Lógica Computacional 1
Descrição do Projeto
Formalização do algoritmo de ordenação *bubblesort*
3 de Outubro de 2020
Profs. Mauricio Ayala-Rincón & Flávio L. C. de Moura

1 Introdução

O algoritmo *bubblesort* é um algoritmo de ordenação simples que, repetidamente compara dois elementos adjacentes de uma lista, e os troca de posição se estiverem na posição errada. Por exemplo, ao receber a lista $4 :: 3 :: 2 :: 1 :: nil$ como entrada, este algoritmo inicialmente compara os dois primeiros elementos da lista e os troca de posição gerando a lista $3 :: 4 :: 2 :: 1 :: nil$. Agora o segundo e terceiros elementos são comparados, a saber o 4 e o 2, e a lista resultante é $3 :: 2 :: 4 :: 1 :: nil$. Neste momento o terceiro e quarto elementos são comparados e a lista resultante é $3 :: 2 :: 1 :: 4 :: nil$. Estas comparações de elementos sucessivos faz com que os elementos maiores da lista flutuem (como bolhas) para o final da lista, enquanto que os elementos menores são deslocados para o início da lista. Esta é a ideia do algoritmo *bubblesort*. No exemplo anterior, o processo de comparação de elementos sucessivos é repetido até que a lista fique ordenada. A partir do ponto em que paramos teríamos os seguintes passos, onde os elementos em negrito estão sendo comparados:

- $\mathbf{3} :: \mathbf{2} :: 1 :: 4 :: nil$.
- $2 :: \mathbf{3} :: \mathbf{1} :: 4 :: nil$.
- $2 :: 1 :: \mathbf{3} :: \mathbf{4} :: nil$.
- $2 :: 1 :: 3 :: 4 :: nil$.

Mais uma rodada de comparações é necessária para que finalmente a lista fique ordenada:

- $\mathbf{2} :: \mathbf{1} :: 3 :: 4 :: nil$.
- $1 :: \mathbf{2} :: \mathbf{3} :: 4 :: nil$.
- $1 :: 2 :: \mathbf{3} :: \mathbf{4} :: nil$.
- $1 :: 2 :: 3 :: 4 :: nil$.

Este algoritmo, apesar de ineficiente, nos permitirá explorar as etapas para demonstrar formalmente a correção de um algoritmo de ordenação. Para uma descrição mais detalhada do *bubblesort*, consulte as referências no final deste arquivo.

2 Descrição do Projeto

A prova da correção de um algoritmo de ordenação consiste de duas etapas. Inicialmente, provaremos que o algoritmo efetivamente ordena os elementos da lista dada como argumento, e em seguida precisamos mostrar que a lista de saída é uma permutação da lista dada como entrada.

2.1 Parte 1

Inicialmente, definimos indutivamente o predicado *sorted*, que nos permite provar se uma lista dada como argumento está ordenada:

```
Inductive sorted :list nat → Prop :=
  | nil_sorted : sorted nil
  | one_sorted: ∀ n:nat, sorted (n :: nil)
  | all_sorted : ∀ (x y: nat) (l:list nat), sorted (y :: l) → x ≤ y → sorted (x :: y :: l).
```

O predicado *sorted* possui três construtores, a saber *nil_sorted*, *one_sorted* and *all_sorted*. Os dois primeiros construtores são axiomas que afirmam que a lista vazia e que listas unitárias estão ordenadas:

$$\frac{}{sorted\ nil}\ (nil\_sorted) \qquad \frac{}{\forall n, sorted(n :: nil)}\ (one\_sorted)$$

O terceiro construtor, i.e. *all_sorted* estabelece as condições para que uma lista com pelo menos dois elementos esteja ordenada. Assim, quaisquer que sejam os elementos x e y, e a lista l, temos:

$$\frac{sorted(y :: l) \qquad x \leq y}{sorted(x :: y :: l)}\ (all\_sorted)$$

Ou seja, para provarmos que a lista $x :: y :: l$ está ordenada, precisamos provar que $x \leq y$ e que a lista $y :: l$ também está ordenada.

No lema a seguir, mostramos como provar que se uma lista não vazia está ordenada, então sua cauda também está ordenada. Observe que l é uma sublista de $a :: l$ e compare este fato com a questão 1.

Lemma *tail_sorted*: ∀ *l a*, *sorted* (*a* :: *l*) → *sorted l*.
Proof.

`intro l.`	Este comando introduz uma constante no contexto da prova, e é sempre utilizado quando temos quantificação universal ou implicação no consequente. Este processo é conhecido como skolemização. Na prática de provas em Matemática, isto corresponde a dizer "seja l uma lista qualquer".
`case l.`	Este comando faz uma análise de casos sobre l. Temos então dois casos a considerar:
`- intros a H.`	No primeiro subcaso, a lista l é a lista vazia. Sejam a um natural e H a hipótese de que a lista $a :: nil$ está ordenada.
`apply nil_sorted.`	Precisamos provar que a lista vazia está ordenada. Para isto basta aplicarmos o axioma *nil_sorted*.
`- intros n l' a H.`	No caso indutivo, sejam n um natural, l' uma lista, a um natural e H a hipótese de que a lista $a :: n :: l'$ está ordenada. Precisamos provar que a lista $n :: l'$ está ordenada.
`inversion H; subst.`	Observe que o fato de $sorted(a :: n :: l')$ (hipótese H) significa que $a \leq n$ e $sorted(n :: l')$ de acordo com a regra all_sorted. O comando `inversion` gera as condições que permitem construir uma hipótese, e é normalmente combinada com a tática *subst* que faz substituições e elimina as igualdades geradas pelo comando *inversion*.

`assumption.` Observe que uma das hipóteses geradas pelo comando anterior é exatamente o que queremos provar. O comando `assumption` verifica que o objetivo atual corresponde a uma das hipóteses.

`Qed.`

2.2 Questão 1:

A primeira questão deste projeto consiste em provar que se temos uma lista com pelo menos dois elementos $a1 :: a2 :: l$ e removemos o segundo elemento, a lista obtida $a1 :: l$ (que não é uma sublista da lista original!) também está ordenada. Se você precisar utilizar a transitividade de $\leq$, use o lema *Nat.le_trans*.

```
Lemma remove_sorted: ∀ l a1 a2, sorted (a1 :: a2 :: l) → sorted (a1 :: l).
Proof.
  Admitted.
```

O algoritmo *bubblesort* é baseado na função *bubble* que percorre a lista dada como argumento comparando seus elementos consecutivos:

```
Function bubble (l: list nat) {measure length} :=
  match l with
  | h1 :: h2 :: tl ⇒
    if h1 ≤? h2
    then h1 :: (bubble (h2 :: tl))
    else h2 :: (bubble (h1 :: tl))
    | _ ⇒ l
  end.
```

Mais precisamente, a função *bubble* recebe uma lista l de números naturais como argumento, e é definida recursivamente. A recursão é baseada no tamanho da lista, e daí a necessidade do parâmetro `measure length`. Observe que se a lista dada como entrada possui dois ou mais elementos então os dois primeiros elementos são comparados, e se necessário, suas posições são trocadas. O processo continua com a comparação do segundo e terceiro elementos, e assim até que o penúltimo e o último elementos sejam comparados. Quando a lista é vazia ou unitária, a função *bubble* não faz nada. Por exemplo, aplicando a função *bubble* à lista $4 :: 3 :: 2 :: 1 :: nil$ obtemos como resultado a lista $3 :: 2 :: 1 :: 4 :: nil$ porque inicialmente 4 é comparado com 3, e neste caso o 3 é movido para fora da recursão e o processo continua como $3 ::$(*bubble*“ $4 :: 2 :: 1 :: nil$). No passo seguinte, o 4 é comparado com o 2 e temos $3 :: 2 ::$(*bubble*“ $4 :: 1 :: nil$), e finalmente, 4 é comparado com o 1 e obtemos a lista $3 :: 2 :: 1 :: 4 :: nil$.

2.3 Questão 2:

A segunda questão a ser resolvida neste projeto consiste em provar que a função *bubble* não faz nada em listas ordenadas:

```
Lemma bubble_sorted: ∀ l, sorted l → bubble l = l.
Proof.
   Admitted.
```

O algoritmo *bubblesort* é definido recursivamente como abaixo. A palavra reservada `Fixpoint` é utilizada para definir funções recursivas (simples) enquanto que `Function` usada na definição de *bubble* é utilizada para definir funções recursivas mais sofisticadas, cuja métrica que garante a sua boa definição precisa ser fornecida explicitamente:

```
Fixpoint bubblesort (l: list nat) :=
  match l with
  | nil => nil
  | h :: tl => bubble (h :: bubblesort tl)
  end.
```

O predicado *le_all*, definido a seguir, recebe um natural n e uma lista l como argumentos, e a fórmula *le_all* n l possui uma prova quando n é menor ou igual a todos os elementos da lista l. Escreveremos $n \leq * \ l$ ao invés de *le_all* n l.

```
Definition le_all x l := Forall (fun y => x <= y) l.
```

O predicado *Forall* acima é definido indutivamente pelas seguintes regras:

$$\frac{}{Forall\ P\ nil}\ (Forall\_nil)$$

$$\frac{P\ x \qquad Forall\ P\ l}{Forall\ P\ (x :: l)}\ (Forall\_cons)$$

Assim, dada uma propriedade P sobre elementos de um dado tipo A, $Forall\ P\ l$ consiste em uma prova de que todos os elementos de l satisfazem a propriedade P. De fato, a regra $Forall\_nil$ consiste no axioma que diz que, por vacuidade, todos os elementos da lista vazia satisfazem a propriedade P. Já a regra $Forall\_cons$ fornece uma prova de que a lista $x :: l$ satisfaz a propriedade P a partir das provas de que x satisfaz P, e de que todos os elementos de l também satisfazem P.

A seguir provaremos duas propriedades envolvendo o predicado *le_all*. Nosso primeiro exemplo, consiste em mostrar que se a lista $a :: l$ está ordenada então a é menor ou igual do que qualquer elemento de l.

```
Lemma sorted_le_all: ∀ l a, sorted(a :: l) → a <=* l.
Proof.
  induction l.
```

Esta prova é feita por indução na estrutura da lista l. Teremos então dois casos a considerar: o caso em que l é a lista vazia, e o caso em que l não é vazia.

```
  - intros a H.
```

A base de indução consiste no caso em que a lista l é a lista vazia. Sejam então a um número natural, e H a hipótese de que a lista $a :: nil$ está ordenada. Precisamos provar que o natural a é menor ou igual a todos os elementos da lista vazia.

```
    apply Forall_nil.
```

Mas como comentado anteriormente, este caso consiste na aplicação do axioma $Forall\_nil$.

```
  - intros a' H.
```

No caso em que a lista não é vazia, digamos $a :: l$, sejam a' um número natural e H a hipótese de que a lista $a' :: a :: l$ está ordenada. Precisamos provar que a' é menor ou igual a todos os elementos de $a :: l$.

`inversion` H; `subst`.

Como $a' :: a :: l$ está ordenada, então pela definição de *sorted* temos que $a' \leq a$ e que $a :: l$ está ordenada. A tática *inversion* deriva para cada construtor possível de *sorted* $(a' :: a :: l)$ as condições necessárias para a sua prova. Neste caso, o único construtor possível é *all_sorted* que nos dá como hipóteses que $a' \leq a$ e que a lista $a :: l$ está ordenada.

`apply` *Forall_cons*.

\+ `assumption`.

Inicialmente precisamos mostrar que $a' \leq a$, mas esta foi uma das hipóteses geradas por *inversion H*.

\+ `apply` *IHl*.

Agora no passo indutivo, temos por hipótese de indução que, para qualquer natural a, se a lista $a :: l$ está ordenada então $a <= *l$. Podemos aplicar a hipótese de indução instanciando a com a' e então temos que provar que a lista $a' :: l$ está ordenada.

`apply` *remove_sorted* `in` H; `assumption`.

Como a hipótese H nos diz que a lista $a' :: a :: l$ está ordenada, podemos concluir esta prova usando o lema provado na questão 1.

`Qed`.

2.4 Questão 3

Agora sejam a um natural e l uma lista ordenada. Prove que se a é menor ou igual do que todos os elementos de l então a lista $a :: l$ está ordenada.

`Lemma` *le_all_sorted*: $\forall\ l\ a,\ a \leq *\ l \rightarrow sorted\ l \rightarrow sorted\ (a :: l)$.
`Proof`.
Admitted.

A seguir provaremos que se o natural a é menor ou igual do que todos os elementos da lista l então a é menor ou igual do que todos os elementos da lista *bubble* l.

`Lemma` *le_all_bubble*: $\forall\ l\ a,\ a \leq *\ l \rightarrow a \leq *\ bubble\ l$.
`Proof`.

`intros` $l\ a\ H$.

Sejam l uma lista de naturais, a um natural e H a hipótese de que a é menor ou igual a todos os elementos de l.

functional `induction` (*bubble* l).

É natural tentarmos iniciar esta prova fazendo indução na estrutura de l, mas nossa hipótese de indução não será expressiva suficiente porque a função [bubble] não é definida sobre a estrutura de l, mas sobre o comprimento de l. O princípio de indução baseado no comprimento de l é obtido via o comando *functional induction* (*bubble* l), e temos 3 casos a considerar. Suponha que l tem a forma $h1 :: h2 :: tl$:

\- `inversion` H; `subst`.

Quando $h1 \leq h2$, precisamos provar que $a \leq *h1 :: bubble(h2 :: tl)$. Como l tem a forma $h1 :: h2 :: tl$, a hípotese H nos diz que $a \leq *h1 :: h2 :: tl$ de onde obtemos que $a \leq h1$ e que $a \leq (h2 :: tl)$ via o comando *inversion H*.

`apply` *Forall_cons*.

Neste passo dividimos a prova de $a \leq *h1 :: bubble(h2 :: tl)$ em duas subprovas de acordo com o construtor $Forall_cons$ como visto anteriormente:

+ `assumption`.

+ `apply` *IHl0*; `assumption`. | Agora precisamos provar que $a \leq *bubble(h2 :: tl)$ que pode ser provado pela hipótese de indução desde que $a \leq (h2 :: tl)$, mas este fato é uma das nossas hipóteses.

- `inversion` *H*; `subst`. | Quando $h2 < h1$, precisamos provar que $a \leq *h2 :: bubble(h1 :: tl)$. Da hipótese H obtemos novamente que $a \leq h1$ e $a \leq (h2 :: tl)$.

`apply` *Forall_cons*. | A prova de $a \leq *h2 :: bubble(h1 :: tl)$ pode ser dividida em duas subprovas de acordo com o construtor $Forall_cons$:

+ `inversion` *H3*; `subst`; `assumption`. | A primeira subprova consiste em mostrar que $a \leq h2$ e pode ser obtida a partir da inversão da hipótese $a \leq (h2 :: tl)$.

+ `apply` *IHl0*. | A segunda subprova consiste em mostrar que $a \leq *bubble(h1 :: tl)$. Para isto utilizamos a hipótese de indução.

`inversion` *H3*; `subst`. | Ao aplicarmos a hipótese de indução reduzimos nosso problema a mostrar que $a \leq *h1 :: tl$.

`apply` *Forall_cons*; `assumption`. | A prova de $a \leq *h1 :: tl$ é dividida nas provas de que $a \leq h1$ e $a \leq *tl$ que são hipóteses obtidas de duas inversões anteriores.

- `assumption`. | Por fim, o terceiro caso da definição da função *bubble* retorna a própria lista l, e portanto este é um caso trivial.

`Qed`.

2.5 Questão 4

Mostre que se l é uma lista ordenada então a lista *bubble* $(a :: l)$ também está ordenada, qualquer que seja o natural a. Alguns resultados, além dos já provados anteriormente podem ser úteis nesta prova como, por exemplo *Nat.leb_le*, *Nat.leb_nle*, *Nat.nle_gt* e *Nat.lt_le_incl*.

`Lemma` *bubble_sorted_sorted*: $\forall\ l\ a,\ sorted\ l \rightarrow sorted\ (bubble\ (a :: l))$.
`Proof`.
  Admitted.

Neste momento podemos provar que *bubblesort* l retorna uma lista ordenada, qualquer que seja a lista l:

```
Theorem bubblesort_sorts: ∀ l, sorted (bubblesort l).
Proof.
  induction l.
  - simpl.
    apply nil_sorted.
  - simpl.
    apply bubble_sorted_sorted.
    assumption.
Qed.
```

2.6 Parte 2

A segunda parte da prova da correção do algoritmo *bubblesort* consiste em mostrar que a lista de saída é uma permutação da lista de entrada.

A permutação de listas é definida indutivamente como a seguir:

```
Inductive perm: list nat → list nat → Prop :=
| perm_refl: ∀ l, perm l l
| perm_hd: ∀ x l l', perm l l' → perm (x::l) (x::l')
| perm_swap: ∀ x y l l', perm l l' → perm (x::y::l) (y::x::l')
| perm_trans: ∀ l1 l2 l3, perm l1 l2 → perm l2 l3 → perm l1 l3.
```

Nesta definição, o construtor *perm_refl* corresponde ao axioma que estabelece que uma lista é permutação dela mesma:

$$\frac{}{perm\ l\ l}\ (perm\_refl)$$

Já o construtor *perm_hd* estabelece que, se a lista l é uma permutação da lista l' então a lista com cabeça x e cauda l é uma permutação da lista que tem cabeça x e cauda l':

$$\frac{perm\ l\ l'}{perm\ (x :: l)\ (x :: l')}\ (perm\_hd)$$

O construtor *perm_swap* nos permite provar que listas que tenham os dois primeiros elementos permutados sejam permutações uma da outra desde que as sublistas correspondentes a partir do terceiro elemento sejam permutação uma da outra:

$$\frac{perm\ l\ l'}{perm\ (x :: y :: l)\ (y :: x :: l')}\ (perm\_swap)$$

E por fim, o construtor *perm_trans* estabelece que a permutação de listas é transitiva.

$$\frac{perm\ l1\ l2 \qquad perm\ l2\ l3}{perm\ l1\ l3}\ (perm\_trans)$$

2.7 Questão 5

Prove que a função *bubble* gera uma permutação da lista de entrada:

```
Lemma bubble_is_perm: ∀ l, perm (bubble l) l.
Proof.
  Admitted.
```

2.8 Questão 6

Mostre que se a lista l é uma permutação da lista l' então *bubble* $(a :: l)$ é uma permutação de $(a :: l')$.

```
Lemma bubble_is_perm': ∀ l l' a, perm l l' → perm (bubble (a::l)) (a :: l').
Proof.
   Admitted.
```

Agora podemos concluir a segunda parte da formalização com a prova de que *bubblesort* gera uma permutação da lista de entrada.

```
Theorem bubblesort_is_perm: ∀ l, perm (bubblesort l) l.
Proof.
  induction l.
  - simpl.
    apply perm_refl.
  - simpl.
    apply bubble_is_perm'.
    assumption.
Qed.
```

O resultado principal, que caracteriza a correção do algoritmo de ordenação *bubblesort*, é dado a seguir:

```
Proposition bubblesort_is_correct: ∀ l, perm (bubblesort l) l ∧ sorted (bubblesort l).
Proof.
  intro l; split.
  - apply bubblesort_is_perm.
  - apply bubblesort_sorts.
Qed.
```

3 Etapas de desenvolvimento do Projeto

Os alunos deverão definir grupos de trabalho limitados a quatro membros até o dia 7 de outubro.

O projeto será dividido em duas etapas como segue:

- **Verificação das Formalizações** (peso 6.0): Os grupos deverão ter prontas todas as provas do arquivo `bubblesort.v` até o dia 27/10.
- **Entrega do Relatório Final** (peso 4.0): Cada grupo de trabalho deverá entregar um relatório inédito, limitado a oito páginas (12 pts, A4, espaçamento simples) do projeto até o dia 03/11 com o seguinte conteúdo:
  1. Introdução e contextualização do problema;
  2. Explicação da soluções;
  3. Especificação do problema e explicação do método de solução;
  4. Descrição da formalização;
  5. Conclusões;
  6. Referências.

Referências

[ARdM17] M. Ayala-Rincón and F.L.C. de Moura. *Applied Logic for Computer Scientists - computational deduction and formal proofs*. UTiCS, Springer, 2017.

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

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, third edition, 2009.