ESPECIFICAÇAO EXECUTAVEL DE UM FORMATADOR DE TEXTO

George Roger Justo

Silvio Lemos Meira

Departamento de Informática
Universidade Federal de Pernambuco
50739, Recife, PE

SUMARIO

A especificação formal vem se tornando uma ferramenta importante no desenvolvimento de sistemas mais confiáveis. Um problema, porém da sua aplicação é a escolha de uma metodologia de validação. Neste trabalho é utilizada a validação através da execução da especificação, implementada numa linguagem funcional. O principal mérito deste método é oferecer um meio fácil de encontrar erros de especificação sem, contudo, perder as principais propriedades matemáticas da especificação formal.

ABSTRACT

Formal specification has become an important tool for the development of reliable software. One of the problems of its application is the choice of a methodology for validation. In this paper, we execute the specification, implemented in a functional language, as a form of validation. The main advantage of this approach is to offer an easy way to find out errors in the specification without loosing the most of the mathemathical properties of the formal specification.

1. INTRODUÇAO

Uma especificação é uma descrição das propriedades requeridas para a solução de um problema. Devido ao grande número de problemas que pode ocorrer, está sendo questionada, há muito tempo, a utilização da linguagem natural na especificação de sistemas computacionais, pois esta conduz à perda das principais propriedades de uma especificação, ou seja, precisão, consistência, completude, compreensão e modularidade. O formalismo matemático é capaz de garantir estas propriedades de forma natural e, assim, permitir a execução eficiente desta fase importante do ciclo de vida do software.

Uma questão importante levantada quando se faz referência à aplicação da especificação formal é a escolha de uma metodologia de validação, principalmente quando se trata de especificações extensas. O caminho mais lógico é o estabelecimento de axiomas em uma teoria e, assim, provar teoremas acerca das propriedades requeridas pelos objetos e operações especificados. Uma alternativa, mais prática, seria a possibilidade de se desenvolver uma especificação executável. Desta forma, teríamos a possibilidade de validar a especificação através de teste do programa que a implementa. Assim, erros poderiam ser encontrados mais rapidamente no decorrer do projeto diminuindo os custos de desenvolvimento.

Neste trabalho, a executabilidade da especificação é conseguida através da utilização de uma metodologia de especificação funcional, de modo que se possa "implementar" a especificação numa linguagem funcional.

A escolha de programação funcional é decorrente da base matemática simples que esta oferece e que facilita o projeto de programas confiáveis num estilo funcional puro. Além disso, a programação funcional apresenta a clareza requerida para a especificação formal.

Serão introduzidos alguns conceitos sobre programação funcional através da apresentação da linguagem funcional HOPE [1,2], que será utilizada para a especificação do exemplo. No item seguinte, será discutido informalmente o problema que se propõe como exemplo que é um modelo simples de formatador de textos. Logo depois, será mostrada a especificação formal do problema, na forma funcional executável, utilizando a linguagem HOPE [1,2].

2. PROGRAMAÇÃO FUNCIONAL

As linguagens funcionais como paradigma de programação tem uma longa história, porém, por razões práticas relacionadas com a baixa capacidade dos computadores, não era viável construir programas em estilo funcional puro. Daí, era preciso misturar às formas basicamente matemáticas características de programação imperativa, como atribuição, para se obter melhor performance. Mas esta não foi a única razão para não se explorar as linguagens funcionais. O estágio em que a programação funcionl ainda se encontrava, há alguns anos, não oferecia boas ferramentas para desenvolvimento eficiente de programas. Vale ressaltar que a programação funcional enfrenta, ainda, o problema de muitas implementações que usam interpretadores para implementar arquiteturas não von Neumann em máquinas von Neumann. Este problema torna os programas mais lentos que uma linguagem imperativa compilada no mesmo hardware. Isto não ocorre em máquinas projetadas especialmente para implementar linguagens funcionais, como SKIM[3] e NORMA[2]. Devido a este tipo de problema, o objetivo deste trabalho não será a justificativa da utilização da programação funcional nos projetos de engenharia de sofware, mas como linguagem de especificação.

Serão apresentados, agora, alguns conceitos sobre programação funcional no estilo da linguagem HOPE [1,2] .

Uma linguagem como Pascal é dita imperativa porque programas escritos nela são "receitas para fazer alguma coisa". Quando se considera um program funcional a preocupação é com os resultados, desprezando como são computados, ou seja, são declarados quais são os resultados, ao invés de se descrever como obtê-los. Uma característica importante dos programas funcionais é a ausência de atribuições e, consequentemente, de efeitos colaterais, o que facilita o estudo formal dos programas.

Um "programa" em uma linguagem funcional compeende as definições das funções e as suas combinações. Em HOPE, a definição de uma função é dividida em duas partes, a definição do domínio (entrada) e do contra-domínio (saída) e depois a declaração do comportamento da função de acordo com as entradas. Na primeira parte se declara o domínio e o contra-domínio da função da seguinte forma:

```
dec <nome da função> : <domínio> -> <contra_domínio>;
```

por exemplo:

```
dec g : num -> num;

dec f : num X num -> num;

dec m : num -> num X char;
```

A segunda parte é a definição da função em si que é dada por uma ou mais equações recursivas. A forma geral é a seguinte:

```
--- <nome da função> (<elemento do domínio>)
    <= <equação que gera valores do contra-domínio>;
```

por exemplo:

```
--- g (x) <= x + 1;

--- f (x, y) <= if (x>y) then x else y;

--- m (x) <= if x > 0 then (x, 'a') else (x, ' ');
```

Note que a expressão condicional testa uma condição e escolhe entre valores alternativos e não entre ações alternativas.

HOPE tem alguns tipos primitivos como Truval (valores booleanos), Num (inteiros) e Char (caracteres ASCII). Além disso, existem dois tipos estruturados de dados, as tuplas e as listas. Uma tupla corresponde a um conjunto de objetos onde a posição de cada um é importante e eles podem ter tipos diferentes. Por exemplo:

```
(2, 3) : (num, num)

('a' , true ) : (char, truval)
```

As tuplas são usadas principalmente em funções que recebem ou devolvem vários objetos de tipos diferentes, ou na construção de registros (como os "records" de Pascal).

Uma lista corresponde a uma seqüência de objetos do mesmo tipo, por exemplo:

```
[1, 2, 3] : list(num)

['a', 'b'] : list(char)
```

O construtor '::' define uma lista em termos de um objeto e uma lista:

```
1 :: [2, 3] constrói a lista [1, 2, 3]
```

Existem algumas maneiras de resumir a construção das listas:

1) a lista vazia é representada por []

2) 1::(2::(nil)) <==> [1, 2]

3) 'A'::('B'::('C'::NIL)) <==> ['A', 'B', 'C'] <==> "ABC"

Estes são os conceitos básicos que permitem definir funções em HOPE. Mostramos a seguir um exemplo ilustrativo que obtém a primeira palavra de uma cadeia, cujas palavras estão separadas por espaços.

```
dec palavra : list(char) -> list(char);

--- palavra(nil) <= nil;

--- palavra(c::s) <= if c=' ' then nil else c::palavra(s);
```

Não discutimos todas as características da linguagem, pois o objetivo foi apenas dar a idéia de um programa funcional e, principalmente, mostrar a filosofia da linguagem HOPE de modo a facilitar o entendimento das funções que serão mostradas no decorrer do texto.

3. O FORMATADOR - DEFINIÇÃO INFORMAL

O objetivo da especificação será definir um formatador de textos simples. O formatador deverá impedir, sempre que possível, a existência de "buracos" no final das linhas do texto, devendo o texto de saída conter no máximo l caracteres por linha.

A entrada do formatador é um texto contendo um conjunto de linhas que podem ter tamanhos diferentes, sendo uma linha uma seqüência de caracteres ASCII, exceto CR. O formatador deverá rearrumar o texto de modo que as linhas de saída tenham o mesmo tamanho (l caracteres), exceto quando a linha é a última de um parágrafo ou é a última linha do texto. Ao receber uma linha de entrada, ele vai formando a linha de saída a partir da concatenação das palavras (seqüência de caracteres ASCII, exceto espaço ou CR) da linha de entrada até ocorrer uma das seguintes condições:

i. a linha de saída atingiu o tamanho l ;

ii. a linha de saída não atingiu o tamanho l , porém se concatenarmos a próxima palavra da linha de entrada o tamanho ultrapassará l caracteres;

iii. não existe mais nenhuma palavra na linha de entrada.

No caso (i), restando ainda palavras na linha de entrada a serem processadas (restante da linha), ou no caso (ii), estas palavras são concatenadas à próxima linha de entrada.

No caso (iii), o formatador deverá obter a próxima linha de entrada para continuar a formatação da linha.

Se p palavras cabem em uma linha de saída, separadas por um espaço, e o número de espaços remanescentes r é menore que o tamanho da próxima palavra de entrada, como na condição (ii) anterior, então os r espaços serão distribuídos uniformemente entre as palavras. Se a divisão (r/(p-1)), ou seja, a quantidade de espaços remanescentes pelo número de espaços entre as palavras, não for inteira, então o resto da divisão será distribuído de um em um da esquerda para a direita.

O único comando do formatador é o de definição de parágfrafo, cuja sintaxe é representada através da colocação de um espaço no início da linha. Quando um parágrafo é solicitado, então a linha de saída atual é terminada e uma linha em branco é inserida no texto, sofrendo a próxima linha de saída uma indentação de i caracteres.

Alguns casos serão tratados especialmente. O primeiro diz respeito à existência de uma palavra cujo comprimento seja superior ao tamanho de uma linha formatada (l caracteres). Neste caso, uma mensagem de erro deve ser exibida e o processo de formatação encerrado. Um outro problema é a existência de uma linha que só contém uma palavra cujo tamanho seja inferior a l caracteres, ou seja, uma linha contendo esta palavra e a palavra seguinte tem tamanho superior a l caracteres. Neste caso, a linha de saída só poderá conter uma palavra e não é possível evitar a ocorrência de buracos no final da linha, pois não existe espaço entre palavras para permitir a distribuição dos espaços restantes.

4. A ESPECIFICAÇAO FORMAL

A primeira questão a ser resolvida para a definição das funções é determinar a organização da entrada e saída, ou seja, os domínios e contra-domínios das funções. Isto corresponde à definição dos tipos de dados que melhor se adaptam ao comportamento do problema.

No caso do formatador, se optou pela representação do texto como uma lista de listas de caracteres, ou seja, o texto é formado por uma lista de linhas, onde cada linha é representada por uma lista de caracteres, tal como:

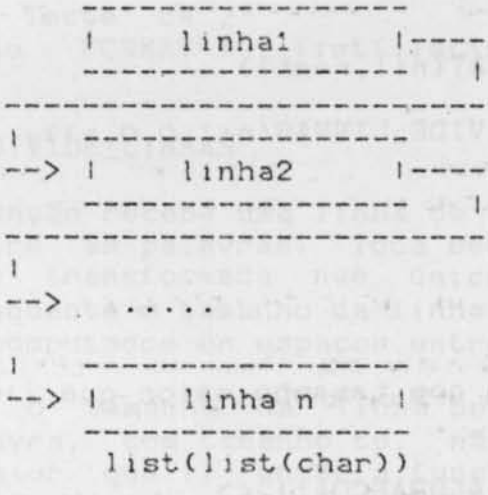

Fig 4.1 Representação do Texto

A especificação está, basicamente, definida através de três funções: FORMAT, DIVIDE_LINHAS e FORMLINHA, descritas a seguir.

4.1 FUNÇAO FORMAT

O objetivo desta função é tomar o texto de entrada do tipo list(list(char)) e desmembrar cada linha, de modo a verificar a existência ou não do comando que define um parágrafo. Caso o comando exista, deve ser inserida uma linha em branco no texto e a próxima linha será indentada em i caracteres. Depois disso, é chamada a função DIVIDE_LINHAS passando a linha como parâmetro. Caso não haja comando de parágrafo é feita simplesmente uma chamada à função DIVIDE_LINHAS e passada a linha para ser formatada.

A função FORMAT é chamada recursivamente até que todo o texto tenha sido formatado, lembrando que se houver uma palavra com tamanho superior a l caracteres esta função devolverá o texto formatado até a linha anterior ao erro e coloca no texto de saída uma mensagem de erro precedida por uma linha em branco.

Para facilitar os exemplos, decidimos utilizar um formatador para texto com l=15 (tamanho da linha formatada) e i=6 (indentação).

A função FORMAT definida em HOPE é a seguinte:

```
value l = 15;
value i = 6;
dec FORMAT : (list(list(char)) X list(char)) -> list(list(char));

--- FORMAT(nil,nil) <= nil;
--- FORMAT(nil,r)
    <=(if (tl > l)
       then
           [BRANCO(l)]<>
           (["<erro> palavra com tamanho maior que linha"]<>
           [JOIN(lin)])
       else
           if (length(rest)>0)
           then([JOIN(lin)]<>FORMAT(nil,rest))
           else[JOIN(lin)]
       where (lin,tl,b,rest)== DIVIDE_LINHAS(nil,0,0,r));

--- FORMAT((s:: texto),r)
    <=(if parag
       then
           (if (tp > l)
            then
                [BRANCO(l)]<>
                (["<erro> palavra com tamanho maior que linha"]<>
                [JOIN(lin)])
            else
                ((FORMAT(nil,r)<>([BRANCO(l)]<>
                [(BRANCO(l)<>JOIN(lin))]))<>FORMAT(texto,rest))
       where(lin,tp,b,rest)==DIVIDE_LINHAS(nil,i,0,a))
       else
           (if (tp > l)
```

```
        then
            [BRANCO(l)]<>
            (["<erro> palavra com tamanho maior que linha"]<>
            [JOIN(lin)])
        else
            ([JOIN(lin)]<>FORMAT(texto,rest))
        where(lin,tp,b,rest)==DIVIDE_LINHAS(nil,0,0,((r<>" ")<>a)))
    where(parag,a)==PARAGRAFO(s));
```

A função auxiliar

```
JOIN: list(list(char)) -> list(char)
```

transforma uma lista do tipo list(list(char)) numa lista do tipo list(char), enquanto que a função

```
PARAGRAFO : list(char) -> (truval X list(char))
```

recebe uma linha e devolve a linha depois de analisar a existência do comando de parágrafo. Se existir o comando de parágrafo, então este é retirado da linha (espaço no início da linha) e a função devolve o valor "true", caso contrário a linha de entrada é devolvida juntamente com um valor "false".

A expressão qualificada que utiliza a cláusula "where" usada na função FORMAT é uma forma de simplificar expressões. Assim, é possível utilizar os valores e depois definir o seu significado. Note que não se está atribuindo valores a variáveis através do "==", mas definindo o significado de um nome que será utilizado na expressão.

Veja a seguir o resultado de uma aplicação da função FORMAT:

```
FORMAT([" Teste da Funcao",
        "FORMAT"],nil);
```

O resultado desta aplicação é:

```
["                ",
 "      Teste  da",
 "funcao   FORMAT"] : list(list(char));
```

4.2 FUNÇÃO DIVIDE_LINHAS

Esta função recebe uma linha do texto e separa todas as palavras e espaços entre as palavras. Toda seqüência de espaços na entrada é inicialmente transformada num único espaço. As palavras vão sendo agrupadas enquanto o tamanho da linha não for superior a l caracteres. São também computados os espaços entre as palavras.

Quando o tamanho da linha sendo formatada (t) for tal que a próxima palavra, com tamanho tp, não possa ser concatenada, ou seja, t+tp é maior que l, então a função DIVIDE_LINHAS chama a função FORMLINHA a fim de que a linha seja formatada.

A função DIVIDE_LINHAS é também responsável pelo cálculo do número de espaços que deverão ser inseridos na linha a ser formatada de modo que o seu tamanho se torne igual a l caracteres, computando quantos espaços deverão ser inseridos entre cada palavra da linha. Se o número de espaços a ser inserido não for inteiro, então o resto da divisão é computado para ser, também, passado como entrada da função FORMLINHA.

A função DIVIDE_LINHAS devolve como resultado uma linha formatada e o resto da linha de entrada que excedeu o tamanho de uma linha formatada. Mostramos a seguir a definição desta função em HOPE:

```
dec DIVIDE_LINHAS : (list(list(char)) X num X num X list(char)) ->

                    (list(list(char)) X num X num X  list(char));

--- DIVIDE_LINHAS(lin,t,b,nil)
    <=if ((b>0)
      then (FORMLINHA(lin,(l-t) mod b,(l-t) div b),0,0,nil)
      else (lin,t,b,nil));

--- DIVIDE_LINHAS(lin,t,b,s)
    <= (if (tp > l )
       then (nil,tp,0,p)
       else
           if((tp+t) < l)
           then if(p=" ")
                then DIVIDE_LINHAS((lin<>[p]),(t+1),(b+1),r)
                else DIVIDE_LINHAS((lin<>[p]),(t+tp),b,r)
           else if(p=" ")
                then(FORMLINHA(lin,(l-t) mod b,(l-t) div b),0,0,r)
                else(FORMLINHA(NUM_BRANCOS(lin,t,b,nil)),0,0,(p<>r))

       where(p,tp,r) == FIRSTWORD(s));
```

A função

```
FIRSTWORD : list(char) -> (list(char)) X num X list(char))
```

devolve a primeira palavra de uma lista e o seu tamanho, ou um espaço. Se o primeiro caracter da lista for um espaço, a função verifica a existência de outros espaços sendo todos eles retirados da lista.

Será ilustrada a seguir a aplicação da função DIVIDE_LINHAS quando recebe a primeira linha do texto exemplo aplicado anteriormente à função FORMAT, ou seja :

```
DIVIDE_LINHAS (nil,6,0,"Teste da funcao");
```

A função devolve os seguintes valores:

```
(["Teste","  ","da"],0,0,"funcao") : list(list(char)) X num X num
                                      X list(char);
```

4.3 FUNÇAO FORMLINHA

Esta função é responsável pela distribuição dos espaços na linha, de modo a evitar a existência de "buracos" no final da linha. Ela recebe como entrada uma lista contendo as palavras e espaços da linha a ser formatada, o número de espaços que deverão ser inseridos entre as palavras e um número de espaços que deverão ser distribuídos um a um da esquerda para a direita entre as palavras da linha. A saída desta função é uma linha formatada sem "buracos", caso seja possível. Se não existir espaços entre palavras na linha, então não será possível evitar "buracos" no final da linha. Veja a seguir a definição desta função em HOPE:

```
dec FORMLINHA : (list(list(char)) X num X num ) -> list(list(char));

--- FORMLINHA (c::s,n,0)
    <= if (n=0)
       then (c::s)
       else if (c= " ")
            then (" "::FORMLINHA(s,(n-1),0))
            else c::FORMLINHA(s,n,0) :

--- FORMLINHA (c::s,0,n)
    <= if (c= " ")
       then (BRANCO(n+1)::FORMLINHA(s,0,n))
       else (c::FORMLINHA(s,0,n));

--- FORMLINHA(nil,n,m) <= nil;

--- FORMLINHA(c::s,dif,n)
    <= if (c=" ")
       then(BRANCO(n+2)::FORMLINHA(s,(dif-1),n))
       else(c::FORMLINHA(s,dif,n) );
```

A função auxiliar

```
BRANCO : num -> list(char),
```

simplesmente, devolve uma lista de espaços de acordo com o número de entrada. Veja abaixo uma aplicação da função FORMLINHA:

```
FORMLINHA (["Teste"," ","da"],0,1);
```

que resulta em:

```
["teste","  ","da"] : list(list(char));
```

5. CONCLUSOES

A principal contribuição que a especificação em estilo funcional executável oferece é a possibilidade da validação através da execução. Deste modo, tem-se um método para verificar as propriedades da especificação de uma forma prática, sem por outro lado, perder a precisão da especificação. Isto permite encontrar erros de especificação de forma rápida e simples.

Uma desvantagem importante é a adaptação de certas funções aos moldes da linguagem de especificação o que impede a criação de notações que ofereceriam um maior poder de expressão. Mas este problema pode ser amenizado através da escolha de linguagens que ofereçam alto poder de expressão. Inegavelmente, as linguagens funcionais oferecem um nível de abstração bem mais elevado que as linguagens imperativas, mas certamente inferior a formalismos como Z[9]. É interessante notar como as linguagens funcionais se colocam naturalmente entre métodos como Z e linguagens imperativas.

Outro ponto importante na utilização da linguagem funcional é possibilitar o desenvolvimento modular da especificação, permitindo que os módulos, no caso funções, possam ser validados separadamente.

Finalmente, notamos que esta metodologia de especificação apresenta as principais vantagens da especificação formal, ao mesmo tempo que a validação simples ajuda a diminuir os custos de desenvolvimento de software, pois diminui o possível número de "bugs" em fases mais avançadas do projeto.

REFERENCIAS

[1] Bailey, R., "A HOPE Tutorial". Imperial College, London, UK.

[2] Bailey, R., "Using IC-HOPE under MS-DOS". Imperial College, London, UK.

[2] Bailey, R., "Using IC-HOPE under MS-DOS". Imperial College, London, UK.

[3] Henderson,Peter, "Functional Programming, Formal Specification and Rapid Prototyping". Transactions on Software Engineering, Vol 12. No.2, Fev/86.

[4] J. Stoye et Al., "The SKIM Microprogramming's Guide". Tech. Rep. 40, Computer Lab., University of Cambridge , 1984.

[5] H. Richard, "An Overview of the Burroughs NORMA". Burroughs Corp. ARC, Austin, Tx, 1985

[6] Meira, S.L., "Programação Funcional". V JAI , Recife, PE , Jul/86.

[7] Turner, D., "MIRANDA: A Non-strict Functional Language with Polymorphic Types". Springer Lecture Notes in Computer Science, Vol 201, 1985.

[8] Wong,E. and Samson,W.B., "The Specification of a Relational Database (PRDC1) as an Abstract Data Type and its Realization in HOPE". The Computer Journal, Vol.29. No.3, 1986.

[9] A.C.Sampaio, C.H.Salazar, L.L.Sombra, R.S.Maior & S.R.L.Meira, "Z: Notação Computacional e Especificações". RT-DI/UFPE-007/87.