Verificação Formal (2021/22)

Coq (3)

O mecanismo de extração de programas do Coq permite escrever programas funcionais dentro do Coq; usar a lógica de Coq para provar propriedades de correcção sobre esses programas; e, em seguida, fazer a extração do programa para OCaml ou Haskell.

Esta aula é dedicada à especificação e prova de correcção de programas em Coq.

O website do Coq disponibiliza toda a documentação.

1 Programando e provando

Existem duas abordagens para definir funções e fornecer provas de que elas satisfazem uma determinada especificação:

- Definir essas funções com uma *especificação fraca* e, em seguida, adicionar *lemas complementares* que garantam a sua correcção. Por exemplo, definimos uma função $f : A \to B$ e provamos que $\forall x : A, R\, x\, (f\, x)$, onde R é uma relação que codifica o comportamento de entrada/saída pretendido da função.
- Fornecer uma *especificação forte* da função: o tipo dessa função determina directamente que a entrada é um valor x do tipo A e que a saída é a combinação de um valor v do tipo B e uma prova de que v satisfaz $R\, x\, v$. Esta abordagem usa tipos Σ para exprimir as restrições de especificação no tipo de uma função (restringindo o tipo do codomínio aos valores que satisfazem a especificação).

  Ao provar o teorema da especificação forte estamos a construir um habitante (um termo) deste tipo. É então possível extrair o conteúdo computacional dessa prova e assim obter uma função que satisfaz a especificação.

Carregue ficheiro `lesson3.v`. Esse ficheiro ilusta com pequenos exemplos as duas abordagens acima descritas.

Execute, passo a passo, as instruções deste ficheiro e analize o seu efeito. Atente nos comentários lá colocados e no efeito da aplicação de cada táctica de prova na evolução do estado da prova. Consulte o reference manual do Coq, quando necessário

Apresente as provas para os exercícios propostos (em comentário) ao longo do ficheiro.

2 Insertion sort

Analise o caso de estudo do *insertion sort*, onde o programa (Haskell) é extraído da prova de sua especificação.

Apresente as provas para os exercícios propostos (em comentário) ao longo do ficheiro. Execute as instruções deste ficheiro e analize o seu efeito. Atente no efeito da aplicação de cada táctica, tática automática e combinação de tácticas de prova na evolução do estado da prova.

3 Recursão não estrutural

Verificar a conversibilidade entre tipos pode exigir computação com funções recursivas. Por esse motivo, a combinação da não-terminação com tipos dependentes faz com que o *type checking* não seja decidível. Assim, o sistema Coq não permite codificar funções que não terminem (para que o *type checking* não possa divergir), nem funções parciais (para que o *type checking* não possa quebrar).

Para alcançar a decidibilidade do *type checking*, o Coq exige que as funções recursivas sejam codificadas em termos de recursores ou permite formas restritas de expressões de ponto fixo. A maneira usual de assegurar a terminação de expressões de ponto fixo é impor restrições sintáticas, restringindo todas as chamadas recursivas a serem aplicadas a termos estruturalmente menores do que o argumento formal da função.

O Coq disponibiliza o comando `Function` que permite codificar diretamente funções recursivas de forma menos restrictiva. O comando `Function` é parametrizado como uma *função de medida* que especifica como o argumento "decresce" nas chamadas recursivas da função. Ao definir uma função deste modo, são geradas obrigações de prova que devem ser verificadas para garantir a terminação.

Analise o exemplo da divisão Euclideana e complete a função merge.

4 Outros principios de indução

Ao definir um tipo indutivo o sistema Coq gera automaticamente um eliminador que permite fazer provas por indução estrutural sobre elementros desse tipo de dados. Nem sempre esse princípio de indução é o mais adquado para certo tipo de provas. De qualquer forma podemos definir outros princípios de indução que, uma vez demonstrados como correctos, poderão ser depois utilizados.

Para ilustrar esta situação, no ficheiro `lesson3.v` vai encontrar a função

```
Fixpoint split (A:Type) (l:list A) : (list A * list A) :=
  match l with
  | [] => ([],[])     (* Import ListNotations. *)
  | [x] => ([x],[])
  | x1::x2::l' => let (l1,l2) := split l' in (x1::l1,x2::l2)
  end.
```

que coloca alguns desafios ao nível da prova de propriedades. Isso é ilustrado com a prova do seguinte lema:

```
forall (A:Type) (l l1 l2: list A),
          split l = (l1,l2) -> length l1 <= length l /\ length l2 <= length l.
```

Analise este caso de estudo que envolve a definição de um novo princípio de indução.

5 Exercícios

1. Apresente uma especificação forte adequada a cada uma das funções abaixo indicadas. Faça a prova desses teoremas (especificação) e, por fim, proceda à extracção do respectivo programa Haskell.

   (a) Uma função que, dado um número natural n e um valor x, constroi uma lista de tamanho n cujos elementos são todos iguais a x.

   (b) Uma função que recebe uma lista de pares de números naturais, e produz a lista com as somas das partes constituintes de cada par da lista.

2. Recorde a função `count` que conta o número de ocorrencias de um inteiro numa lista de inteiros.

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

   (a) Prove a seguinte propriedade

```
forall (x:Z) (a:Z) (l:list Z), x <> a ->  count x (a :: l) = count x l
```

   (b) Defina uma relação indutiva que descreva a relação entre o input e o output para a função `count`, ou seja, a sua especificação.

   (c) Prove que a função dada acima satisfaz a especiificação apresentada na alínea anterior.