Why3: Verificação de programas funcionais - Documento original

Simão Cunha

5 de dezembro de 2022

Conteúdo

Resumo

Esta nota é a introdução ao uso do Why3 para o propósito específico de verificar funcionalmente programas (apenas um dos muitos usos da ferramenta). Vamos fazer uma *tour* à lógica do Why3 e a linguagens de programação e as formas como interagem e ilustram as diferentes formas de conduzir às provas indutivas. Finalmente, vamos cobrir o mecanismo de *refinement* que está disponível no `module cloning`.

Nós vamos começar por definir o algoritmos *insertion sort* sobre inteiros da seguinte forma:

1 Insertion Sort como uma função lógica

```
1  module InsertionSort
2    use int.Int
3    use list.List
4    use list.Permut
5    use list.SortedInt
6
7    function insert (i :int) (l :list int) : list int =
8      match l with
9      | Nil      -> Cons i Nil
10     | Cons h t -> if i <= h then Cons i (Cons h t) else Cons h (insert i t)
11     end
12
13   function iSort (l :list int) : list int =
14     match l with
15     | Nil      -> Nil
16     | Cons h t -> insert h (iSort t)
17     end
18
```

As propriedades de permutação e de ordem podem ser expressas usando predicados definidos em módulos de livrarias (http://why3.lri.fr/stdlib/list.html) tal como mostrado abaixo. Por exemplo, o predicado indutivo `sorted` é definido no módulo *list.Sorted* como se segue (e por sua vez é clonado no módulo `list.SortedInt`:

```
1  inductive sorted (l: list t) =
2    | Sorted_Nil:
3        sorted Nil
4    | Sorted_One:
5        forall x: t. sorted (Cons x Nil)
6    | Sorted_Two:
7        forall x y: t, l: list t.
8        le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
```

Tal como definido abaixo, as funções `insert` e `iSort` são ambas funções lógicas, e podemos escrever lemas acerca delas. Por exemplo, nós requeremos os seguintes dois lemas acerca `insert`:

```
lemma insert_sorted: forall a :int, l :list int.
                     sorted l -> sorted (insert a l)

lemma insert_perm: forall x :int, l :list int.
                   permut (Cons x l) (insert x l)
```

Agpra, reparese que `insert` tem uma definição estrutural recursiva (a chamada recursiva é feira na cauda da lista). Isto significa que ambos os lemas podem ser provados por um simples princípio de indução; em Why3, podem ser provados usando a transformação de prova `induction_ty_lex`.

Com os lemas escritos, agora podemos seguir para a prova dos resultados do algoritmo de ordenação:

```
lemma sort_sorted: forall l :list int.
                   sorted (iSort l)

lemma sort_perm: forall l :list int.
                 permut l (iSort l)
```

`iSort` é também definido de uma forma semelhante, logo ambos os lemas podem ser provados usando `induction_ty_lex`.

Podemos agora constatar o resultado final do algoritmo ao definir o que é uma função de ordenação e depois provar que o `iSort` é uma função desse tipo.

```
predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)

goal insertion_sort_correct: is_a_sorting_algorithm iSort

end
```

A prova não requer indução - resulta diretamente dos dois lemas anteriores.

2 Insertion Sort como uma função de um programa

Uma outra possibilidade é escrever o algoritmo como uma função de uma programa (WhyML) com um *contrato*, semelhantemente ao que pode ser feito numa função de um programa imperativo.

Vamos começar com a função auxiliar `insert`: o seu contrato diz que deve receber uma lista ordenada e que deve retornar também uma lista ordenada. Além disso, o resultado contém o mesmo multiconjunto de elementos da lista de *input*, extendendo com o elemento inserido.

```
1  module InsertionSortProgram
2    use  ...
3
4    let rec function insert (i: int) (l: list int) : list int
5      requires { sorted l }
6      ensures  { sorted result }
7      ensures  { permut result (Cons i l) }
8      =
9      match l with
10     | Nil -> Cons i Nil
11     | Cons h t -> if i <= h then Cons i l else Cons h (insert i t)
12     end
```

As condições de verificação geradas pelo Why3 para esta função são todas facilmente provadas com a ajuda de um SMT solver, usando uma das estratégias automáticas.

Existem aqui várias observações a serem feitas aqui. Antes de mais, note-se que, para um típico algoritmo iterativo, nós temos de dar um ou mais variantes de ciclo que permitem que o contrato da funcionalidade seja estabelecido; numa função recursiva, o contrato tem um papel de invariante ele próprio: o contrato da função que está a ser verificado é usado para gerar uma condição de verificação de acordo com a chamada recursiva.

A segunda nota aqui é, quando comparado com a versão lógica do algoritmo, não é necessária nenhuma transformação de prova manual para explicitar o princípio indutivo a ser usado: isto é explicitamente dado pela definição da função em si.

A mesma ideia é aplicada à função `iSort`, também provada corretamente com uma estratégia automática:

```
module InsertionSortProgram
    let rec function iSort (l: list int) : list int
      ensures { sorted result }
      ensures { permut result l }
      =
      match l with
      | Nil -> Nil
      | Cons h t -> insert h (iSort t )
      end

    predicate is_a_sorting_algorithm (f: list int -> list int) =
      forall al :list int. permut al (f al) /\ sorted (f al)

    goal insertion_sort_correct: is_a_sorting_algorithm iSort

end
```

Observe-se o objetivo final acima: uma declaração lógica envolvendo a função do programa "iSort". O facto de que isto pode ser escrito significa que, de facto, `iSort` não habita em ambos os *namespaces*: é simultaneamente uma função de programa e uma função lógica. Denotámos o seguinte:

- Isto é opcional: poderíamos escolher fazer a função existir apenas ao nível do programa, onde não seria possível mencioná-la em lógica (tal como se observa no objetivo acima)
- Apenas puras funções de programa, sem efeitos secundários, podem ser declaradas como *funções*. Quando argumetando acerca de programas funcionais, faz sentido fazer isto, onde vai resultar num uso específico do Why3. Na prática, programas funcionais podem ser usados na lógica mas especificados e verificados usando contratos, o que facilita as provas indutivas.

3 Mergesort como uma função de programa

Vamos agora para um algoritmo mais complicado. O algoritmo funcional `mergesort` requer duas funções auxiliares: uma para separar a lista de input em duas listas do mesmo tamanho e outra para dar *merge* a duas listas ordenadas. Nós vamos escrevê-las coomo funções WhyML equipadas de contratos; para o *split* o contrato diz que as listas resultantes contêm o mesmo nº de elementos que o input; para o *merge* queremos que as listas de input sejam ordenadas e, consequentemente, o resultado seja uma lista ordenada:

```
1  module MergeSort
2    use ...
3
4    let rec function split (l :list int) : (list int, list int)
5      ensures { let (l1,l2) = result in permut l (l1 ++ l2) }
6    =
7      match l with
8      | Nil -> (Nil, Nil)
9      | Cons x Nil -> (Cons x Nil, Nil)
10     | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
11                              in (Cons x1 l1, Cons x2 l2)
12     end
13
14   let rec function merge (l1 l2 :list int) : list int
15     requires { sorted l1 /\ sorted l2  }
16     ensures { sorted result }
17     ensures { permut (l1 ++ l2) result }
18   =
19     match l1, l2 with
20     | Nil, _ -> l2
21     | _, Nil -> l1
22     | (Cons a1 l1'), (Cons a2 l2') -> if (a1 <= a2)
23                                       then (Cons a1 (merge l1' l2))
24                                       else (Cons a2 (merge l1 l2'))
25     end
```

Todas as condições de verificação destas funções são automaticamente provadas com SMT solvers usando uma auto estratégia. É importante considerar que isto por um momento, uma vez que ambas as funções são bastante diferentes do que tinhamos na *insertion sort* e na sua função auxiliar.

- A chamada recursiva no `split` não é feita na cauda da lista, mas sim na cauda da cauda;
- `merge` toma duas listas como argumentos; a chamada recursiva alternadamente preserva uma delas, e a estrutura noutro argumento;
- Why3 é capaz de provar a terminação de ambas as funções automaticamente

A função principal `mergesort` implementa a estratégia *divide and conquer* usando os seguintes auxiliares:

```
  let rec function mergesort (l :list int)
    ensures { sorted result }
    ensures { permut l result }
    variant { length l }
  =
    match l with
    | Nil -> Nil
    | Cons x Nil -> Cons x Nil
    | _ ->   let (l1,l2) = split l in merge (mergesort l1) (mergesortl2)
    end

end
```

A primeira coisa a reparar é a presença de um *variante*. A terminação desta função não pode ser provada automaticamente, e, de facto, o Why3 vai rejeitar a definição se o variante não for dado com parte do contrato.

No entanto, o variante { `length l` } vai levar à geração de uma condição de verificação que não pode ser provada: não é possível estabelecer automaticamente que o `split` produz duas listas que são estritamente mais pequenas que o seu argumento.

De forma a permitir a terminação do `mergesort` a ser provada, podemos juntar a seguinte pós-condição no contrato do *split*:

```
ensures {
    let (l1,l2) = result in length l < 2 \/
        (length l >= 2 /\ length l1 < length l /\ length l2 < length l)
}
```

4 Mergesort como uma função lógica [Leitura opcional]

Nós começamos por ver a versão lógica do *insertion sort*, seguido pela definição WhyML do *insertion sort* e depois do *mergesort*. É apenas natural perguntar se o último algoritmo pode ser definido em lógica.

Novas dificuldades surgem, onde teremos oportunidades em discutir funcionalidades adicionais do Why3. Vamos começar por olhar para a primeira função auxiliar *split*. Pode ser definida da seguinte forma:

```
function split (l :list int) : (list int, list int) =
  match l with
  | Nil -> (Nil, Nil)
  | Cons x Nil -> (Cons x Nil, Nil)
  | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                           in (Cons x1 l1, Cons x2 l2)
  end
```

Nós queremos agora provar o seguinte lema, significando que o multiconjunto de elementos é preservado ao separar:

```
lemma split_lm: forall l :list int.
                let (l1,l2) = split l in permut l (l1 ++ l2)
```

No entanto, a transformação de prova *induction_ty_lex* não vai funcionar porque, ao contrário do *insert*, a função *split* não é definido por uma simples estrutura de recursão.

Why3 oferece uma forma de ultrapassar esta dificuldade sob a forma de uma *função de lema*. Isto pega do nível programacional do Why3 a capacidade de faer provas de indução baseada em contratos. Nós vamos definir a função WhyML que pega numa lista como argumento e escreve a pós-condição correspondente ao lema que estamos a tentar provar (logo o *split* é naturalmente lá mencionado). A definição da função simplesmente expressa o princípio indutivo que é requerido para a prova ao seguir a definição do *split*.

```
let rec lemma split_lm (l :list int) : ()
  ensures { let (l1,l2) = split l in permut l (l1 ++ l2) }
= match l with
  | Nil -> ()
  | Cons _ Nil -> ()
  | Cons _ (Cons _ l') -> split_lm l'
  end
```

A condição de verificação é provada e o contrato é inserido no contexto lógico sob a forma de lema. A função lema é muito semelhante à definição do *split* que vimos antes, mas tem o único propósito de dar a estrutura de prova para a sua pós-condição.

O mesmo pode ser feito para o *merge*:

```
function merge (l1 l2 :list int) : list int =
  match l1, l2 with
  | Nil, _ -> l2
  | _, Nil -> l1
  | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
                                    then (Cons a1 (merge l1' l2))
                                    else (Cons a2 (merge l1 l2'))
  end

let rec lemma merge_lm (l1 l2 :list int) : ()
  requires { sorted l1 /\ sorted l2  }
  ensures { sorted (merge l1 l2) }
  ensures { permut (l1 ++ l2) (merge l1 l2) }
= match l1, l2 with
  | Nil, _ -> ()
  | _, Nil -> ()
  | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
                                       then merge_lm l1' l2
                                       else merge_lm l1 l2'
      end
```

Pode ser tentador escrever o *mergesort* sob a forma da seguinte função lógica:

```
function mergesort (l :list int) : list int
= match l with
  | Nil -> Nil
  | Cons x Nil -> Cons x Nil
  | _ ->   let (l1,l2) = split l
                in  merge (mergesort l1) (mergesort l2)
  end
```

Isto não vai funcionar: a terminação não pode ser automaticamente estabelecida (porque as funções recursivas não estão a serem feitas em sublistas do argumento l), logo não é possível definir o *mergesort* como uma "função".

Existe uma forma de ultrapassar isto. Lembre-se que sabemos como definir o *mergesort* como uma função WhyML que também é uma função lógica. O que não estamos aqui a conseguir fazer é definir a função lógica que *não é uma função do programa*. A dificuldade é que, desde que a terminação automática não pode ser estabelecida, nós queremos provar um *variante*, mas os variantes só podem ser usados em funções de programas.

Isto pode soar um pouco confuso no início, mas o Why3 dá uma forma de definir funções lógicas que não são funções de programas, usando construtores de programas e contratos. Funções *ghost* são escritas em código WhyML que pode ser interpretado logicamente e não têm o objetivo de serem executadas. *Mergesort* pode ser definido como se segue num namespace lógico, como o comprimento da lista de argumento como variante:

```
let rec ghost function mergesort (l :list int) : list int
  variant { length l }
= match l with
  | Nil -> Nil
  | Cons x Nil -> Cons x Nil
  | _ ->   let (l1,l2) = split l
                in  merge (mergesort l1) (mergesort l2)
  end
```

A correção é estabelecida através da seguinte função de lema, onde as afirmações *assert* agem como lemas intermédios:

```
let rec lemma mergesort_lm (l :list int) : ()
  ensures { sorted (mergesort l) }
  ensures { permut l (mergesort l) }
  variant { length l }
=
  match l with
  | Nil -> ()
  | Cons _ Nil -> ()
  | _ ->   let (l1,l2) = split l
           in assert { permut l (l1 ++ l2) } ;
              mergesort_lm l1 ;
              mergesort_lm l2 ;
              assert { permut l (mergesort l1 ++ mergesort l2) }
  end
```

5 Refinamento em WhyML

Considera a seguinte alternativa WhyML do *mergesort*:

```
module MergeSort
  use ...

  val function split (l :list int) : (list int, list int)
    ensures { let (l1,l2) = result in length l < 2 \/
              (length l >= 2 /\ length l1 < length l /\ length l2 <length l) }
    ensures { let (l1,l2) = result in permut l (l1 ++ l2) }


  val function merge (l1 l2 :list int) : list int
    requires { sorted l1 /\ sorted l2  }
    ensures { sorted result }
    ensures { permut (l1 ++ l2) result }


  let rec function mergesort (l :list int)
    ensures { sorted result }
    ensures { permut result l }
    variant { length l }
  = match l with
    | Nil -> Nil
    | Cons x Nil -> Cons x Nil
    | _ ->   let (l1,l2) = split l
             in merge (mergesort l1) (mergesort l2)
    end

end
```

Difere da seguinte versão no ponto onde não são dadas definições às funções auxiliares. Em vez disso, são apenas declaradas (usando `val`) com a assinatura e contrato. Os contratos contêm toda a informação necessária para permitir a correção do *mergesort* a ser provada. A sua condição de verificação pode ser provada da mesma forma que na versão anterior.

Se quisermos uma definição concreta do *mergesort*, ao dar definições do *split* e *merge*, nós devemos agora *clonar* o módulo *MergeSort* dentro de um novo módulo, da seguinte forma:

```
module MergeSortRefnm
  use ...

  let rec function split (l :list int) : (list int, list int)
    ensures { let (l1,l2) = result in length l < 2 \/
              (length l >= 2 /\ length l1 < length l /\ length l2 < length l) }
    ensures { let (l1,l2) = result in permut l (l1 ++ l2) }
  = match l with
    | Nil -> (Nil, Nil)
    | Cons x Nil -> (Cons x Nil, Nil)
    | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                              in (Cons x1 l1, Cons x2 l2)
    end

  let rec function merge (l1 l2 :list int) : list int
    requires { sorted l1 /\ sorted l2 }
    ensures { sorted result }
    ensures { permut (l1 ++ l2) result }
    variant { length (l1 ++ l2) }
  = match l1, l2 with
    | Nil, _ -> l2
    | _, Nil -> l1
    | (Cons a1 l1'), (Cons a2 l2') -> if (a1 <= a2)
                                      then (Cons a1 (merge l1' l2))
                                      else (Cons a2 (merge l1 l2'))
    end

  clone MergeSort with val split, val merge

  goal thisReallyWorks :
    forall l :list int. let ls = mergesort l
                        in sorted ls /\ permut ls l

end
```

Clonar o módulo *MergeSort* dentro do *MergeSortRefnm* vai copiar o atual para o antigo, instanciando os elementos mencionados depois de *with*: as funções *split* e *merge* são dadas como a definição do módulo de clone. Observa-se que isto vai gerar condições de verificação a estabelecer que o refinamento dos contratos declarados no *MergeSort* com as definições de *MergeSortRefnm* que se consideram corretas.

No presente exemplo, ambas as funções têm exatamente o mesmo contrato na função definida como uma declaração clonada, logo estas condições de verificação são provadas trivialmente. Juntas, as condições de verificação geradas para os dois módulos implicam que tenhamos uma correta implementação do *merge sort*, porque:

- A validade das condições de verificação do *MergeSort* implicam que a implementação do *mergesort* esteja correta se o *split* e *merge* forem implementadas de acordo com as especificações dadas no módulo;
- A validade das condições de verificação do *MergeSortRefnm* implicam que as implementações do *split* e *merge* estão corretas de acordo com as especificações dadas no módulo e, além

disso, para as condições de verificação no clone, estão também corretas no que diz respeito aos contratos no módulo *MergeSort* (o que acontece por ser o mesmo).