Lógica Computacional 117366
Descrição do Projeto
Formalização de Algoritmos para Ordenação com Heaps

24 de setembro de 2015

Prof. Mauricio Ayala-Rincón
Prof. Flávio L. C. de Moura

A estagiária de docência Ariane Alves Almeida (`arianealvesalmeida@gmail.com`) dará suporte aos alunos no desenvolvimento do projeto. Laboratórios do LINF têm instalado o *software* necessário (PVS 6.0 com as bibliotecas PVS da NASA).

1 Introdução

Algoritmos de busca e ordenação são fundamentais em Ciência da Computação. Busca é um mecanismo essencial em estruturas de dados e ordenação é relevante para diminuir o tempo de busca em diversas estruturas de dados. Neste projeto considerar-se-ão algoritmos de ordenção sobre o tipo abstrato de dados `finite_sequences` como especificado no assistente de demonstração PVS.

O objetivo do projeto da disciplina é introduzir os mecanismos básicos de manuseio de tecnologias de verificação e formalização que utilizam técnicas dedutivas lógicas, como as estudadas na disciplina, para garantir que objetos computacionais são logicamente corretos.

2 Descrição do Projeto

Este projeto aborda questões apresentadas now arquivos de especificação e prova do algoritmo de ordenação por heaps, que são, respectivamente, `heapsort.pvs` e `heapsort.prf`. Os arquivos estão disponível na página da disciplina, especificados na linguagem do assistente de demonstração PVS (`pvs.csl.sri.com`) executável em plataformas Unix/Linux. Os alunos deverão formalizar propriedades da especificação para ordenação por `heap` sobre a estrutura de dados de sequências finitas de naturais.

2.1 Ordenação por heap: *Heapsort*

Algumas funções utilizadas na especificação deste algoritmo estão especificadas para listas de naturais no arquivo `sorting.pvs` (arquivo de provas `sorting.prf`), como a noção de `occurrence`, que especifica quantas vezes determinado natural está presente em uma lista, dada por:

```
occurrence(l)(x): RECURSIVE nat =
IF  null?(l) THEN 0
ELSIF
car(l) = x THEN 1 + occurrence(cdr(l))(x)
ELSE
occurrence(cdr(l))(x)
ENDIF
MEASURE length(l)
```

Também temos para sequências finitas uma função semelhante, dada por:

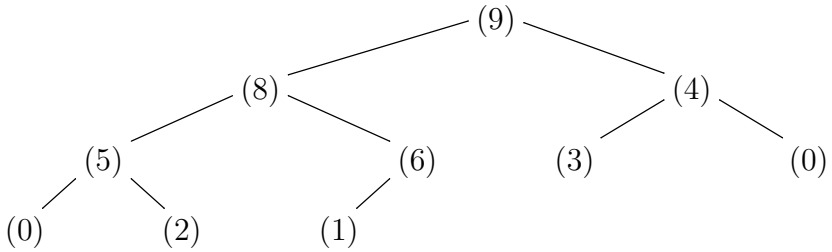
```
occurrences(h)(x:nat): RECURSIVE nat =
IF  length(h) = 0 THEN 0
ELSIF
h(0) = x THEN 1 + occurrences(h^(1,length(h)-1))(x)
ELSE
occurrences(h^(1,length(h)-1))(x)
ENDIF
MEASURE length(h)
```

A primeira questão do projeto está relacionada com a correspondência entre estas especificações de occurrências en listas e sequências finitas.

O objetivo é formalizar que a a especificação de ordenação por heap, abaixo, é correta. Ou seja, provar que ela ordena a entrada preservando o número de ocorrências dos elementos na entrada (veja questão 3).

```
heapsort(h) : finite_sequence[nat] =
IF length(h) > 1 THEN
heapsort_aux(heapify(h)(floor(length(h)/2) - 1))(length(h)-1)
ELSE h
ENDIF
```

A função `heapsort` inicialmente transforma a sequência dada em um *heap*, usando a função `heapify`. Um *heap* é uma árvore binária balanceada, mas não necessariamente completa, i.e., os irmos do último nó interno tem exatamente dois filhos, e este pode ter um único filho esquerdo. Adicionalmente, um *heap* satisfaz a seguinte relação de ordem: a chave de cada nó interno é maior ou igual que as chaves dos seus filhos ([CLRS01, BvG99]). Por exemplo, a árvore:



É um *heap* e pode ser representado como a sequência finita:

9	8	4	5	6	3	0	0	2	1

A função `heapify` deve transformar sequências finitas em um heap, e está especificada como:

```
heapify(h)(i : nat | i <= floor(length(h)/2) - 1) :
RECURSIVE finite_sequence[nat] =
IF i > 0 THEN
heapify(sink(h)(i, length(h) - 1))(i - 1)
ELSE
sink(h)(i, length(h) -1)
ENDIF
MEASURE i
```

Logo `heapsort` aplica a função recursiva `heapsort_aux`, abaixo, que se encarrega de todo o processo de ordenação.

```
heapsort_aux(h)(n :below[length(h)]): RECURSIVE finite_sequence[nat] =
IF n = 0 THEN h
ELSIF n = 1 THEN swap(h)(0,1)
ELSE heapsort_aux(sink(swap(h)(0,n))(0, n - 1))(n - 1)
ENDIF
MEASURE n
```

`heapsort_aux` aproveita a ordenação do `heap`, que implica que a raiz da estrutura contém a maior chave, colocando-o então na última posição, por meio da função `swap`.

```
swap(h)(i, j : below[length(h)]) :  finite_sequence[nat] =
(# length := length(h) ,
seq := (LAMBDA (k : below[length(h)]) :
IF k = i THEN h(j)
ELSIF k = j THEN h(i)
ELSE h(k) ENDIF)  #)
```

Depois disto, `heapsort` reorganiza o `heap` por meio da função `sink` até a posição anterior àquela da troca do maior elemento. O processo se realiza recursivamente:

```
sink(h)((i : below[length(h)]), (n : below[length(h)] | n >= i)) :
RECURSIVE finite_sequence[nat] =
IF i > floor((n+1)/2) - 1 THEN h
ELSE LET k = ind_gc(h)(n,i) IN
IF h(k) > h(i) THEN sink(swap(h)(i,k))(k,n)
ELSE h ENDIF
ENDIF
MEASURE n - i
```

3 Questões

Deverão ser formalizados resultados a seguir, conforme a especificação `heapsort.pvs`.
Questão 01 *As especificações de ocorrência para sequências finitas e listas coincidem.*

```
same_occ_fseq_list  : CONJECTURE
FORALL(h, (x :nat)) : occurrences(h)(x) = occurrence(finseq2list(h))(x)
```

Como ambas definições são recursivas, a prova deve ser feita por indução no tamanho da sequência.

A informação em sequências se mantém se estas tem exatamente o mesmo n'mero de ocorrências para qualquer natural. Isso da origem à noção de `permutations`. As ações sobre sequências finitas, no processo de ordenação, devem preservar a informação. Assim, por exemplo é necessário formalizar que operações como `swap` geram permutações.

Finalmente, temos uma questão desafio e uma principal sobre a correção de `heapsort`.
Questão 02 - desafio *Com as invariantes necessárias o algoritmo* `heapsort_aux` *recursivamente ordena corretamente sequências finitas com a ordenação de* heap.

```
heapsort_aux_psorts : CONJECTURE
 FORALL (h, (n : nat | n<length(h)-1)) :
         (psorted(h)(n+1,length(h)-1) AND
         is_p_heap(h)(0,n) AND
         FORALL (i : below[n+1]) : h(i) <= h(n+1) )
 IMPLIES psorted(heapsort_aux(h)(n))(0,length(h) - 1)
```

A formalização desta conjectura realiza-se por indução em `n`. Mas deve ser notado que `n` depende do comprimento de `h`. No passo indutivo, é necessário aplicar resultados sobre as funções `swap` e `sink` que são as aplicadas alternadamente em cada passo recursivo do `heapsort_aux`. Mas também é necessário uma consideração muito precisa das mudanças dos índices das sequências finitas.

Questão 03

Quando aplicamos a função `heapsort` *a uma sequência, o resultado deve ser uma sequência ordenada que preserva a informação da entrada.*

```
heapsort_works : THEOREM
FORALL (h) : sorted(heapsort(h)) AND permutations(h, heapsort(h))
```

Esta questão não requer uso de indução, porém necessita total entendimento da especificação e dos resultados auxiliares formalizados fornecidos.

4 Etapas do desenvolvimento do projeto

Os alunos deverão definir os grupos de trabalho limitados a **três** membros até o dia 28 de Setembro. Exceto pelo dia da segunda prova, 2 de Dezembro de 2015, as aulas serão realizadas no LINF a partir do dia 23 de Setembro.

O projeto será dividido em duas etapas como segue:

- A primeira etapa do projeto e a de Verificação das Formalizações. Os grupos deverão ter prontas as suas formalizações na linguagem do assistente de demonstração PVS e enviar via e-mail à estagiária com cópia para o professor os arquivos de especificação e de provas desenvolvidos (`heapsort.pvs` e `heapsort.prf`) até o dia **09.11.2015**. Na mesma semana dias **9 e 11.11.2015**, durante o horário de aula, realizar-se-á a verificação do trabalho para a qual os grupos deverão, em acordo com o monitor e professor, determinar um horário (de uma hora) no qual todos membros do grupo deverão comparecer.
  Avaliação (peso 6.0):
  - Um dos membros, selecionado por sorteio, explicará os detalhes da formalização em, no máximo, 20 minutos.
  - Os quatro membros do grupo poderão complementar a explicação inicial em, no máximo, 10 minutos.
  - A formalização será testada nos 30 minutos seguintes.
- A segunda etapa do projeto consiste da apresentação dos resultados finais e conclusões do estudo do problema.
  Avaliação (peso 4.0): Cada grupo de trabalho devera entregar um Relatório Final inédito, editado em LaTeX, limitado a oito páginas (12 pts, A4, espaçamento simples) do projeto até o dia **18.11.2015** com o seguinte conteúdo:

- Introdução e contextualização do problema.
- Explicação da soluções.
- Especificação do problema e explicação do método de solução.
- Descrição da formalização.
- Conclusões.
- Referências.

Referências

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms — Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, second edition, 2001.