Computação Fiável
Lógica, Programação e Demonstração em Coq
Ficha de exercícios

Simão Melo de Sousa

Avisos

Os exercícios seguintes deverão ser resolvidos tanto teoricamente (com papel, caneta e rigor matemático) como em prática, usando o sistema COQ.

Lembramos que a matemática/lógica subjacente ao o sistema COQ é construtiva. Como tal não são assumidos por defeito algumas propriedades ou factos que o são na matemática clássica como por exemplo a redução ao absurdo (ou axioma da dupla negação). Não se esqueçam assim de proceder às devidas medidas[1] quando o processo de demonstração escolhido o necessitar.

Como o definimos e justificamos nas aulas teóricas, utilizaremos a relação de tipagem (representada pelo símbolo :) como a relação de pertence (tradicionalmente representada por $\in$).

Parte destes exercícios foram retirados da documentação, das bibliotecas standards e das contribuições ao sistema Coq.

O Coq activa, via o comando `Set Implicits Arguments`, a inferência de argumentos e de tipos quando possível. Alguns dos enunciados aqui apresentados fazem uso deste comando.

1 Lógica Proposicional

Nos seguintes execícios, acompanhe a resolução pedida com a apresentação das árvores de prova em Dedução Natural.

Exercício 1 *Admitindo que A, B, C, D, P, Q e R são proposições, demonstre, no sistema Coq, os seguintes enunciados:*

[1] Como carregar os módulos para o raciocínio clássico ou explicitar, pela exibição dum elemento, que um determinado conjunto não é vazio

- $((P \vee Q) \wedge R) \rightarrow ((P \wedge R) \vee (Q \wedge R))$
- $((P \wedge Q) \rightarrow R) \rightarrow ((\neg R \wedge P) \rightarrow \neg Q)$
- $((A \rightarrow C) \wedge (B \rightarrow D)) \rightarrow (A \wedge B) \rightarrow (C \wedge D)$
- $\neg(A \wedge B) \rightarrow (\neg A \vee \neg B)$
- $(\neg A \vee \neg B) \rightarrow \neg(A \wedge B)$
- $\neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$
- $((P \wedge Q) \rightarrow R) \rightarrow ((\neg R \wedge P) \rightarrow \neg Q)$
- $(A \rightarrow \neg A) \rightarrow ((A \rightarrow B) \wedge (A \rightarrow \neg B))$
- $\neg(A \rightarrow \neg A)$
- *(Praeclarum):* $(A \rightarrow C) \wedge (B \rightarrow D) \rightarrow (A \wedge B) \rightarrow (C \wedge D)$
- $((A \wedge B) \vee C) \leftrightarrow (A \vee C) \wedge (B \vee C)$
- $\bot \leftrightarrow (A \wedge \neg A)$
- $A \rightarrow \neg\neg A$
- $\neg\neg A \rightarrow A$
- $(A \rightarrow B \wedge \neg B) \rightarrow \neg A$
- $((((A \rightarrow B) \rightarrow A) \rightarrow A) \rightarrow B) \rightarrow B$
- $(A \vee B) \leftrightarrow (\neg A \rightarrow B)$
- $(\neg A \vee (B \rightarrow C)) \rightarrow (A \wedge B) \rightarrow C$
- $\neg(A \wedge \neg A)$
- $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$
- $(A \rightarrow B) \vee (B \rightarrow A)$

□

2 Lógica de Predicados

Nos seguintes execícios, acompanhe a resolução pedida com a apresentação das árvores de prova em Dedução Natural.

Exercício 2 *Admitindo que A não é o conjunto vazio, que P e Q são predicados unário sobre um conjunto A ($A : Set$ e $P,\ Q : A \to Prop$)) e que R é um predicado binário sobre A ($R : A \to A \to Prop$), demonstre os seguintes enunciados:*

- $(\forall x : A.(P\ x)) \to (\exists x : A.(P\ x))$
- $\neg(\forall x : A.(P\ x)) \leftrightarrow \exists x : A.\neg(P\ x)$
- $(\forall x : A, (Px) \wedge (Qx)) \to (\forall x : A, (Px)) \wedge (\forall x : A, (Qx))$
- $\neg(\exists x : A\ (P\ x) \to \forall x : A.\ (P\ x))$
- $\exists x : A.\ ((P\ x) \to \forall y : A(P\ y))$
- $\exists x\ \exists y : A.\ (R\ x\ y) \leftrightarrow \exists y : A\ \exists x : A.\ (R\ x\ y)$
- $\forall A : Set, \forall R : A \to A \to Prop, (\forall x, y, z : A, (R\ x\ y) \wedge (R\ y\ z) \to (R\ x\ z)) \to (\forall x, y : A, (R\ x\ y) \to (R\ y\ x)) \to (\forall x : A, (\exists y : A, (R\ x\ y)) \to (R\ x\ x))$

□

Exercício 3 (Lógica de Predicados com igualdade)
Demonstre os seguintes enunciados:

- $\forall x : A, \exists y.\ (x = y)$
- $\forall x : A.((P\ x) \leftrightarrow (\exists y : A.\ (x = y \wedge (P\ y))))$
- $\forall f : A \to A, \forall x : A, \forall y : A, x = y \to (f\ x) = (f\ y)$

□

Exercício 4 (Relações e Ordens) *Admitindo as seguintes definições:*

```
1  Section Relacoes
2     Variable U : Set.
3
4     Definition Relation := U → U → Prop.
5
6     Variable R : Relation.
7
8     Definition Reflexive : Prop := ∀ x : U, R x x.
9
```

```
    Definition Transitive : Prop := ∀ x y z : U, R x y → R y z → R x z.

    Definition Symmetric : Prop := ∀ x y : U, R x y → R y x.

    Definition Antisymmetric : Prop :=
      ∀ x y : U, R x y → R y x → x = y :>U.

    Definition contains (R R' : Relation) : Prop :=
      ∀ x y : U, R' x y → R x y.

    Definition same_relation (R R' : Relation) : Prop :=
      contains R R' /\ contains R' R.

    Inductive Preorder : Prop :=
        Definition_of_preorder : Reflexive → Transitive → Preorder.

    Inductive Large_Order : Prop :=
        Definition_of_order :
          Reflexive → Transitive → Antisymmetric → Large_Order.

    Inductive Equivalence : Prop :=
        Definition_of_equivalence :
          Reflexive → Transitive → Symmetric → Equivalence.

Definition Complement (U : Set) (R : Relation U) : Relation U :=
        λ x y : U ⇒ ¬ R x y.

    Definition Symrel (U : Set) (R : Relation U) : Relation U :=
        λ x y : U ⇒ R x y /\ R y x


    Inductive Rstar : Relation U :=
      | Rstar_0 : ∀ x : U, Rstar x x
      | Rstar_n : ∀ x y z : U, R x y → Rstar y z → Rstar x z.

    Inductive Rstar1 : Relation U :=
      | Rstar1_0 : ∀ x : U, Rstar1 x x
      | Rstar1_1 : ∀ x y : U, R x y → Rstar1 x y
      | Rstar1_n : ∀ x y z : U, Rstar1 x y → Rstar1 y z → Rstar1 x z.

    Inductive Rplus : Relation U :=
      | Rplus_0 : ∀ x y : U, R x y → Rplus x y
      | Rplus_n : ∀ x y z : U, R x y → Rplus y z → Rplus x z.

End Relacoes.
```

Demonstre os seguintes enunciados:

- `∀ (U : Set) (R : Relation U), Symmetric U R →Symmetric U (Complement U R).`
- `∀ (U : Set) (R : Relation U), Preorder U R →Equivalence U Symrel.`

- ∀ *(U : Set) (R : Relation U), Large_Order U R →Equivalence U Symrel.*
- ∀ *(U : Set), Equivalence (Relation U) (same_relation U).*
- ∀ *(U : Set) (R : Relation U), Reflexive U (Rstar U R).*
- ∀ *(U : Set) (R : Relation U), contains U (Rplus U R) R.*
- ∀ *(U : Set) (R : Relation U), contains U (Rstar U R) R.*
- ∀ *(U : Set) (R : Relation U), Transitive U (Rstar U R).*
- ∀ *(U : Set) (R : Relation U), Symmetric U R →Symmetric U (Rstar U R).*
- ∀ *(U : Set) (R S : Relation U), contains U (Rstar U S) R →contains U (Rstar U S) (Rstar U R).*
- ∀ *(U : Set) (R S : Relation U), contains U S R →contains U (Rstar U S) (Rstar U R).*

□

3 Predicados e Conjuntos Indutivos

Exercício 5 (Semana)

1. *Defina o tipo dia_da_semana dos dias da semana.*
2. *Qual é o princípio de indução associado a este tipo?*
3. *Defina o predicado indutivo* `fim_de_semana: semana →Prop`
4. *Demonstre* $\forall d : dia\_da\_semana, \{fim\_de\_semana\ d\} + \{fim\_de\_semana\ d\}$
5. *Defina a função dia_de_trabalho* : $dia\_da\_semana \rightarrow bool$ *que devolve true quando o seu parâmetro é um dia de trabalho. Defina igualmente a negação desta função: dia_de_descano.*
6. *Diga que diferenças existem entre a função dia_de_descano e o predicado* fim_de_semana*. Estas diferenças são as mesmas entre um predicado (de tipo Prop) e uma função booleana.*
7. *Demonstre que* $\forall d : dia\_da\_semana, fim\_de\_semana\ d \leftrightarrow dia\_de\_descanod$

□

Exercício 6 *Tendo em conta as seguintes definições:*

```
1
2  Inductive nat : Set :=
3    | O : nat
4    | S : nat → nat.
5
6
7  Fixpoint plus (n m:nat) {struct n} : nat :=
8    match n with
9    | O ⇒ m
10   | S p ⇒ S (plus p m)
11   end.
12
13 Infix "+" := plus : nat_scope.
14
15 Fixpoint mult (n m:nat) {struct n} : nat :=
16   match n with
17   | O ⇒ O
18   | S p ⇒ m + mult p m
19   end.
20
21 Infix "*" := mult : nat_scope.
22
23 Inductive le (n:nat) : nat → Prop :=
24   | le_n : le n n
25   | le_S : ∀ m:nat, le n m → le n (S m).
26
27 Infix "≤" := le : nat_scope.
28
29 Definition lt (n m:nat) := S n ≤ m.
30
31 Infix "<" := lt : nat_scope.
32
33
34 Definition ge (n m:nat) := m ≤ n.
35
36 Infix "≥" := ge : nat_scope.
37
38 Definition gt (n m:nat) := m < n.
39
40 Infix ">" := gt : nat_scope.
41
42
43 Fixpoint plus_acc q n {struct n} : nat :=
44   match n with
45   | O ⇒ q
46   | S p ⇒ plus_acc (S q) p
47   end.
48
49 Definition tail_plus n m := plus_acc m n.
```

1. *Define as funções seguintes (ver ficha OCaml): fibbonacci, n!, C_p^n*

2. *Demonstre as seguintes propriedades:*

a soma e a multiplicação

- ∀ n : nat, ¬(S n) = 0.
- ∀ n m p: nat, (n < (m+p)) →((n=m)\/(n<m)).
- ∀ n, (S n) = n + (S 0).
- ∀ n, 0 + n = n.
- ∀ n m, n + m = m + n.
- ∀ n m, S n + m = n + S m.
- ∀ n m p, n + (m + p) = n + m + p.
- ∀ n m p, n + (m + p) = m + (n + p).
- ∀ n m p, p + n ≤p + m →n ≤m.
- ∀ n m, n ≤n + m.
- ∀ n m, n + m = 0 →n = 0 /\ m = 0.
- ∀ n m p, n * (m + p) = n * m + n * p.
- ∀ m n, m + n = 1 →{m = 0 /\ n = 1} + {m = 1 /\ n = 0}.
- ∀ n m, n + m = tail_plus n m.
- ∀ n m : nat, (n≤m) →{z:nat | n + z = m}.
- ∀ n, n > 0 →∀m:nat, {q : nat | ∃r : nat, m = q * n + r /\ n > r}.

as relações $=$ e $\leq$ sobre $\mathbb{N}$

- *a relação $=$ é uma relação de equivalência;*
- *a relação $\leq$ é uma ordem larga;*
- *a relação $\leq$ é uma ordem total.*

□

Exercício 7 *Tendo em conta as seguintes definições:*

```
1  Set Implicit Arguments.
2  Variable A : Set.
3
4  Inductive list : Set :=
5    | nil : list
6    | cons : A → list → list.
7
8  Infix "::" := cons (at level 60, right associativity) : list_scope.
9
10 Fixpoint app (l m:list) {struct l} : list :=
11   match l with
12   | nil ⇒ m
13   | a :: l1 ⇒ a :: app l1 m
14   end.
15
```

```
Infix "++" := app (right associativity, at level 60) : list_scope.


Fixpoint In (a:A) (l:list) {struct l} : Prop :=
  match l with
  | nil ⇒ False
  | b :: m ⇒ b = a \/ In a m
  end.
```

1. *Defina as funções seguinte:*
   - *o comprimento de lista: length;*
   - *a função tail que retira o primeiro elemento do seu parâmetro lista;*
   - *a função head que devolve o primeiro elemento do seu parâmetro lista;*
   - *a função que devolve o n-ésimo elemento duma lista se este existir: nth ;*
   - *a função que inverte uma lista: rev;*
   - *a função map.*
2. *Demonstre as seguintes afirmações:*
   - `∀ l:list, l = l ++ nil.`
   - `∀ (x y:list) (a:A), nil <> x ++ a :: y.`
   - `∀ (x y:list) (a b:A), x ++ a :: nil = y ++ b :: nil →x = y /\ a = b.`
   - `∀ a:A, ¬In a nil.`
   - `∀ x y:list, rev (x ++ y) = rev y ++ rev x.`
   - `(∀ x y:A, {x = y} + {x <> y}) →∀(a:A) (l:list), {In a l} + {¬In a l}.`

□

Exercício 8 *Considere o tipo das expressões aritméticas e a semântica operacional seguinte:*

```
Inductive ArithExp :Set :=
  Zero : ArithExp
 | Succ : ArithExp → ArithExp
 | Plus : ArithExp → ArithExp → ArithExp.


Inductive RewriteRel : ArithExp → ArithExp →Prop:=
  RewSucc : (e1,e2:ArithExp)
 (RewriteRel e1 e2) → (RewriteRel (Succ e1) (Succ e2))
 | RewPlusZ : (e:ArithExp)(RewriteRel (Plus Zero e) e)
 | RewPlusS : (e1,e2:ArithExp)
 (RewriteRel e1 e2) → (RewriteRel (Plus (Succ e1) e2) (Succ (Plus e1 e2)))
| RewPluscom : (e1,e2:ArithExp) (RewriteRel (Plus e1 e2) (Plus e2 e1)).
```

Define os enunciados Coq e demonstre as seguintes propriedades:

- *A igualdade sobre ArithExp é decidível;*
- *Nenhum cálculo é possível a partir de Zero;*
- *Qualquer cálculo duma expressão aritmética começando por Succ resulta numa expressão aritmética começando por Succ.*

□

Exercício 9 (árvores binárias e árvore n-árias) *Implemente em Coq os tipos e as funções dos exercícios* 19 *e* 22 *da ficha prática OCaml.* □

4 Conjuntos, Funções e Indução bem fundada

Exercício 10 *Demonstre em Coq que* $<$ *é uma ordem bem fundada.* □

Exercício 11 *Considere o programa OCaml seguinte:*

```
let rec partition (e:int) (l:'a list) =
 match l with
   [] → ([],[])
 | d::r →
    let (l1,l2) = partition e r in
      if (d < e)
      then (d::l1, l2)
      else (l1, d::l2)

let rec quicksort  (l: 'a list) =
match l with
  [] → []
| [e] → [e]
| e::r →
     let (l1,l2) = partition e r in
       (quicksort l1)@(e::quicksort l2)
```

A semelhança da definição da função div exposta no documento Tutorial on Recursvive Types in Coq *de E. Giménez, defina a função quicksort por indução bem fundada.* □

5 Exemplos completos

Exercício 12 *Resolva o seguinte puzzle lógico.*

Na rua Aspirina existem 5 casas. Cada casa está pintada com uma cor diferente das outras casas. Os donos das casas tem todos uma nacionalidade diferente da dos vizinhos. Os seus

nomes são igualmente diferentes, tais como as suas bebidas preferidas. Em cada casa vive um animal (mais uma vez diferente do animal de cada vizinho).

- *O inglês vive na casa vermelha.*
- *O sueco tem um cão.*
- *Quem vive na casa verde bebe café.*
- *A casa do João é vizinha da casa onde vive um gato.*
- *O dinamarquês bebe chá.*
- *A casa verde está a direita da casa branca.*
- *Pedro tem um pássaro.*
- *Paulo mora na casa amarela.*
- *Quem bebe leite mora na casa do meio.*
- *O norueguês vive na primeira casa a esquerda.*
- *Paulo é vizinho da casa onde está um cavalo.*
- *Tiago bebe cerveja.*
- *O alemão chama-se Hans.*
- *O norueguês vive ao lado da casa azul.*
- *A pessoa que bebe água é vizinha de João.*

Quem tem uma zebra? Demonstre a sua resposta usando o sistema Coq. □

Exercício 13 *Dispomos de três piões brancos e de três piões pretos dispostos em linha e separados por um espaço (a seguir chamamos a esta disposição estado inicial).*

Desejamos descobrir a sequência de operações a realizar para alcançar o estado seguinte (chamado estado final)

sabendo que dispomos das quatros operações seguinte:

Operação 1: *deslocação para a esquerda dum pião preto*

Operação 2: *deslocação para a direita dum pião branco*

Operação 3: *salto para a esquerda dum pião preto por cima dum pião branco*

Operação 4: *salto para a direita dum pião branco por cima dum pião preto*

Uma representação gráfica destas operações é dada pela figura 1:

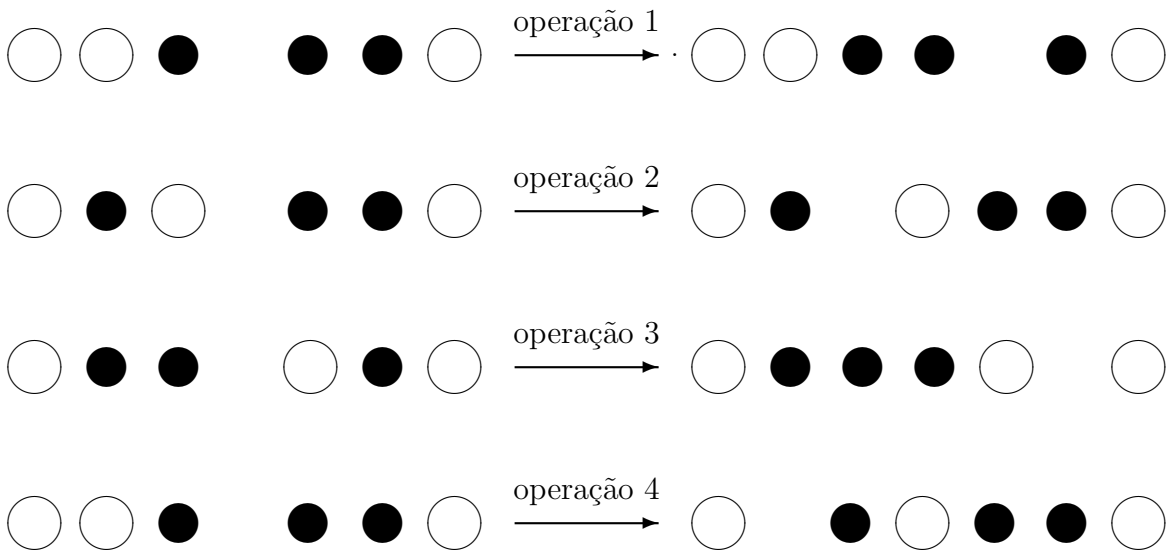

Figura 1: Operações possíveis

Demonstre em Coq que existe uma solução para este jogo. □

Exercício 14 *Considere a pequena linguagem aritmética seguinte*

```
e::= <nat> | <var> | e1 + e2 | var = e
```

considere igualmente a máquina virtual aritmética cuja linguagem é definida em Coq da seguinte forma:

```
1 (* tipos das instrucções da máquina virtual (o assembly) *)
2 Inductive instr : Set :=
3 | LOAD : ident → instr
4 | STORE : ident → instr
5 | PUSH : nat → instr
6 | DUP :  instr
7 | ADD :  instr
8 .
```

Vamos definir a seguir a semântica operacional [2] *da linguagem e da máquina virtual.*

Para a linguagem fonte, o calculo é formalizado como uma transformação dum estado contendo o ambiente de declaração e a expressão para reduzir. No início, o ambiente é vazio e no final a expressão é um valor inteiro (i, i_1, i_2 são valores inteiros, x é uma variável):

$$\frac{(x,i) \in \Gamma}{(\Gamma, x) \rightsquigarrow (\Gamma, i)} \qquad \frac{}{(\Gamma, x = i) \rightsquigarrow ((x,i) :: \Gamma, i)}$$

$$\frac{(\Gamma, e_1) \rightsquigarrow (\Gamma', i_1)}{(\Gamma, e_1 + e_2) \rightsquigarrow (\Gamma', (i_1 + e_2))} \qquad \frac{(\Gamma, e_2) \rightsquigarrow (\Gamma', i_2) \quad i = i_1 + i_2}{(\Gamma, (i_1 + e_2)) \rightsquigarrow (\Gamma', i)}$$

A semântica operacional da máquina virtual trata de modificar uma pilha de operandos (zona dos cálculos) e uma memória (zona onde são arquivadas os valores das variáveis). Nesta configuração, um programa assembly é uma lista de instruções. Na configuração inicial tanto a pilha de operando como a memória estão vazios e o programa p está por executar. Na configuração final, não resta nada por executar (a lista de instruções está vazia) e só há um valor na pilha de operandos: o valor final do cálculo. Assim sendo a semântica é definida da seguinte forma:

$$< op, (\Gamma, (x,i)), (load\ x) :: l > \rightsquigarrow < i :: op, (\Gamma, (x,i)), l >$$

$$< v :: op, \Gamma, (store\ x) :: l > \rightsquigarrow < op, (\Gamma, (x,v)), l >$$

$$< op, \Gamma, (push\ i) :: l > \rightsquigarrow < i :: op, \Gamma, l >$$

$$< i :: op, \Gamma, dup :: l > \rightsquigarrow < i :: i :: op, \Gamma, l >$$

$$< i_1 :: i_2 :: op, \Gamma, add :: l > \rightsquigarrow < (i_1 + i_2) :: op, \Gamma, l >$$

- *Defina o tipo das expressões.*
- *Defina a função de compilação das expressões aritméticas para o assembly.*
- *Defina as funções associadas a semântica operacional da linguagem fonte e da máquina virtual.*
- *Demonstre a correcção parcial da sua função de compilação.*

□

[2] De forma resumida, semântica operacional $\triangleq$ descrição rigorosa e passo a passo do processo de cálculo.