S. Lemos Meira◇

Sumário

Alguns aspectos de complexidade de algoritmos puramente funcionais são considerados, dentro de um modelo de execução de linguagens de alta ordem, não estritas. O problema de espaço residente criado pelo critério de execução é estudado para alguns exemplos, e soluções são propostas para os casos analisados, usando uma variação da técnica de programação transformacional.

Abstract

We consider some aspects of complexity of purely applicative algorithms, in higher-order, non-strict languages. We study the increased residency problem (if compared to the corresponding imperative versions of the same algorithms) due to lazy evaluation of the language, which is exemplified in some basic definitions. For the cases in question, solutions are proposed using a variation of the methods of transformational programming.

◇ Engenheiro de Eletrônica (ITA, 1977), M. Sc. em Computação (UFPE, 1981), Ph. D. in Computing (Kent, UK, 1985). Áreas de interesse: linguagens funcionais, complexidade de algoritmos, teoria da computação, arquiteturas não-von-Neumann. Endereço: Departamento de Eletrônica e Sistemas, UFPE, Cidade Universitária, 50.000, Recife-PE, Brasil.

1. Introdução: Porque Linguagens Funcionais?

Nesta Introdução, o nosso objetivo é mostrar porque as linguagens funcionais são necessárias à atividade de produção de software, visto como ciência ou engenharia.

Há muito tempo que se fala na *crise de software*. Esta, para a computação, pode ser comparada à crise do petróleo, na década de 70, em relação aos seus custos e efeitos na sociedade.

O custo decrescente de hardware, juntamente com o aumento de complexidade dos pacotes de software, e consequente aumento dos custos de especificação, escrita e teste dos mesmos, faz com que se estime, hoje em dia, uma razão de 4 para 1 entre os custos de software e hardware em sistemas integrados [Boehm 81].

Claramente, a ciência de software deveria buscar uma solução para tal problema, da mesma forma que VLSI é uma solução, nem que seja temporária, para o custo de hardware. Analisando o desenvolvimento das linguagens de programação de uso geral, dos anos 50-60 até agora, vê-se que estas não mudaram muito, e em muitos casos, mudaram para pior.

Linguagens de programação deveriam ser usadas para especificar, de forma executável, o problema que o programador quer resolver. Isto significa que uma boa linguagem de programação tem que ser *programmer-friendly*: afinal de contas, ela teria sido criada para facilitar a vida do programador.

Agora consideremos o caso de ADA^{TM} [Honeywell 80]. ADA foi desenvolvida quando já se sabia das causas e efeitos da *crise de software*, e um dos seus objetivos é justamente contornar as causas da *crise*. No entanto, numa análise de complexidade da linguagem, Bjørner e Oest (em [Bjørner et al 80]) dão um grau de complexidade 8 para ADA, relativo a 1 para ALGOL60 e 5 para PL/I. Como os programadores que já tentaram escrever um programa em PL/I devem se lembrar, ela não é exatamente uma linguagem que se possa chamar de fácil, e isto coloca ADA numa posição bastante incômoda.

Aumentando a complexidade da linguagem, aumenta-se a dificuldade de uso, juntamente com a probabilidade de erros de programação, e, talvez muito pior, torna-se impossível a manipulação formal de programas –como transformação e prova de correção. O resultado óbvio é um aumento, *e não diminuição*, do custo de software. Uma possível conclusão é que a mais provável contribuição de linguagens como ADA é agravar, e não resolver, ou mesmo amenizar, a *crise de software*.

Ao invés de linguagens cada vez mais complicadas, as quais, à parte das críticas acima, também não ajudam a produtividade do programador, o que se precisa é de linguagens **simples**, mas que englobem conceitos (de programação) mais poderosos. Este é o caso do estilo tratado neste artigo. A seguir, apresentaremos alguns aspectos da linguagem de programação KRC (Kent Recursive Calculator), desenvolvida na University of Kent at Canterbury, UK, pelo Prof. David Turner. A apresentação é informal e segue a mesma linha de [Turner 82b]. A sintaxe e semântica denotacional formais da linguagem estão especificadas em [Meira 85t].

2. Programação Denotacional[1]

O termo programação denotacional é usado para acentuar o fato de que a nossa preocupação é com a real forma dos objetos dos quais as nossas definições tratam, e não com o processo através dos quais eles são construídos. Um exemplo disso é a definição, em KRC, da função *fatorial*:

$$fatorial\ 0 = 1$$
$$fatorial\ n = n * fatorial\ (n-1).$$

A primeira equação define o caso base para a função, ou seja, o *fatorial* de *0* é *1*. A segunda equação define os casos restantes, recursivamente, e é claro que *fatorial* é uma função total em N, o domínio dos números naturais. Note o uso de *pattern matching* na primeira equação, funcionando como um condicional.

O estilo informal de programação que discutimos aqui é baseado em equações recursivas de alta ordem (≥ 1), não estrito, e usa conceitos da teoria dos conjuntos de Zermelo-Fraenkel [Mendelson 79].

[1] Termo devido a Christopher Strachey.

Neste contexto, não estrito significa que os parâmetros reais de definições, semanticamente, são passados por nome, ou que o mecanismo de execução da linguagem é "preguiçoso" ou *lazy*. Não há atribuição (*assignment*) de valores a variáveis –porque não há conceito de armazenamento, efeitos colaterais não existem, e também não há conceito de controle como em linguagens imperativas.

Em KRC, funções são *cidadãos de primeira classe*, i.e., podem tanto ser argumento como resultado de outras funções. Todas as funções são tratadas como funções de apenas 1 parâmetro. Por exemplo,

$$some\ a\ b = a + b$$

é uma função de um parâmetro que, quando aplicada somente ao parâmetro real *77*, em *some 77*, retorna a função (de 1 parâmetro)

$$(some\ 77)\ b = 77 + b.$$

Tal dispositivo é chamado *currying*[2], e é de fundamental importância para o estilo de programação funcional. KRC é uma linguagem "estática": os valores de objetos descritos na linguagem não mudam com o tempo.

À parte de objetos escalares como números, *strings*, funções e *booleans*, KRC lida com um único tipo estruturado, a **lista**. Listas são coleções de objetos de qualquer tipo –KRC é *weakly-typed*, ver [Strachey 72]– e exemplos básicos de listas são:

$$["semish", 1985] \qquad [fatorial, 12, \mathbf{true}] \qquad [\,],$$

o último dos quais é a lista vazia ou **nil**. Algumas operações definidas sobre listas são:

$a : x$ adiciona o objeto a à frente da lista x, é equivalente à operação (**cons** a x) em LISP;

$hd\ x$ retorna o primeiro objeto da lista x, isto é, o *head* de x;

$tl\ x$ retorna a lista x a menos do seu primeiro objeto, i.e., *tail* de x.

O construtor ":" também pode ser usado em definições baseadas em "pattern matching", de tal forma que podemos definir

$$hd\ (a : x) = a$$
$$tl\ (a : x) = x,$$

e tanto *hd* como *tl* da lista vazia são indefinidos, ou, semanticamente, $\perp$ (*bottom*).

Usando tais conceitos básicos, a função $\sum$ (somatória) sobre uma lista de números pode ser definida como:

$$somat\ [\,] = 0$$
$$somat\ (a : x) = a + somat\ x.$$

A explicação desta definição é trivial. Se assumirmos que **num** representa os objetos de tipo *número* e [*] representa o tipo *lista de* *, o tipo da definição acima é

$$somat :: [\mathbf{num}] -> \mathbf{num},$$

onde "::" é lido *tem tipo*.

Um exemplo de uso de funções de alta ordem é a definição de *map*

$$map\ f\ [\,] = [\,]$$
$$map\ f\ (a : x) = f\ a : map\ f\ x.$$

map é uma função de tipo

$$(* -> **) -> [*] -> [**].$$

[2]Em homenagem a Haskell B. Curry.

O resultado da aplicação *map g l* é aplicar a função *g* a todos os elementos da lista *l*. Para o tipo de dados [[**num**]], ou seja "lista de lista de **num**" (representando matrizes...), pode-se definir a função

$$matsom = map\ somat,$$

que computa a somatória das linhas (poderia ser colunas) da "matriz" à qual é aplicada.

Outra definição que faz uso do mesmo conceito é

$$\begin{aligned} filter\ cond\ [\,] &= [\,] \\ filter\ cond\ (a : x) &= a : filter\ cond\ x, \quad cond\ a \\ &= filter\ cond\ x, \end{aligned}$$

que também serve para demonstrar o uso de equações condicionais (*guarded equations*) em KRC. O resultado de *filter* é a lista de entrada **menos** os elementos para os quais o teste *cond a* é **false**. Por exemplo, para

$$impar\ b = b\ \%\ 2\ \backslash =\ 0,$$

onde $a\ \%\ b$ indica o resto da divisão de a por b e $\backslash =$ representa *diferente de*, podemos definir

$$impares = filter\ impar,$$

a função que elimina números pares da sua lista-argumento.

Note como a junção de *map* e *somat* para formar *matsom* e de *filter* e *impar* para formar *impares* é extremamente simples e natural. Esta modularização e acoplamento, que é possível no tipo de linguagem do qual KRC é um representante, é peça fundamental do estilo de programação que defendemos. Em linguagens imperativas como FORTRAN ou ADA, o exercício acima não pode ser resolvido com a mesma facilidade, e certamente necessitará reescritura do programa inicialmente desenvolvido para implementar *filter* ou *map*. Devido à ausência do conceito de funções de alta ordem, pode-se dizer que *filter* e *map* não *podem ser implementados* em tais linguagens.

Mais um exemplo: *Sorting*

Considere a definição de ordenação por inserção (*insertion sorting*). Esta pode ser implementada em dois passos. Primeiro, define-se uma função que insere um objeto (comparável sob $\leq$) em uma lista já ordenada (sob $\leq$):

$$\begin{aligned} insert\ a\ [\,] &= [a] \\ insert\ a\ (b : x) &= a : b : x, \quad a <= b \\ &= b : insert\ a\ x. \end{aligned}$$

Em segundo lugar, definimos a função que cuida da ordenação

$$\begin{aligned} insort\ [\,] &= [\,] \\ insort\ (a : x) &= insert\ a\ (sort\ x). \end{aligned}$$

Dignos de nota, nesta definição, são a clareza com que o algoritmo foi especificado e a facilidade com que se pode derivar uma prova de sua correção total (ver [Meira 82] para tal). É importante notar também que a complexidade, no tempo, desta definição é a mesma do programa imperativo que implementa o mesmo algoritmo usando *arrays* (ver [Sedgewick 83] por exemplo). A complexidade espacial da definição funcional acima é discutida na Seção 6.

3. Expressões de Zermelo-Fraenkel

Estas formam uma das partes mais interessantes da linguagem KRC. Definições usando expressões ZF são extremamente compactas e auto-explicativas, às vezes ao ponto de serem, ao mesmo tempo, a

especificação, solução e documentação de um problema. Tome como primeiro exemplo a definição ZF da função *impares* já vista anteriormente:

$$impares\ li = \{n \mid n\ <-\ li;\ impar\ n\}.$$

Esta definição é lida "a lista de todos os objetos n tirados da lista li tal que *impar* n".

Um exemplo mais complexo é a definição da função que gera todas as permutações dos elementos de uma lista li,

$$perms\ [\,] = [[\,]]$$
$$perms\ li = \{a : p \mid a\ <-\ li;\ p\ <-\ perms\ (li\ --\ [a])\},$$

onde $--$ é o operador *diferença* sobre listas (e.g. $[1,2,1,3]\ --\ [1] = [2,1,3]$). *perms*, juntamente com um predicado que decide a ordenação de listas (sob $\leq$)

$$orden\ li = and\ \{li\ i\ <=\ li\ (i+1) \mid i\ <-\ [1..\#li - 1]\},$$

onde

$li\ i$ é o i-ésimo elemento da lista li;

$[n..m]$ é a lista $[n, n+1, \cdots, m]$;

$\#li$ é o tamanho da lista li e

$$and\ [\,] = \mathbf{true}$$
$$and\ (\mathbf{true} : x) = and\ x$$
$$and\ (\mathbf{false} : x) = \mathbf{false}$$

pode ser usada para especificar o problema de ordenação sem usar nenhum "algoritmo" específico:

$$sort\ li = hd\ \{p \mid p\ <-\ perms\ li;\ orden\ p\}.$$

Apesar de executável, esta definição tem complexidade fatorial, o que a torna absolutamente inútil na prática.

Para um algoritmo bem mais rápido ($\mathbf{O}(n \log n)$), que encerra os exemplos e a apresentação do estilo de programação, considere nossa definição do algoritmo de *quick sort* [Hoare 62], usando expressões ZF:

$$quick\ [\,] = [\,]$$
$$quick\ (a : x) = quick\ \{b \mid b\ <-\ x;\ b\ <=\ a\}\ ++\ a : quick\ \{b \mid b\ <-\ x;\ b > a\}.$$

A beleza e concisão desta definição é melhor apreciada quando em comparação com definições imperativas do mesmo algoritmo (ver [Wirth 76], pp. 76).

4. Complexidade de Algoritmos Funcionais

A seguir nós consideramos a complexidade das definições funcionais de *fatorial*, *insertion* e *quick sort*, comparando à complexidade dos equivalentes imperativos.

O modelo de execução que adotamos para a análise de complexidade é o de sistemas "preguiçosos" de reescritura (*lazy term rewrite systems* [Huet & Oppen 80]), que é ótimo em relação aos problemas aqui discutidos. Modelos práticos de implementação de linguagens funcionais, tais como *combinadores* [Turner 79a] terão eficiência necessariamente mais baixa, pelo menos por um fator constante (ver [Meira 85t]). Dito isto, a análise aqui apresentada se refere ao limite inferior da complexidade média das definições.

A palavra preguiçoso acima se refere ao fato de que os termos em questão são reescritos em uma ordem *mais-à-esquerda/mais-externa*, e também (implicitamente) que qualquer sub-termo compartilhado é computado no máximo uma vez. Este mecanismo informal é equivalente ao *fully lazy normal graph reduction* para o λ-cálculo [Wadsworth 71] e lógica de combinadores [Turner 81a]. Agora passemos à análise prometida, onde $\mathbf{T}$ é a complexidade temporal e $\mathbf{R}$ é a complexidade espacial em $\mathrm{P_reg}$, o nosso modelo de execução.

5. Fatorial

A nossa definição funcional de *fatorial* é linear no tempo -$\mathbf{T}$ (*fatorial* n) = $\mathbf{O}(n)$- e no espaço --$\mathbf{R}$ (*fatorial* n) = $\mathbf{O}(n)$. O último fato se deve à mesma razão da definição imperativa recursiva ser linear em espaço. Basicamente, para se calcular *fatorial* n, é necessário trabalhar ao contrário e descobrir primeiro o fatorial de 0, 1, etc. Assim sendo, tem-se

$$fatorial\ n = \underbrace{n * ((n-1) * (\cdots * 1))}_{n+1\ sub-termos}$$

o que implica em $\mathbf{R}(fatorial\ n) = \mathbf{O}(n)$. A história, no entanto não acaba aqui.

Como todo principiante sabe, uma alternativa mais eficiente do que o programa imperativo recursivo é um programa *for*,

$$f := 1;$$
$$\textbf{for } c := 1 \textbf{ to } n \textbf{ do } f := f * c;$$

para o qual a complexidade espacial é constante.

Uma alternativa imediata para melhorar a eficiência da definição funcional é reescrevê-la imitando o programa acima, o que é possível usando *tail recursion*:

$$fatorial\ n = tfat\ n\ 1$$
$$tfat\ 0\ res = res$$
$$tfat\ n\ res = tfat\ (n-1)\ (n * res).$$

Infelizmente, a nova definição tem a mesma complexidade da definição puramente recursiva apresentada anteriormente. Em KRC, como na maioria das linguagens não estritas, os parâmetros reais das definições são passados "por necessidade" (*call-by-need*), isto é, com a mesma semântica de *call-by-name*, mas computados uma única vez, quando seu valor é requisitado pelo mecanismo de execução. O segundo parâmetro de *tfat* só tem seu valor requerido quando do término (e consequente impressão) do cálculo de *tfat*. Daí, o cálculo procede segundo os estágios a seguir:

$$fatorial\ n = tfat\ n\ 1 = tfat\ (n-1)\ (n * 1) =$$
$$= tfat\ (n-2)\ ((n-1) * (n * 1)) = \cdots = tfat\ 0\ \underbrace{(1 * \cdots ((n-1) * (n * 1)))}_{n\ termos}$$

com o segundo parâmetro crescendo em tamanho a cada nova chamada recursiva à *tfat*. O primeiro parâmetro ocupa espaço constante, porque é computado a cada chamada, para decidir a terminação de *tfat* (em *tfat 0 res = res*).

Na realidade, este comportamento só será eliminado se conseguirmos forçar a computação do segundo parâmetro de *tfat* a cada chamada. Tal procedimento é seguro, pois preserva a semântica da definição, dado que *tfat n res* $\neq \perp$ se e somente se tanto n como *res* são diferentes de $\perp$. Assim, pode-se escrever

$$tfat\ n\ res = res, \quad res > 0\ \&\ n = 0$$
$$= tfat\ (n-1)\ (n * res).$$

Analisando a condição na primeira equação, tem-se

$$res > 0\ \&\ n = 0 \equiv \textbf{true}\ \&\ n = 0 \equiv n = 0,$$

supondo que o domínio de *fatorial* é $\mathbb{N}$. A semântica de &,

$$\perp\ \&\ exp = \perp$$
$$\textbf{true}\ \&\ exp = exp$$
$$\textbf{false}\ \&\ exp = \textbf{false},$$

garante que o segundo parâmetro de *tfat* é sempre computado (porque ele é o primeiro de & e este é sempre computado). Consequentemente, $\mathbb{R}$ (*fatorial* n) = $\mathbb{k}$.

Uma crítica séria a este tipo de procedimento é que ele vai contra os princípios de programação denotacional advogados neste artigo. Em outras palavras, o desenvolvimento anterior mostrou que o programador tem que se preocupar com o método de construção dos objetos com que trata, de forma a tornar sua computação mais eficiente. Enquanto reconhecemos que este possa ser um dos calcanhares de Aquiles deste estilo de programação, a conversa não acaba aqui, e a contra-argumentação tem três aspectos diferentes:

1- Justamente porque a primeira aproximação à solução foi denotacional, é possível aplicar com mais facilidade transformações que tornam a *especificação* mais eficiente, e provar que tais transformações não modificaram o significado da definição por onde começamos;

2- Existem métodos automáticos que podem lidar com, por exemplo, transformação de *call-by-need* para *call-by-value* [Meira 84, Mycroft 81]. Um destes métodos, aplicado a *tfat*, tornará aquela definição tão eficiente quanto a que conseguimos adicionando o teste redundante no condicional;

3- Em linguagens nas quais o programador pode definir tipos abstratos de dados e seus construtores, com a facilidade especial de especificar *leis* que guiam o uso de construtores, como MIRANDA [Turner 84] o trabalho de transformação de programa pode fazer parte da fase de especificação do tipo de dados com que se lida. Por exemplo, se o operador $*$ de um tipo de dado **inteiro** é definido como sendo associativo, a primeira definição de *fatorial* que usamos tem residência constante. Para outros exemplos usando a mesma idéia, ver [Meira 85e].

6. Insertion Sorting

O leitor atento já deve ter descoberto que a definição de *insertion sorting* na Seção 2 tem complexidade espacial linear. Para ordenar uma lista $[l_1, l_2, \cdots, l_n]$ usando *insort*, o mecanismo de execução primeiro cria o fechamento

$$insort\ [l_1, l_2, \cdots, l_n] = insert\ l_1\ (insert\ l_2\ (\cdots (insert\ l_n\ [\,]))),$$

que ocupa espaço $\mathbb{O}(n)$. Como a lista resultante da ordenação tem tamanho n, o uso de espaço por *insort* é constante, $\mathbb{O}(n)$.

O algoritmo imperativo, usando *arrays*, tem complexidade espacial constante (ver [Sedgewick 83]). Para ter complexidade constante, o algoritmo funcional precisa ser modificado de tal forma que o fechamento discutido acima não é criado. Novamente, usando *tail recursion*, definimos

$$\begin{aligned} trinsort\ [\,] &= [\,] \\ trinsort\ (a : x) &= trins\ [\,]\ [a]\ x, \end{aligned}$$

onde *trins* é uma função de tipo

$$trins :: lo -> lo -> [*] -> lo,$$

e *lo* representa o tipo "lista ordenada de $*$". Uma possível definição de *trins* é

$$\begin{aligned} trins\ l\ [\,]\ (b : y) &= trins\ [\,]\ (l\ +\!+\ [b])\ y \\ trins\ l\ x\ [\,] &= l\ +\!+\ x \\ trins\ l\ (a : x)\ (b : y) &= trins\ (l\ +\!+\ [a])\ x\ (b : y), \quad a < b \\ &= trins\ [\,]\ (l\ +\!+\ b : a : x)\ y. \end{aligned}$$

A prova de correção desta definição por indução estrutural [Burstall 69, Meira 82] é imediata.

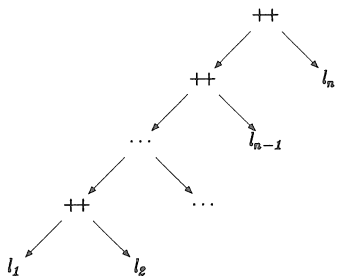
Assumindo que o espaço usado por $[a]\ +\!+\ b$ é o mesmo usado por $a : b$ (obviamente, se b é uma lista, $[a]\ +\!+\ b = a : b$), esta definição usa espaço constante, porque em qualquer estágio da computação o

espaço ocupado por $l \mathbin{+\!\!+} a : x \mathbin{+\!\!+} b : y$ é o mesmo ocupado pela lista argumento de *trinsort*. Resolvido o problema de complexidade espacial, mas infelizmente criamos um problema no tempo, devido ao uso do operador $+\!\!+$ na terceira e quarta equações de *trins*.

Considere a computação de

$$((l_1 \mathbin{+\!\!+} l_2) \mathbin{+\!\!+} \cdots) \mathbin{+\!\!+} l_n,$$

um fechamento da mesma forma que o criado em *trins*. Este é representado pela árvore (degenerada à esquerda)



Por sua vez, o operador $+\!\!+$ (*append*, em forma prefixa) é especificado pela definição

$$\begin{aligned} append\ [\,]\ y &= y \\ append\ (a : x)\ y &= a : append\ x\ y, \end{aligned}$$

donde se conclui que a lista à esquerda de $+\!\!+$ tem que ser percorrida até o último elemento, para que se compute $l_i \mathbin{+\!\!+} l_j$. Na árvore acima, as seguintes listas são percorridas *in totum*:

$$l_1, \quad l_1 \mathbin{+\!\!+} l_2, \quad l_1 \mathbin{+\!\!+} l_2 \mathbin{+\!\!+} l_3, \quad \cdots \quad l_1 \mathbin{+\!\!+} l_2 \mathbin{+\!\!+} l_3 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} l_{n-1}$$

para que se compute a lista final. Isso implica numa complexidade temporal $\mathbb{O}(n^2)$ para tal computação. Em *trins*, um fechamento deste tipo tem que ser computado a cada passo. Em n passos, teremos uma complexidade temporal $\mathbb{O}(n^3)$, o que faz o algoritmo uma ordem de magnitude mais lento que o seu equivalente imperativo.

Para resolver tal problema, considere primeiro a definição

$$\begin{aligned} reverse\ [\,] &= [\,] \\ reverse\ (a : x) &= reverse\ x \mathbin{+\!\!+} [a], \end{aligned}$$

que especifica a função que inverte uma lista, i.e., $l'_1 = l_n$, $l'_2 = l_{n-1}$, etc. A complexidade temporal de *reverse* é $\mathbb{O}(n^2)$, pela mesma razão acima. Mas se a reescrevermos como

$$\begin{aligned} reverse\ x &= rev\ x\ [\,] \\ rev\ [\,]\ y &= y \\ rev\ (a : x)\ y &= rev\ x\ (a : y), \end{aligned}$$

sua complexidade temporal é linear em $\#x$. Aplicando a mesma idéia à definição de *trins*, e usando *reverse* para inverter a lista criada em cada ciclo por *trins* (porque $reverse\ (reverse\ x) = x$), tem-se

$$\begin{aligned} trins\ l\ [\,]\ (b : y) &= trins\ [\,]\ (reverse\ l \mathbin{+\!\!+} [b])\ y \\ trins\ l\ x\ [\,] &= reverse\ l \mathbin{+\!\!+} x \\ trins\ l\ (a : x)\ (b : y) &= trins\ (a : l)\ x\ (b : y), \quad a < b \\ &= trins\ [\,]\ (reverse\ l \mathbin{+\!\!+} b : a : x)\ y. \end{aligned}$$

A complexidade temporal de *trins* agora é $\mathbb{O}(n^2)$, e a espacial é constante, ou seja, nossa definição tem a mesma eficiência do algoritmo imperativo. O leitor pode verificar que as tranformações aplicadas preservam a semântica de *trins*.

7. Quick Sort

O último caso a ser analisado é a definição de *quick sort*. Em implementações imperativas, pode-se ter uma versão deste algoritmo de tal forma que a complexidade temporal média é $\mathbb{O}(n \log n)$, com pior caso $\mathbb{O}(n^2)$, e o pior caso para a complexidade espacial é $\mathbb{O}(\log n)$, quando se elimina recursão. Relembrando nossa definição de *quick sort*

$$\begin{aligned} &quick\ [\,] = [\,] \\ &quick\ (a : x) = quick\ \{b \mid b\ <-\ x;\ b\ <=\ a\}\ ++\ a : quick\ \{b \mid b\ <-\ x;\ b > a\}, \end{aligned}$$

observa-se, em primeiro lugar, que ela não faz nenhuma concessão à eficiência. Por exemplo, a partição das listas é feita simplesmente percorrendo cada lista duas vezes para extrair os elementos $\leq a$ e $> a$. Como já foi visto em casos anteriores, também há uma componente quadrática na complexidade – agora menos séria– devido à complexidade de computação de séries de $++$'s.

Considerando que o problema gerado por $++$ pode ser resolvido como nos casos anteriores, ou usando outras regras de computação para $++$ [Sleep & Holmstrom 82], vamos concentrar a análise no que nos parece o principal problema desta definição, que é o uso de espaço. Primeiro considere o melhor caso, em termos de complexidade espacial, que é a lista de entrada já ordenada (este é o pior caso no tempo!). Para simplificar a análise, assume-se que todos os elementos da lista são diferentes entre si. Neste caso, para a lista ordenada, todas as sub-listas geradas à esquerda de $++$ são vazias, e as listas à direita tem tamanho $n-1$, $n-2$, $\cdots$, 1, e somente uma delas existe em um determinado tempo. A complexidade espacial *total*, considerando que a lista de entrada é impressa à medida que o resultado é computado é $\mathbb{O}(n)$, máxima. Na verdade, o espaco extra usado é constante, assim $\mathbb{R}(quick\ l) = \mathbb{k}$.

O pior caso para a complexidade espacial é quando o inverso da lista ordenada é a entrada de *quick*. Agora, as sublistas à esquerda de $++$ têm tamanho $n-1$, $n-2$, $\cdots$, 1, somente uma existindo em um determinado momento. As sub-listas à direita de $++$ são todas vazias, mas, *como o mecanismo de execução é preguiçoso*, elas não são computadas até que a necessidade obriga a tal. Isto significa que fechamentos contendo sub-listas de tamanho $n-1$, $n-2$, $\cdots$, 1, são retidos até que a computação de $++$ exija sua redução, o que implica numa complexidade espacial $\mathbb{O}(n^2)$.

Em média, se a árvore de $++$'s é balanceada, a complexidade espacial é dada pela fórmula $\phi(n) = \phi(n/2) + n$, ou $\mathbb{R}(quick\ l) = \mathbb{O}(2 * (\#l))$, o que é muito pobre comparado com o melhor algoritmo imperativo.

Observando que a causa deste comportamento é *preguiça*, nós podemos derivar uma solução de complexidade espacial $\mathbb{O}(n)$, forçando a redução das listas em todos os níveis, independentes delas estarem à esquerda ou à direita de $++$. Isto é feito em

$$\begin{aligned} quicker\ [\,] &= [\,] \\ quicker\ [a] &= [a] \\ quicker\ (a : x) &= split\ a [\,]\ [\,]\ x \\ split\ a\ left\ right\ [\,] &= quicker\ left\ ++\ a : quicker\ right \\ split\ a\ left\ right\ (b : x) &= split\ a\ (b : left)\ right\ x, \quad b\ <=\ a \\ &= split\ a\ left\ (b : right)\ x, \end{aligned}$$

que é $\mathbb{O}(n)$ em espaço no pior caso. Não faz sentido, em uma linguagem recursiva, eliminar recursão, a não ser em termos dos detalhes de implementação da linguagem. Assim sendo, não tentaremos melhorar a eficiência deste algoritmo ainda mais, apesar de o nosso arsenal de transformações ainda não ter se esgotado (ver [Meira 85t]). Em termos práticos, a definição acima é comparável à melhor definição imperativa, e este era o objetivo inicial deste exercício: mostrar que é possível, usando linguagens como KRC, ter todas as vantagens que podem ser tiradas da sua elegância semântica, sem necessariamente ter que usar algoritmos e definições impraticáveis.

8. Conclusão

Este artigo mostra uma das pontas do iceberg de programação funcional, a transformação de especificações em definições eficientes. Em alguns casos, esta tranformação não é necessária. Um exemplo disso é a definição de *perms* na Seção 3, que é teoricamente quase ótima.

Nos casos em que transformações são necessárias, elas são realizadas usando o mesmo formalismo (a linguagem de programação), mantendo o mesmo ambiente usado na especificação, o que certamente ajuda no desenvolvimento de programas. Apesar de termos usado um método informal neste artigo, todas as transformações usadas podem ser realizadas formalmente, passo a passo, acompanhadas de uma prova de correção de cada nova equação gerada. Dado à complexidade de tal exercício, ele é viável somente quando um sistema automático possa verificar a validade das transformações utilizadas. Tal sistema – que já existe para prova de correção de programas, LCF [Milner et al. 77] é um exemplo, poderia mesmo aplicar transformações elementares sem interferência do programador.

Finalizando, temos a dizer que, tendo sido ganho o argumento filosófico e psicológico em favor de linguagens funcionais como KRC e MIRANDA para desenvolvimento de software, em detrimento de FORTRAN e coisas do mesmo tipo, resta agora vencer a barreira que ainda impede o uso em larga escala de linguagens funcionais: eficiência e velocidade. Este trabalho é uma contribuição à solução do primeiro problema.

9. Agradecimentos

Este artigo foi composto em LaTeX, e impresso numa Canon LBP-10 Laser Printer, no Computing Laboratory da University of Kent at Canterbury. Gostaria de registrar meus agradecimentos ao Computing Lab por ter me dado acesso aos recursos acima.

Agradeço a Pedro Silveira pela ajuda na tradução de alguns termos técnicos para o Português.

Este trabalho teve o auxílio financeiro do CNPq, Proc. 200.510/81, e do Departamento de Eletrônica e Sistemas da UFPE.

Referências

Bjørner et al 80
BJØRNER, B. B., LOVEGREEN, H. H., BUNDGAARD, J., DOMMERGAARD, O. H., OEST, O. H., PEDERSEN, J. S. and SCHULTZ, L., *Towards a Formal Description of ADA*. ed.: D. B. Bjørner and O. N. Oest. Springer-Verlag, Berlin, (1980).

Boehm 81
BOEHM, B. W., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, (1981).

Hoare 62
HOARE, C. A. R., Quicksort. *Comp. J.* **5**, (1962), pp. 10-15.

Honeywell 80
CII HONEYWELL BULL, *Reference Manual for the ADA Programming Language. Proposed Standard Document.* (1980).

Huet & Oppen 80
HUET, G. and OPPEN, D. C., Equations and Rewrite Rules: A Survey. *in Formal Language Theory, Perspectives and Open Problems*, ed.: Book, R. V. (1980), Acad. Press, New York,

Burstall 69
BURSTALL, R. M., Proving Properties of Programs by Structural Induction. *Comp. J.* **12**, (1), (Feb. 1969), pp. 41-48.

Meira 82
MEIRA, S. L., Sorting Algorithms in KRC: Implementation, Proof and Performance. *Comp. Lab. Rep. 14*, Computing Laboratory, University of Kent, Canterbury, UK, (1982).

Meira 84

MEIRA, S. L., Optimized Combinatoric Code for Applicative Language Implementation. *in Proc. of the VI Intl. Symp. on Programming*, Springer Verlag, *Lec. Notes in Comp. Sci.*, Heidelberg, (1984).

Meira 85t

MEIRA, S. L. *On the Efficiency of Applicative Algorithms. Ph. D. Thesis*, Computing Laboratory, University of Kent, Canterbury, UK, (1985).

Meira 85e

MEIRA, S. L. Efficient Algorithms in MIRANDA. *in preparation*, Dep. Elet. e Sistemas, UFPE, Recife-PE, Brasil, (1985).

Mendelson 79

MENDELSON, E., *Introduction to Mathematical Logic.* Van Nostrand, New York, (1979).

Milner et al. 77

MILNER, R., WADSWORTH, C. and GORDON, M., Edinburgh LCF. *CSR-11-77, Parts I and II*, Dep. Comp. Sci, University of Edinburgh, Edinburgh, Scotland, (1977).

Mycroft 81

MYCROFT, A., *Abstract Interpretation and Optimising Transformations for Applicative Programs. Ph. D. Thesis*, Dep. of Comp. Sci., Univ. of Edinburgh, Edinburgh, (1981).

Sedgewick 83

SEDGEWICK, R., *Algorithms.* Addison-Wesley, Reading, Mass., (1983).

Sleep & Holmstrom 82

SLEEP, M. R. and HOLMSTROM, S., *A Short Note Concerning Lazy Evaluation Rules of APPEND. Univ. of East Anglia Internal Report*, Norwich, U. K., (May 1982).

Strachey 72

STRACHEY, C., Varieties of Programming Language. *Infotech State of the Art Report on "High Level Languages" 7*, (1972).

Turner 79a

TURNER, D. A., A New Implementation Technique for Applicative Languages. *Soft. Pract. and Exp.* **9**, (1979), pp. 31-49.

Turner 81a

TURNER, D. A., *Aspects of the Implementation of Programming Languages. Ph. D. Thesis*, Brasenose College, Oxford University, Oxford, (1981).

Turner 82b

TURNER, D. A., Recursion Equations as a Programming Language. *in Functional Programming and its Applications, An Advanced Course*, C. U. P., Cambridge, UK, (1982).

Turner 84

TURNER, D. A., Functional Programs as Executable Specifications. *Phil. Trans. Royal. Soc. of London* **312**, (1522), (1984), pp. 363-388.

Wadsworth 71

WADSWORTH, C., *Semantics and Pragmatics for the Lambda Calculus. Ph. D. Thesis*, Dept. of Mathematics, Oxford Univ., Oxford, (1971).

Wirth 76

WIRTH, N. Algorithms + Data Structures = Programs. Prentice-Hall, Eng. Cliffs, NJ, (1976).