Guião para aula laboratorial de Verificação Formal (2018/19)

Why3 (2)

Esta aula é dedicada à utilização do Why3 na sua vertente de ferramenta de verificação dedutiva.

No website do Why3 encontratrá diversa documentação e muitos exemplos.

Este guião tem por base o tutorial *"Deductive Program Verification with Why3"*, de Jean-Christophe Filliâtre, assim como exemplos do manual do Why3.

1 Introdução

A plataforma Why3 fornece uma linguagem lógica (chamada Why) e uma linguagem de programação (chamada WhyML). A lógica é uma extensão da lógica de primeira ordem com polimorfismo, tipos de dados algébricos e predicados indutivos. A linguagem de programação é uma linguagem do estilo ML de primeira ordem, com características imperativas, concordância de padrões e excepções. As duas linguagens estão intimamente ligadas: qualquer símbolo lógico pode ser usado nos programas, e a lógica é usada para especificar programas (via pré e pós-condições, invariantes de ciclo, etc.).

A extracção de condições de verificação é feita com base no cálculo da *weakest precondition*. As fórmulas resultantes são então sujeitas a várias transformações, para depois serem enviadas para (vários) demonstradores de teoremas externos.

2 Verificação dedutiva de programas

2.1 demo_turing.mlw

Vamos considerar o exemplo apresentado por Alan Turing em 1949 no artigo *"Checking a Large Routine"* (provavelmente a primeira prova de um programa). O programa calcula $n!$ usando apenas adições.

$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do**
  $v \leftarrow u$
  for $s = 1$ **to** r **do**
    $u \leftarrow u + v$

A variável n contém o parâmetro de entrada e não é modificada. Quando o programa termina, a variável u contém $n!$, o fatorial de n. Existem dois ciclos aninhados. No ciclo externo, a variável u contém $r!$ e seu conteúdo é copiado na variável v. O ciclo interno calcula $(r + 1)!$ em u, por adições repetidas de v.

Vamos provar a correcção deste programa usando Why3. No ficheiro `demo_turing.mlw` encontrará o programa acima na sintaxe do WhyML, a linguagem de programação do Why3.

Declarações de programas são agrupadas em *módulos* (similar, em termos de programas, às teorias lógicas).

```
module CheckingALargeRoutine
  use int.Int
  use int.Fact
  use ref.Ref
```

Primeiro importam-se da biblioteca do Why3, para além das teorias `int.Int` e `int.Fact` o módulo `ref.Ref` das *referências*. Tal como no ML, referências são variáveis mutáveis: uma referência é criada com a função `ref`, acessamos ao conteúdo da referência `x` com `!x`, e atribuímos com `x := e`. Podemos agora escrever uma função `routine` tomando um inteiro `n` como argumento e calculando o seu factorial de acordo com o algoritmos acima.

```
  let routine (n: int) : int
    requires { n >= 0 }
    ensures { result = fact n }
  = let u = ref 1 in
    for r = 0 to n-1 do
      let v = !u in
      for s = 1 to r do
        u := !u + v
      done
    done;
    !u
  end
```

A especificação é introduzida com palavras-reservadas `requires` e `ensures`. `requires` introduz a pré-condição $n \geq 0$ e `ensures` a pós-condição `result = fact n`. Dentro da pós-condição, a variável `result` representa o valor retornado pela função.

Ao carregar o ficheiro no Why3, vemos que é gerada a seguinte condição de verificação (VC):

$$\forall n, n \geq 0 \Rightarrow (0 \leq n-1 \Rightarrow \forall u, u = \texttt{fact}\, n) \wedge (0 > n-1 \Rightarrow 1 = \texttt{fact}\, n)$$

A última parte corresponde ao caso antes de entrar no ciclo (já que $n = 0$) e temos que mostrar que $\texttt{fact}\, n = 1$, o que é demonstrável. A outra parte corresponde ao caso em que se entra no ciclo. Aqui temos que demonstrar que o valor final da variável `u` é o factorial de `n`, mas isso não parece ser demonstrável uma vez que não temos nunhuma informação sobre `u`.

1. Tente resolver a VC com os solvers dispiníveis. Poderá constatar que nenhum a consegue resolver.

2. Aplique agora a estratégia *"Split VC"*, e tende de novo.

De facto, temos que primeiro definir um invariante de ciclo para obter informações sobre o valor final de `u`. No nosso caso, este invariante deverá indicar que `u` contém o factorial de `r`. Um invariante é introduzido com palavra-reservada `invariant`. Neste caso será:

```
for r = 0 to n-1 do invariant { !u = fact r }
```

1. Acrescente este invariante ao programa.

2. Faça o *"Refresh"* da sessão, e analise as VCs geradas.

3. Tente agora provar essas VCs usado os SMT solvers.

Como poderá constatar, quase todas as VCs são resolvidas. A exceção é a VC relativa à preservação do inveriante de ciclo que acabamos de adicionar. Temos portanto que equipar o ciclo interior com um invariante:

```
for s = 1 to r do invariant { !u = s * fact r }
```

É agora fácil terminar a prova usando qualquer solver. Neste caso não há necessidade de provar a terminação, porque ela é automáticamente garantida no caso dos ciclos `for`.

2.2 Sum and Maximum

Vamos agora ilustrar a utilização de arrays. O Why3 fornece uma biblioteca de arrays no módulo `array.Array`.

Este módulo declara um tipo polimórfico `array 'a`, uma operação de acesso `a[e]`, uma operação de atribuição `a[e1] <- e2` e várias operações como `create`, `length`, `append`, `sub`, `copy`, etc.

Considere um programa que recebe um array a de n números naturais e calcula a sua soma e o seu máximo. Considere ainda que queremos verificar que o programa cumpre o seguinte contrato

Pré-condição: $n = \texttt{length}\, a \wedge \forall i.\, 0 \leq i < n \Rightarrow a[i] \geq 0$

Pós-condição: $\texttt{sum} \leq n \times \texttt{max}$

Uma codificação possível, em whyML, para este triplo de Hoare será

```
module MaxAndSum
  use int.Int
  use ref.Ref
  use array.Array

  let max_sum (a: array int) (n: int) : (sum: int, max: int)
    requires { n = length a }
    requires { forall i. 0 <= i < n -> a[i] >= 0 }
    ensures  { sum <= n * max }
  = let sum = ref 0 in
    let max = ref 0 in
    for i = 0 to n - 1 do
      if !max < a[i] then max := a[i];
      sum := !sum + a[i]
    done;
    !sum, !max
end
```

1. Carregue o ficheiro `max_sum.mlw` no Why3.
2. Tente resolver a VC com os solvers dispiníveis. Poderá constatar que nenhum a consegue resolver.

3. Aplique agora a estratégia *"Split VC"*, e tende de novo.

4. Analise a VC que falha e complete a prova, escrevendo o invariante de ciclo apropriado. Lembre-se que pode tentar completar a prova com vários SMT solvers.

Como deve ter reparado, o contrato acima não dá garantias de que `sum` e `max` contêm os valores da soma e do máximo elemento do array. Abaixo apresenta-se uma nova versão do contrato onde estes requisitos são comtemplados. Note que a função lógica `sum` dá o somatório dos valores de um array entre duas posições. Esta função vem do módulo `array.ArraySum`. O predicado `is_max` está a ser definido no próprio módulo. Repare como o invariante do ciclo tem agora de ter mais informação.

```
module MaxAndSum2
  use int.Int
  use ref.Ref
  use array.Array
  use array.ArraySum

  predicate is_max (a: array int) (l h: int) (m: int) =
    (forall k: int. l <= k < h -> a[k] <= m) &&
    ((h <= l && m = 0) ||
     (l <  h && exists k: int. l <= k < h && m = a[k]))

  let max_sum (a: array int) (n: int)
    requires { n = length a /\ forall i:int. 0 <= i < n -> a[i] >= 0 }
    ensures  { let s, m = result in
               s = sum a 0 n /\ is_max a 0 n m /\ s <= n * m }
  = let s = ref 0 in
    let m = ref 0 in
    for i = 0 to n - 1 do
      invariant { !s = sum a 0 i /\ is_max a 0 i !m /\ !s <= i * !m }
      if !m < a[i] then m := a[i];
      s := !s + a[i]
    done;
    !s, !m
end
```

2.3 Inverting an Injection

Pretendemos agora inverter um array a injectivo, de n elementos, no subintervalo de 0 a $n-1$. Isto é, o array de saida b deve ser tal que $b[a[i]] = i$ para $0 \leq i < n$. O cógido da função é muito simples:

```
  for i = 0 to n - 1 do b[a[i]] <- i done
```

Para o contrato é conveniente definir predicados para as propriedades "ser injetivo", "ser sobrejectivo" e "estar no subintervalo". Estas são declarações puramente lógicas:

```
predicate injective (a: array int) (n: int) =
  forall i j. 0 <= i < n -> 0 <= j < n -> i <> j -> a[i] <> a[j]

predicate surjective (a: array int) (n: int) =
  forall i. 0 <= i < n -> exists j: int. (0 <= j < n /\ a[j] = i)

predicate range (a: array int) (n: int) =
  forall i. 0 <= i < n -> 0 <= a[i] < n
```

Usando esses predicados, podemos formular o lema de que qualquer array injectivo de tamanho n dentro do intervalo de 0 a $n-1$, também é sobrejetivo:

```
lemma injective_surjective:
  forall a: array int, n: int.
    injective a n -> range a n -> surjective a n
```

Optou-se por declarar isso como lema e não como axioma, pois é realmente demonstrável, mas requer indução... Finalmente, podemos dar ao código uma especificação,

```
let inverting (a: array int) (b: array int) (n: int)
  requires { n = length a = length b }
  requires { injective a n /\ range a n }
  ensures  { injective b n }
= for i = 0 to n - 1 do
    b[a[i]] <- i
  done
```

Carregue o ficheiro `invert_injection.mlw` no Why3. Escreva um inveriante de ciclo adquado para provar a VC gerada.

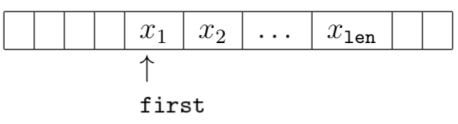
2.4 Buffer circular

O código deste exemplo encontra-se no ficheiro `ring_buffer.mlw`, e serve para ilustrar as noções de *ghost code* e *invariante de tipo*.

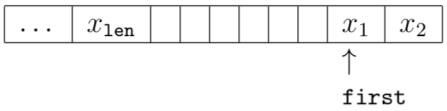
A ideia é implementar um buffer circular num array. Depois de importar algumas bibliotecas do Why3, o buffer é implementado usando o seguinte tipo *record*:

```
type buffer 'a = {
    mutable first: int;
    mutable len  : int;
            data : array 'a;
  }
```

Os elementos são armazenados no array `data` começando no índice `first` e tem `len` elementos:



Esta estrutura de dados possui uma interface de fila de espera: se a fila não estiver cheia, um novo elemento pode ser adicionado à direita de x_{len}; se a fila não estiver vazia, x_1 pode ser exibido. Elementos podem-se distribuir dentro dos limites da array, da seguinte forma circular:



Para especificar as várias operações sobre essa estrutura de dados, é conveniente modelar seu conteúdo como uma lista, ou seja, a lista $[x_1, x_2, \ldots, x_{len}]$, e armazená-la dentro do próprio record, num campo *gost*:

```
type buffer 'a = {
  mutable first: int;
  mutable len  : int;
          data : array 'a;
  ghost mutable sequence: list 'a;
}
```

Um campo *ghost* (mais geralmente, qualquer código *ghost*) é para ser usado apenas na especificação, e não pode interferir no código regular no seguinte sentido:

- o código ghost não pode modificar dados regulares (mas pode acessá-lo apenas para leitura);
- código ghost não pode modificar o fluxo de controle do código regular;
- código regular não pode acessar ou modificar dados ghost.

Por outras palavras, dados ghost e código ghost podem ser removidos sem afetar a execução do programa. No exemplo acima, isso significa que apenas o código ghost poderá acessar e atribuir o campo `sequence`.

Para relacionar o conteúdo dos campos regulares e do campo ghost, equipamos o tipo `buffer 'a` com um invariante de tipo. Um invariante de tipo tem que ser válido nas "frontreiras" da cada função: é assumido à entrada da função (no que se refere a argumentos da função e a variáveis globais), e tem que ser garantido na saída da função (no resultado da função e em valores modificados pela função).

Um invariante de tipo é introduzido após a definição de tipo, usando a palavra-reservada `invariant`. No nosso caso, ele primeiro diz que o array `data` contêm pelo menos tamanho 1 e que `first` é um índice válido de `data`:

```
invariant {
  let size = A.length data in
  0 <= first <  size /\ ...
```

Depois adicionamos que o campo `len` não pode ser maior que o número de elementos em `data`:

```
  0 <= len <= size /\ ...
```

Finalmente, relacionamos o conteúdo do array `data` com a lista no campo `sequence`. Primeiro, dizemos que `sequence` tem comprimento `len`:

```
len = L.length sequence /\ ...
```

Depois dizemos que os elementos em `sequence` são exactamente os que estão no array `data` nos índices `first`, `first+1`, ... Para captar a circularidade em torno do array, criamos dois casos:

```
  forall i: int. 0 <= i < len ->
    (first + i < size ->
       nth i sequence = Some data[first + i]) /\
    (0 <= first + i - size ->
       nth i sequence = Some data[first + i - size])
}
by { first = 0; len = 0; data = make 1 (any 'a); sequence = Nil }
```

A função `nth`, importada da biblioteca do Why3, retorna a i-ésimo elemento da lista, como `Some x`, se existir, e `None` se não existir.

Dado esse invariante de tipo, podemos agora usar o campo `sequence` para especificar operações sobre o buffer circular. Por exemplo, a função `push` tem uma pós-condição indicando que a sequência foi estendida para a direita com um novo elemento:

```
let push (b: buffer 'a) (x: 'a) : unit
  ...
  ensures  { b.sequence = (old b.sequence) ++ Cons x Nil }
```

A notação `old v` representa o valor inícial da variável `v`, ou seja o seu valor à entrada da função.

A especificação completa desta e de outras operações é dada no ficheiro `ring_buffer.mlw`. Analise-o, carregue-o no Why3 e tente provar o maior número de VCs que conseguir.