Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação 20 de agosto de 2020

Projeto 1

Projeto e Análise de Algoritmos

Prof. Flávio L. C. de Moura*

20 de agosto de 2020

Introdução

A construção de programas corretos e eficientes é um ponto central na Ciência da Computação. Mas como garantir que um programa está correto? A correção de um programa é estabelecida via uma série de propriedades que o programa deve satisfazer.

As provas em papel e lápis normalmente são suficientes para a análise de programas simples, mas podem esconder erros no caso de programas mais complexos. De fato, alguns exemplos famosos de erros envolvendo sistemas críticos incluem:

1. **Pentium FDIV**: Um erro na construção da unidade de ponto flutuante do processador Pentium da Intel causou um prejuízo de aproximadamente 500 milhões de dólares para a empresa que se viu forçada a substituir os processadores que já estavam no mercado em 1994.

2. **Therac-25**: Uma máquina de radioterapia controlada por computador causou a morte de pelo menos 6 pacientes entre 1985 e 1987 por overdose de radiação.

3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

Neste contexto, a utilização de métodos formais na construção de programas é cada vez mais comum:

*flavio@flaviomoura.info

1. A Intel e AMD utilizam assistentes de provas na verificação de processadores.

2. A Microsoft utiliza métodos formais na verificação de programas e drivers.

3. CompCert: Compilador C verificado em Coq.

4. A AirBus e a NASA utilizam assistentes de provas na verificação de programas de aviação.

5. A Toyota utiliza métodos formais na verificação de sistemas híbridos de controle.

6. A linha 14 do metrô de Paris é totalmente controlada por um programa de computador verificado formalmente.

Apesar da utilização cada vez mais frequente de métodos formais na construção de programas, esta não é uma tarefa fácil. Intuitivamente, um programa é correto se faz exatamente o que se propõe em tempo e espaço finitos. Por exemplo, um programa P que ordena listas de números naturais em ordem crescente é correto se, para qualquer lista l dada, o resultado retornado por P após um tempo finito é uma lista contendo exatamente os elementos de l ordenados de forma crescente.

Descrição do projeto

A proposta deste projeto é formalizar a correção de uma versão recursiva do algoritmo de ordenação por inserção sobre listas de números naturais no assistente de provas Coq (https://coq.inria.fr).

O algoritmo de ordenação por inserção pode ser definido recursivamente como a seguir:

```
Fixpoint ord_insercao l :=
 match l with
   | nil => nil
   | h :: tl => insere h (ord_insercao tl)
 end.
```

onde a função `insere` é definida por:

```
Fixpoint insere (n:nat) (l: list nat) :=
  match l with
  | nil => n :: nil
  | h :: tl => if n <=? h then (n :: l)
     else (h :: (insere n tl))
end.
```

A correção de `ord_insercao`

Observe que a função `insere` é construída de forma a preservar a ordenação após a inserção. Este comportamento de `insere` pode ser representado por meio do seguinte lema:

```
Lemma insere_preserva_ordem : forall (l:list nat) (n:nat),
ordenada l -> ordenada (insere n l).
```

onde `ordenada` é o predicado que captura o fato de uma lista estar ordenada:

```
Inductive ordenada: list nat -> Prop :=
| lista_vazia: ordenada nil
| lista_unit: forall x, ordenada (x :: nil)
| lista_mult: forall x y l, x <= y -> ordenada (y :: l) ->
  ordenada (x :: y :: l).
```

Parte da prova da correção deste algoritmo consiste em provar que `ord_insercao` gera uma lista ordenada para qualquer lista dada como entrada:

```
Lemma ord_insercao_ordena: forall l, ordenada (ord_insercao l).
```

A segunda parte da prova da correção consiste em provar que para qualquer lista `l`, `ord_insercao(l)` gera como saída uma permutação de `l`. Definimos o predicado `perm l1 l2`, para denotar que `l1` é uma permutação de ~l2~da seguinte forma:

```
Inductive perm: list nat -> list nat -> Prop :=
| perm_refl: forall l, perm l l
| perm_hd: forall x l l', perm l l' -> perm (x::l) (x::l')
| perm_swap: forall x y l l', perm l l' ->
  perm (x::y::l) (y::x::l')
| perm_trans: forall l l' l'', perm l l' ->
  perm l' l'' -> perm l l''.
```

O lema `ord_insercao_perm` afirma que `ord_insercao l` é uma permutação de `l`.

```
Lemma ord_insercao_perm: forall l, perm l (ord_insercao l).
```

O teorema principal deste projeto caracteriza a correção do algoritmo de ordenação por inserção dado pela função `ord_insercao`:

```
Theorem correcao_ord_insercao: forall l,
ordenada (ord_insercao l) /\ perm l (ord_insercao l).
```

Etapas do projeto

O trabalho, que possui duas etapas, poderá ser realizado individualmente ou em duplas. Os grupos deverão ser formados no GitHub a partir do link:

https://classroom.github.com/g/AeUWbQJi

Para trabalhar individualmente, clique no link e depois clique em "Create team"para criar o seu repositório individual. Caso queira formar um dupla, primeiro combine com o colega com quem você quer formar a dupla. Em seguida, APENAS UM dos elementos da dupla cria o repositório clicando em "Create team". Depois disto, o outro elemento da dupla clica no link fornecido acima e a janela mostrará todos os repositórios já criados. Neste momento, clique em "join"para entrar no repositório criado pela sua dupla.

ATENÇÃO: Se você está em um repositório sem mais participantes, então seu trabalho é individual! Trabalhos idênticos em repositórios distintos serão identificados como plágio, resultando em nota 0 (zero) para todos os trabalhos envolvidos, assim como as sanções previstas em lei.

Etapa 1: Formalização do algoritmo (15 pontos)

Esta etapa consiste na construção das provas dos lemas e teoremas apresentados no arquivo `ord_insercao.v`.

Etapa 2: Relatório (10 pontos)

Esta etapa consiste na elaboração de um relatório inédito em formato pdf ou página web, ambos devem estar disponíveis no próprio repositório GitHub.

Referências

[AdM17] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs*. Undergraduate Topics in Computer Science. Springer, 2017.

[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.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume Volume 3 of The Art of Computer Programming. Reading, Massachusetts: Addison-Wesley, 1973. Also, 2nd edition, 1998.

[Lev12] A. Levitin. *Introduction to the Design & Analysis of Algorithms*. Pearson, third edition, 2012.