Verificação Formal de Software

Verificação da Correcção de Programas Funcionais em Coq

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

Mestrado Ciência de Computadores, FCUP 2008

Coq no desenvolvimento de Programas Certificados

- Desde o começo que o desenvolvimento do sistema Coq colocou um grande enfoque na relação com Verificação e Certificação de Programas.

- No que diz respeito a Programas Funcionais, podemos observar:
  - o Coq permite codificar a generalidade das funções sobre as quais se pretende raciocinar - os programas;
  - o seu poder expressivo é suficiente para exprimir as propriedades desejadas desses programas - as especificações;
  - o ambiente interactivo de desenvolvimento de provas permite estabelecer a ligação entre esses dois mundos - as garantias de correcção.

- Na distribuição do sistema (standard library e user contributions) encontram-se numerosos exemplos sobre o tema "algoritmos certificados" e "verificação de programas".

Verificação da Correcção em Coq:

- Abordagem Directa à Verificação de Correcção
- Extracção de Programas
- Estudo de Caso: algoritmos de ordenação
- "hands on"...

Abordagem Directa à Verificação de Correcção

Abordagem Directa

- A correcção funcional estabelece uma relação entre a especificação e a implementação.
- Numa abordagem directa à correcção de programas funcionais:
  - Especificações e Implementações são codificadas como objectos distintos em Coq:
    - A especificação consiste num predicado apropriado (relação binária);
    - A implementação consiste numa função definida em Coq (provavelmente com uma pré-condição lógica apropriada).
  - As garantias de correcção consistem num teorema da forma:

    dada uma especificação (uma relação fSpec e uma précondição fPre), a função f diz-se correcta quando:

```
forall x, fPre x -> fSpec x (f x)
```

Exemplo: divisão euclidiana

- Especificação:

```
Definition divSpec (args:nat*nat) (res:nat*nat) : Prop :=
 let (n,d):=args in let (q,r):=res in q*d+r=n /\ r<d.

Definition divPre (args:nat*nat) : Prop := (snd args)<>0.
```

- Implementação (faz uso do comando Function - coq v8.1):

```
Function div (p:nat*nat) {measure fst} : nat*nat :=
  match p with
  | (_,0) => (0,0)
  | (a,b) => if le_lt_dec b a
             then let (x,y):=div (a-b,b) in (1+x,y)
             else (0,a)
  end.
1 subgoal
  ============================
   forall (p : nat * nat) (a b : nat),
   p = (a, b) ->
   forall n : nat,
   b = S n ->
   forall anonymous : S n <= a,
   le_lt_dec (S n) a = left (a < S n) anonymous ->
   fst (a - S n, S n) < fst (a, S n)
intros; simpl.
omega.
Qed.
```

- A correcção pode assim ser estabelecida como:

```
Theorem div_correct : forall (p:nat*nat),  divPre p -> divSpec p (div p).
Proof.
unfold divPre, divRel.
intro p.
(* we make use of the specialised induction principle to conduct the proof... *)
functional induction (div p); simpl.
intro H; elim H; reflexivity.
(* a first trick: we expand (div (a-b,b)) in order to get rid of the let (q,r)=... *)
replace (div (a-b,b)) with (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
(* again a similar trick: we expand “x” and “y0” in order to use an hypothesis *)
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- e1.
omega.
(* and again... *)
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```

Extração de Programas

Extracção de Programas de objectos de Prova

- A distinção entre os universos Prop e Set destina-se a informar o Coq sobre o "conteúdo computacional" atribuído aos objectos definidos:
  - Prop - caracteriza objectos de natureza lógica (e.g. proposições, predicados, conectivas, ...);
  - Set - caracteriza objectos de natureza computacional (tipos de dados, funções, ...).
- A ideia base da Extracção consiste em estabelecer um mecanismo que "filtre" as componentes referentes aos aspectos lógicos (permanecendo unicamente os aspectos computacionais).
- Para que seja possível estabelecer esse mecanismo, é necessário impor certas restrições na interligação entre esses universos:
  - um objecto computacional pode depender da existência de provas de asserções lógicas (e.g. parcialidade, recursão bem fundada);
  - ...mas não da estrutura interna dessas provas.

- A título de exemplo, considere-se a função:

```
Definition or_to_bool (A B:Prop) (p:A\/B) : bool :=
  match p with
  | or_introl _ => true
  | or_intror _ => flase
  end.
Error:
Incorrect elimination of "p" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Set
because proofs can be eliminated only to build proofs.
```

- Se, em vez disso utilizássemos uma versão "forte" da conectiva "ou" (definida sobre o universo Set ou Type):

```
Inductive sumbool (A B:Prop) : Type := (* notation {A}+{B} *)
| left : A -> sumbool A B
| right : B -> sumbool A B.
```

- Então a definição era já possível:

```
Definition sumbool_to_bool (A B:Prop) (p:{A}+{B}) : bool :=
  match p with
  | left _ => true
  | right _ => flase
  end.
sumbool_to_bool is defined.
```

Mecanismo de Extracção

- O mecanismo de extracção "projecta" os objectos definidos removendo a carga informativa.
- O Coq permite escolher como linguagens alvo o Ocaml, Haskell e Scheme.

```
Extraction Language Haskell.

Extraction sumbool_to_bool.
sumbool_to_bool :: Sumbool -> Bool
sumbool_to_bool p =
  case p of
    Left -> True
    Right -> False
```

- O mecanismo de extração incorpora algumas preocupações com a eficiência dos programas obtidos (pré-processamento para linguagens "call-by-value"; "inlining" de definições; identificação com tipos primitivos da linguagem alvo; ...).
- Note que o resultado da extração de "sumbool" é afinal isomórfico a "bool". Assim, "sumbool" pode ser entendido como:
  - a conectiva-ou definida sobre o universo Type;
  - ou um valor boleano com justificação lógica "embebida".

If - then - else -

- A última observação sugere que o tipo "sumbool" possa ser usado na definição da construção "if-then-else" em Coq.
  - Note que uma expressão como:

    ```
    fun x y => if x<y then 0 then 1
    ```

    não faz sentido: x<y é uma proposição - não um predicado de teste (função com tipo X->X->bool);
  - Coq aceita a sintaxe:

    ```
    if test then ... else ...
    ```

    (quando "test" é uma expressão de tipo bool ou {A}+{B}, para quaisquer proposições A e B).
  - O seu significado é a expressão:

    ```
    match test with
      | left H => ...
      | right H => ...
    end.
    ```

- Alguns predicados de teste definidos na standard library:
  - le_lt_dec : forall n m : nat, {n <= m} + {m < n}
  - Z_eq_dec : forall x y : Z, {x = y} + {x <> y}
  - Z_lt_ge_dec : forall x y : Z, {x < y} + {x >= y}

Utilização de Tipos "informativos"

- O sistema de tipos do Coq permite especificar restrições nos tipos de funções - podemos assim restringir um tipo funcional aos valores que satisfazem uma determinada especificação.
- Esta estratégia explora a abilidade do Coq expressar "sub-tipos" (Σ-types). Podemos assim definir um tipo indutivo:

```
(* Notation: { x:A | P x } *)
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
```

- Note que sig é uma versão "forte" da quantificação existencial (similar à relação existente entre or e sumbool).
- Desta forma podemos especificar de forma precisa uma função apenas pelo seu tipo. Considere o seguinte tipo para um programa de ordenação de uma lista:

  forall A (l:list A), { x:list A | Permute l x & Sorted x }

  (com definições apropriadas de Permute e Sorted).
- Um habitante desse tipo pode ser entendido como "um programa de ordenação com a prova de correcção agregada".
- O mecanismo de extração permite obter uma implementação numa linguagem funcional a partir de um qualquer habitante desse tipo.

Exemplo de Extracção:

- Considere-se o predicado que identifica o último elemento de uma lista:

```
Inductive last (A:Set) (x:A) : list A -> Prop :=
| last_base : last x (cons x nil)
| last_step : forall l y, last x l -> last x (cons y l).
```

- A especificação de um programa que determine o último elemento de uma lista pode então ser expressa como:

forall A (l:list A), l<>nil -> { x:A | last x l }

- Um habitante do tipo:

```
Theorem lastCorrect : forall (A:Type) (l:list A), l<>nil -> { x:A | last x l }.
induction l.
intro H; elim H; reflexivity.
intros. destruct l. exists a; auto.
assert ((a0::l)<>nil). discriminate.
elim (IHl H0). intros r Hr; exists r; auto.
Qed.
```

- E o programa obtido por extracção:

```
Extraction Inline False_rect.
Extraction lastCorrect.
lastCorrect :: (List a1) -> a1
lastCorrect l =
  case l of Nil -> Prelude.error "absurd case"
            Cons a l0 -> case l0 of Nil -> a
                                    Cons a0 l1 -> lastCorrect l0
```

Sumário da Abordagem de Extracção de Programas

- O mecanismo de extracção do Coq:
  - explora a expressividade do sistemas de tipos para exprimir as expecidicações como restrições a tipos funcionais;
  - **não faz distinção** (pelo menos conceptualmente) entre as actividades de programação e de prova. De facto, constrói-se um habitante de um tipo que encapsula o programa funcional e a prova de correcção.
- Os programas na linguagem de programação pretendida são obtidos por extracção da componente computacional do objecto construído. A respectiva garantia de correcção resulta da correcção do próprio mecanismo de extracção.
- Algumas limitações da abordagem:
  - está orientada à "derivação de programas correctos", e não à "verificação de programas";
  - nem sem é fácil ao programador controlar a estrutura do programa obtido (e.g. a utilização de tácticas sofisticadas, ...);
  - estratégias "naturais" de prova podem não dar origem a programas eficientes;
  - pode comprometer a reutilização (e.g. provar propriedades independentes sobre uma mesma função).

Estudo de Caso:
algoritmos de ordenação

Sorting programs

- Algoritmos de Ordenação constituem exemplos interessantes:
  - a sua especificação não é trivial;
  - existem algoritmos bem conhecidos que atingem os requisitos da especificação seguindo estratégias muito distintas.
- Alguns algoritmos:
  - insertion sort
  - merge sort
  - quick sort
  - heap sort
- Especificação: o que é um algoritmo de ordenação?
  - calcula uma permutação da listra passada no argumento...
  - ...que está ordenada.

Predicado "Sorted"

- Uma caracterização simples das listas ordenadas consiste em garantir que dois elementos consecutivos da lista estejam relacionadas pela relação de ordem pretendida:

```
Inductive Sorted : list Z -> Prop :=
  | sorted0 : Sorted nil
  | sorted1 : forall z:Z, Sorted (z :: nil)
  | sorted2 :
      forall (z1 z2:Z) (l:list Z),
        z1 <= z2 ->
        Sorted (z2 :: l) -> Sorted (z1 :: z2 :: l).
```

- **Obs:** existem outras definições razoáveis para o predicado, e.g.

```
Inductive Sorted' : list Z -> Prop :=
  | sorted0' : Sorted nil
  | sorted2 :
      forall (z:Z) (l:list Z),
        (forall x, (InL x l) -> z<=x) -> Sorted (z :: l).
```

- The resulting induction principle is different. It can be viewed as a "different perspective" on the same concept.
- ...it is not uncommon to use multiple characterisations for a single concept (and prove them equivalent).

Permutation

- Para capturar uma permutação vamos fazer uso de uma definição auxiliar que conte o número de ocorrências de um elemento numa lista:

```
Fixpoint count (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0
  | (z' :: l') =>
      if Z_eq_dec z z' then S (count z l') else count z l'
  end.
```

- Duas listas estão relacionadas pela relação de permutação se, para qualquer elemento, o núero de ocorrências coincidir em ambas as listas:

```
Definition Perm (l1 l2:list Z) : Prop :=
  forall z, count z l1 = count z l2.

Notation "x ≈ y" := (equiv x y) (at level 70, no associativity).
```

- **Exercício:** prove que "Perm" é uma relação de equivalência (i.e. é uma relação reflexiva, simétrica e transitiva).

insertion sort

- A simple strategy to sort a list consist in iterate an "insert" function that inserts an element in a sorted list.
- Em haskell:

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) | x<=y = x:y:ys
                | otherwise = y:(insert x ys)
```

- Ambas as funções dispõem de codificações directas em Coq.

```
Fixpoint insert (x:Z) (l:list Z) {struct l} : list Z :=
  match l with
    nil => cons x (@nil Z)
  | cons h t =>
        match Z_lt_ge_dec x h with
          left _ => cons x (cons h t)
        | right _ => cons h (insert x t)
        end
  end.
```

(de forma análoga para isort...)

correctness proof

- Teorema da correcção:

```
Theorem isort_correct : forall (l l':list Z),
                  l'=isort l -> Perm l l' /\ Sorted l'.
```

- ...tentemos uma prova do resultado...

```
Theorem isort_correct : forall (l l':list Z),
  l'=isort l -> Perm l l' /\ Sorted l'.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl in H.
rewrite H.
1 subgoal
  a : Z
  l : list Z
  IHl : forall l' : list Z, l' = isort l -> Perm l l' /\ Sorted l'
  l' : list Z
  H : l' = insert a (isort l)
  ============================
   Perm (a :: l) (insert a (isort l)) /\ Sorted (insert a (isort l))
```

- É sensato considerar alguns lemmas auxiliares:
  - insert_Sorted - relacionando "Sorted" e "insert";
  - insert_Perm - relacionando "Perm", "cons" e "insert".

As provas...

```
Lemma insert_Sorted : forall x l, Sorted l -> Sorted (insert x l).
intros x l H; elim H; simpl; auto with zarith.
intro z; elim (Z_lt_ge_dec x z); intros.
auto with zarith.
auto with zarith.
intros z1 z2 l0 H0 H1;elim (Z_lt_ge_dec x z2);elim (Z_lt_ge_dec x z1);auto with zarith.
Qed.

Lemma insert_Perm : forall x l, Perm (x::l) (insert x l).
unfold Perm; induction l.
simpl; auto with zarith. simpl insert; elim (Z_lt_ge_dec x a); auto with zarith.
intros; rewrite count_cons_cons. pattern (x::l); simpl count; elim (Z_eq_dec z a);
intros. rewrite IHl; reflexivity. apply IHl.
Qed.
```

```
Theorem isort_correct : forall (l l':list Z),
  l'=isort l -> Perm l l' /\ Sorted l'.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl in H. rewrite H.
elim (IHl (isort l)); intros; split.
apply Perm_trans with (a::isort l).
unfold Perm; intro z; simpl; elim (Z_eq_dec z a); intros; auto with zarith.
apply insert_Perm. apply insert_Sorted; auto.
Qed.
```

Outros algoritmos de ordenação...

- A demonstração da correcção do "insertion sort" é particularmente simples. E se considerarmos outros algoritmos de ordenação como "merge sort" ou "quick sort".
- Na perspectiva do Coq, constituem certamente desafios mais interessantes:
  - a estrutura do programa não segue uma estrutura "inductiva" simples;
  - serão necessários resultados auxiliares mais elaborados...
- Um primeiro desafio a ultrapassar é a própria codificação das funções. E.g. no caso do "merge sort", será necessário codificar os seguintes programas:

```
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x<=y = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys

split [] = ([],[])
split (x:xs) = let (a,b)=split xs in (x:b,a)

merge_sort [] = []
merge_sort [x] = [x]
merge_sort l = let (a,b) = split l
               in merge (merge_sort a) (merge_sort b)
```

(o comando Function é aqui uma grande ajuda!!!)

- Projectos interessantes :-)