & f_1, & f_2, & \dots) \\
\hline
= & (f_0, & f_1, & f_2, & f_3, & f_4, & \dots)
\end{array}$$

No olho, podemos descobrir que:

$$\begin{aligned} f_0 &:= 0 \\ f_1 &:= 1 + f_0 \\ f_2 &:= f_0 + f_1 \\ &\vdots \end{aligned}$$

Mas também podemos resolver algebricamente:

$$\begin{aligned} F &= X + F \times X + F \times X \times X \\ F &= X + F \times (X + X \times X) \\ F - F \times (X + X \times X) &= X \\ F \times (1 - X - X \times X) &= X \end{aligned}$$

Agora falta "jogar aquilo para o outro lado".

$$F = \frac{X}{1 - X - X \times X}$$

E pronto!
Mas como funciona essa *matemágica*?

3.4 Inversa de uma Stream

Queremos uma operação de "inversa" com a seguinte propriedade:

$$S \times S^{-1} = \langle 1 \rangle = S^{-1} \times S$$

A seguinte definição funciona:

$$\begin{cases} (S^{-1})[0] & := \frac{1}{S[0]} \\ (S^{-1})' & := \langle \frac{1}{S[0]} \rangle \times S' \times S^{-1} \end{cases}$$

Observe que S^{-1} não está definida quando $S[0] = 0$.

4 A calculadora

A calculadora é um programa iterativo. Ela recebe comandos na notação polonesa reversa, em outras palavras, recebe os argumentos e depois a operação.
Por exemplo:

$$13\ 2 + 4\ 6 *\ -$$

é equivalente à

$$(13 + 2) - (4 * 6)$$

Como ela só trabalha com streams, números são automaticamente inseridos em streams.
O código foi desenvolvido em `C`, e única dependência é um compilador de `C`. Foi usado o `gcc` na versão `11.3.0`.
Existem alguns exemplos em `examples.txt`. O arquivo é uma entrada válida para a calculadora.

4.1 Como rodar

Existe um script para compilar e rodar o programa: `compile_run.sh`. O programa é pequeno o suficiente para compilar o programa logo antes de executar não ser um problema.
Para a compilação ele pode receber dois valores que configuram a saída da calculadora:

- `-DPRINT_COUNT=<N>`: padrão 5.

  Diz quantos elementos da Stream serão mostrados quando ela mostra as Streams que estão na pilha.

- `-DPRINT_TAIL_DEPTH=<N>`: padrão 0.

  Diz o quão profundo o programa tenta ao mostrar a calda da stream quando ela mostra as Streams que estão na pilha.

- `-DDEBUG`

  Liga alguns prints extras para ajudar a debugar a calculadora.

Exemplo, usando as flags:

```
$ ./compile_run.sh -DPRINT_COUNT=10 -DPRINT_TAIL_DEPTH=2
```

Geralmente não é necessário se preocupar com isso.
O programa pode ser encerrado com um EOF (em linux `Ctrl-D`). Não tem um comando `exit` ou similar que encerra a calculadora.

4.1.1 Testes

Existe uma pequena bateria de testes. Para rodar basta executar: `run-tests.sh`.
Os arquivos do teste estão na pasta `tests/`.

4.2 Sintaxe e Comandos

A sintaxe é permissiva, isso quer dizer que: na dúvida, os "erros" serão ignorados ou silenciosamente convertidos para coisas válidas.
Isso não é uma coisa ideal, mas contorna alguns problemas e permite "focar no que importa" (fazer a calculadora funcionar).
Além disso, ela é case-insensitive, ou seja, os seguintes nomes são equivalentes:

- `add`
- `Add`
- `aDd`
- `ADD`

4.2.1 Números

Ao ler um número, converte ele para stream e o empilha.
Números começam com um underscore (`_`) ou digito, aceita vários deles e opcionalmente um ponto (`.`) e mais underscore (`_`) ou digito.
Se o número começa com underscore (`_`) é um número negativo.
Exemplos:

- `0`: 0
- `5`: 5
- `_2`: -2
- `10_000_123.456_7`: 10000123.4567
- `_32.5`: -32.5
- `_`: 0

4.2.2 Operações de Streams

As operações retiram streams da pilha e empilha o resultado na pilha.
Se não tem argumentos o suficiente na pilha, não fazem nada.
Geralmente, o comando pode ser executado usando um símbolo ou um nome.

Soma Pode ser invocada com `+` ou `add`.

Subtração Pode ser invocada com `-` ou `sub`.

Multiplicação Pode ser invocada com `*` ou `mul`.

Inversa Pode ser invocada com `%` ou `inv`.

Derivada/Shift Left Pode ser invocada com `'` ou `tail`.

Integral/Shift Right Essa operação é equivalente a multiplicar por X^n, ou seja, X multiplicado por ele mesmo n vezes. Pode ser invocada com `^n`, onde `n` é um inteiro (preferencialmente maior que 0).

Stream X Esse é um atalho para empilhar a stream X. Pode ser invocada com `x`.
Nota: *Prefira usar `^n` no lugar de várias multiplicações por `x`. O* Shift Right *tem algumas otimizações que economizam memória e tempo de execução. Mas para exemplos pequenos, não faz muita diferença.*

4.2.3 Utilitários da calculadora

Escreve em Registrador A calculadora possui 10 registradores, numerados de 0 até 9. Pode escrever em um deles usando `!r`, onde `r` é um inteiro (preferencialmente no intervalo $[0, 9]$).
Escrever em um registrador pode ser usado para jogar o topo da pilha fora.

Le do Registrador Pode empilhar o conteúdo de um registrador usando `$r`, onde `r` é um inteiro (preferencialmente no intervalo $[0, 9]$).
Os registradores começam inicializados com a Stream $\langle 0 \rangle$.

Copia da Pilha Pode copiar um valor da pilha para o topo dela usando `@s`, onde `s` é um inteiro (dentro do tamanho da pilha, aceita números negativos).

4.2.4 Outros

Comentários Um comentário começa com `#` e vai até o fim da linha.

Não mostra pilha Se a linha termina com `;`, não mostra a pilha.
Isso pode ser útil para não ficar reavaliando os valores das Streams empilhadas.

Mostra registradores Se a linha termina com `;`, não mostra a pilha, mas imprime os registradores que não possuem a Stream $\langle 0 \rangle$.

4.2.5 Ignorados

Outros símbolos e palavras não mapeados são ignorados.
Exemplos:

- `[`
- `]`
- `{`
- `}`
- `~`
- `lalala`
- `arroz`
- `addi`
- `c`

4.3 Alguns detalhes de implementação

4.3.1 Limites arbitrários que podem ser alterados

- Limite de registradores: 10
- Limite de tamanho da pilha: 16 (0x10)
- Limite de tamanho de uma linha da entrada: 1024 (0x400)

4.3.2 Arquitetura

A calculadora possui 3 arquivos principais: `streams.h`, `lexer.h` e `main.c`.

`streams.h` Aqui está toda a implementação de Streams de floats. Streams são representadas como ponteiros para blocos imutáveis de memória.
Elas tem uma "tipagem dinâmica", um campo que diz "qual tipo ela é". Isso permite tratar tudo como se "fosse a mesma coisa" e algumas otimizações, mas a cada operação tem que *ver* qual "tipo" cada coisa é e isso é meio chato (nome disso é *dynamic dispatch*).

`lexer.h` Essa parte "tokeniza" a entrada: lê e segmenta a entrada para um formato mais fácil do programa compreender (*tokens*).

`main.c` É a "cola" de tudo: recebe a entrada, passa para o Lexer, interpreta os *tokens*, realiza as operações de Streams.

4.3.3 Possíveis Melhorias

Reuso de memória Esse programa só aloca memória indefinidamente, nunca se preoculpa em "devolvê-la" ou reutilizá-la. Em teoria, apenas uma stream *não-trivial* sendo exibida várias vezes na tela, pode causar um estouro de memória.
Mas na prática isso não é tão problemático assim, uma Stream ocupa apenas **32 bytes** mais uma memória compartilhada para cada Stream que ela depende (uma soma depende de duas outras streams, por exemplo). Dessa forma, $(\langle 0\rangle + (\langle 0\rangle * \langle 0\rangle))$ ocuparia $\mathbf{32 \cdot 3 = 96}$ **bytes** se compartilhasse todos os $\langle 0\rangle$ e não simplificasse a conta para $\langle 0\rangle$.

Double dispatch As vezes é possível simplificar as streams usando propriedades conhecidas, como:

- $\langle a \rangle + \langle b \rangle \Rightarrow \langle a + b \rangle$
- $\langle a \rangle \times \langle b \rangle \Rightarrow \langle a \cdot b \rangle$
- $(A \times B) + (A \times C) \Rightarrow A \times (B + C)$
- $(A \times X) \times X \Rightarrow A \times X^2$

  Nota: *$A \times X^2$ tem uma representação melhor do que $A \times X \times X$, por isso existe operação `Shift Right`.*
- $(A \times X) + (B \times X^2) \Rightarrow (A + (B \times X)) \times X$
- $\cdots$

Mais "Tipos" de Streams Existem outras representações específicas de Streams que podem ser úteis. Algumas ideias:

- `Loop(L, i)`: guarda uma lista de elementos `L` e um índice `i`:

  Head é `L[i]` e *Tail* é `Loop(L, (i+1) % len(L))`.
- `Autom(L, i, r)`: guarda uma lista de elementos `L`, dois índices `i` e `r`:

  Head é `L[i]` e *Tail* é `if i+1 < len(L) then Autom(L, i+1, r) else Autom(L, r, r)`.

  Deve ser valer a equivalencia: `Loop(L, i) = Autom(L, i, 0)`.

  Nota: *Não foi provado, mas acredito que os Autômatos sempre podem ser transformados em uma sequência de elementos com uma indicação para qual elemento ela deve recomeçar logo depois de chegar ao fim (um desenho agora seria muito bom).*
- **Alguma representação mágica de Razão de Polinômios**:

  Como **Streams Racionais** podem ser representadas como Razões de Polinômios, talvez tenha alguma forma interessante de se calcular *Head* e *Tail* a partir disso.

  Talvez isso tenha alguma representação interessante usando duas listas (uma para a parte "de cima" e outra para a parte "de baixo" da razão).

Lexer Algumas entradas não esperadas quebram o Lexer. Geralmente elas são não-letras e não-digitos, acentos/letras com acentos e caracteres de outras línguas.

Referências

[1] J. J. M. M. Rutten. Elements of stream calculus (an extensive exercise in coinduction). *Theoretical Computer Science*, 2001.

[2] J. J. M. M. Rutten. A tutorial on coinductive stream calculus and signal flow graphs. *Theoretical Computer Science*, 2005.

[3] Pawel Sobocinski. Fibonacci and sustainable rabbit farming, 2016. Disponível em `https://graphicallinearalgebra.net/2016/09/07/31-fibonacci-and-sustainable-rabbit-farming`. Acessado em 14/06/2023.

[4] Herbert S. Wilf. *generatingfunctionology*. Academic Press, 1994. Disponível em `https://www2.math.upenn.edu/~wilf/DownldGF.html`. Acessado em 14/06/2023.