C2. Invariantes de Ciclo

Copiar link

Notas prévias

Neste módulo assumiremos que na linguagem de programação utilizada não é possível atribuir valores às variáveis que são inputs do programa. Isto permitirá simplificar as respectivas especificações (porque torna desnecessária a utilização de variáveis auxiliares).
Na linguagem de programação da ferramenta Dafny, que utilizaremos neste módulo para testar os invariantes de ciclo, não é possível atribuir valor a parâmetros.

A linguagem Dafny incorpora a **sintaxe** habitual de expressões da linguagem C (`==` como operador de comparação de igualdade, `&&` e `||` como operadores Booleanos, etc. No entanto, para a atribuição é utilizado o operador `:=` , muito comum em linguagens "brinquedo".

Invariantes de Ciclo

Estabelecer a correcção de algoritmos que incluam ciclos implica considerar qualquer número de iterações; no entanto, não é viável proceder a esta análise por casos, de forma exaustiva.

Recorremos por isso à noção de *invariante de ciclo* — uma propriedade (fórmula de primeira ordem) que se mantém verdadeira em todas as iterações, e que reflecte as transformações de estado efectuadas durante a execução do ciclo.

A ideia é que se o invariante se mantém verdadeiro ao longo da execução, ele será ainda verdadeiro à saída do ciclo, e deverá ser suficientemente forte para permitir provar a pós-condição desejada para o ciclo.

Raciocinar com um invariante de ciclo corresponde à ideia de prova indutiva no número de iterações de uma execução (que termina) do ciclo. Assim, para provar que uma fórmula I é um invariante, devemos

1. **Inicialização**: Mostrar que I é verdade à entrada do ciclo (i.e. antes de se iniciar a primeira iteração) — *caso de base*
2. **Preservação:** Mostrar que, assumindo que I é verdade no início de uma iteração arbitrária (i.e. assumindo que a condição do ciclo é satisfeita), então será satisfeito no final dessa iteração — *caso indutivo*

Sendo I de facto um invariante, é ainda preciso mostrar que é útil:

3. **Utilidade:** Mostrar que o invariante I, juntamente com a negação da condição do ciclo (uma vez que terminou a sua execução), implica a verdade da pós-condição

Estas 3 propriedades podem por sua vez ser expressas por triplos de Hoare, envolvendo:
1. para a inicialização, o código que antecede o ciclo
2. para a preservação, o código que constitui o corpo do ciclo
3. para a utilidade, o código que sucede ao ciclo

Exemplo: Divisão Inteira

Especificação de um programa que calcula a divisão de x por y, colocando o quociente em q e o resto em r:

$$\{x \geq 0 \land y > 0\} \ \textbf{divide} \ \{0 \leq r < y \land q * y + r = x\}$$

(*especificação simplificada sem vars. auxiliares*)

O seguinte programa está de acordo com esta especificação. Informalmente, conta “o número de vezes que y cabe em x”.

```
r := x;
q := 0;
while (y <= r) {
```

```
  r := r-y;
  q := q+1;
}
```

Simulemos uma execução deste programa para $x = 14$ e $y = 3$, escrevendo o valor das variáveis r e q, alteradas pelo programa, **à entrada de cada iteração** do ciclo:

r	q	$q * y + r$
14	0	14
11	1	14
8	2	14
5	3	14
2	4	14

Constata-se que a expressão $q * y + r$ mantém o seu valor ao longo da execução, à entrada de cada iteração, e que este valor é igual a ao valor de x.

Por outro lado, o valor de r é não-negativo ao longo de toda a execução, pelo que temos o seguinte invariante de ciclo:

$$I \;\equiv\; \mathbf{0 \leq r \;\wedge\; q * y + r = x}$$

Este exemplo encaixa num cenário típico bastante simples para o caso de programas que terminam com um ciclo:

> *A pós-condição é equivalente à conjunção do invariante e da negação da condição do ciclo.*

Vejamos como estabelecer a correcção do programa com este invariante:

- *Inicialização do invariante.* Corresponde ao triplo de Hoare:
  $\{x \geq 0 \wedge y > 0\}\;\; \mathtt{r := x;\; q := 0;}\;\; \{I\}$

  É imediato constatar que é válido, uma vez que $0 \leq 0 \wedge 0 * y + x = x$

- *Preservação do invariante.* Corresponde ao triplo de Hoare:
  $\{I \wedge y \leq r\}\;\; \mathtt{r := r - y;\; q := q + 1;}\;\; \{I\}$

  Intuitivamente, admitimos que $I \equiv 0 \leq r \wedge q * y + r = x$ é verdade à entrada de uma iteração qualquer. Temos também $y \leq r$. Queremos mostrar que I voltará a ser verdade depois da execução da iteração (arbitrária) em causa.

Ora, os valores de r e q no início da próxima iteração serão respectivamente dados por $r - y$ e $q + 1$. Queremos então mostrar que a seguinte condição é verdadeira:
$0 \leq (r - y) \wedge (q + 1) * y + (r - y) = x$
$\equiv y \leq r \wedge q * y + r = x$

E efectivamente,
$(0 \leq r \wedge q * y + r = x) \wedge y \leq r \quad \rightarrow \quad y \leq r \wedge q * y + r = x$

Ou seja, o invariante é preservado por qualquer iteração do ciclo.

- *Utilidade do invariante.* Corresponde ao triplo de Hoare:
  $\{I \wedge \neg(y \leq r)\} \;\; \{\;\} \;\; \{0 \leq r < y \wedge q * y + r = x\}$

  A utilidade é neste caso trivial, uma vez que o programa termina com o ciclo, não sendo executadas quaisquer instruções subsequentes. Ora, $I \wedge \neg(y \leq r)$ implica (neste caso é mesmo equivalente!) a pós-condição $0 \leq r < y \wedge q * y + r = x$

Observe-se que para cada um dos triplos acima pode ser gerada uma *condição de verificação* usando a técnica descrita em +C1. Introdução à Análise de Correcção — Especificação de Programas.

Verificação com a Ferramenta Dafny
Por forma a confirmar se um invariante serve ou não o nosso propósito, podemos recorrer a uma ferramenta automática de verificação de programas. A ferramenta Dafny, desenvolvida pela Microsoft Research, é um verificador de programas escritos numa linguagem simples como a que temos considerado acima. A unidade básica (rotina) de código é aqui o *método.*

Note-se que:

- O programa de divisão dado acima será escrito como um método tendo x e y como inputs, e q e r como outputs. Todas estas variáveis são de tipo nat
- A especificação deste método será escrita utilizando as palavras reservadas *requires* para a pré-condição, e *ensures* para a pós-condição, logo a seguir ao cabeçalho do método

- O invariante de ciclo será escrito logo a seguir à condição do ciclo, usando a palavra reservada *invariant*

Temos assim:

```
method divide(x: nat, y: nat) returns (r: nat, q:nat)
   requires y > 0
   ensures  0 <= r < y
   ensures q*y+r == x
{
   r := x;
   q := 0;
   while (y <= r)
      invariant q*y+r == x
   {
     r := r-y;
     q := q+1;
   }
}
```

O programa poderá ser verificado online aqui. Obtemos como resultado:

```
Dafny 2.2.0.10923
Dafny program verifier finished with 1 verified, 0 errors
```

[Permalink para este exemplo em rise4fun.com]

Exemplo: Factorial

Vejamos como verificar a correcção de um programa de cálculo da noção de factorial de um número natural n. Isto corresponderá a provar a validade do seguinte triplo de Hoare:

$$\{n \geq 0\} \;\; \textbf{ComputeFact} \;\; \{f = n!\}$$

O núcleo do programa é o seguinte:

```
f := 1;
i := 1;
while (i<=n) {
    f := f*i;
    i := i+1;
}
```

O invariante I deverá ser tal que os triplos de Hoare seguintes sejam válidos:

1. **Inicialização**: $\{n \geq 0\}$ $\texttt{f} := \texttt{1};\ \texttt{i} := \texttt{1};$ $\{I\}$
2. **Preservação**: $\{I \wedge i \leq n\}$ $\texttt{f} := \texttt{f} * \texttt{i};\ \texttt{i} := \texttt{i} + \texttt{1};$ $\{I\}$
3. **Utilidade**: $\{I \wedge \neg(i \leq n\}$ $\{\ \}$ $\{f = n!\}$

Para caracterizar o invariante apropriado, simulemos uma execução deste programa. O par de variáveis (i, f) toma sucessivamente os seguintes valores: $(1,1), (2,1), (3,2), (4,6), (5,24), \ldots$

É fácil observar que o valor de f à entrada de uma iteração corresponde ao factorial de $i - 1$.

É imediato ver que:

- este invariante é bem inicializado, uma vez que $0! = 1$
- ele é também preservado, uma vez que o corpo do ciclo multiplica f por i e incrementa esta variável

Notemos também que o valor de i varia entre 0 e $n + 1$. Esta informação deverá constar do invariante.

À saída do ciclo, sendo a condição $i \leq n$ falsa, teremos então que o valor final de i será $i = n + 1$, e o invariante implicará que $f = n!$ como desejado (*utilidade* do invariante para provar a pós-condição).

Vejamos agora como efectuar esta verificação em Dafny. As seguintes definições incorporam já a especificação e invariante discutidos:

```
function fact(n: int): int
   requires n >= 0
{
   if n == 0 then 1
   else n * fact(n-1)
}

method ComputeFact (n: int) returns (f: int)
   requires n >= 0
   ensures f == fact(n)
{
   f := 1;
   var i := 1;
   while (i<=n)
     invariant i <= n+1
     invariant f == fact(i-1)
   {
        f := f*i;
        i := i+1;
    }
}
```

Note-se a presença de uma *função lógica*, recursiva, que capta a definição matemática de factorial, que não existe à partida na teoria do Dafny. Esta função $fact$ é depois usada na especificação do programa, quer na pós-condição quer no invariante de ciclo.

Observe-se também que as variáveis locais que não sejam parâmetros ou outputs de um método devem ser declaradas com a palavra reservada `var`, como `i` no exemplo acima.

A verificação é levada a cabo com sucesso, o que mostra que a escolha do invariante foi apropriada, e também que o programa efectivamente calcula o factorial de n.

```
Dafny program verifier finished with 3 verified, 0 errors
```

[permalink para este exemplo]

Exercício

Uma outra noção que se pode definir recursivamente é a sequência de números de Fibonacci. Também aqui é necessário definir a sequência através de uma função lógica fib, que se pode depois utilizar na especificação e invariante.

1. Complete o invariante de ciclo na definição da função **ComputeFib**, que calcula o n-ésimo número de Fibonacci.
2. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verifique o programa recorrendo ao Dafny.

```
function fib(n: nat): nat
{
   if n == 0 then 0
   else if n == 1 then 1
        else fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
   requires n > 0
   ensures b == fib(n)
{
   var i := 1;
   var a := 0;
   b := 1;
   while i < n
```

```
      invariant ...
    {
      b := b+a;
      a := b-a;
      i := i + 1;
    }
}
```

Exemplo: Contagem de ocorrências num *array*

O seguinte programa conta o número de ocorrências do valor guardado em k no array $vector$, entre as posições 0 e n-1.

```
result := 0;
i := 0;
while (i<n) {
    if (v[i] == k) result := 1+result;
    i := i+1;
}
```

A noção de contagem de ocorrências de um elemento num array não é primitiva, e terá de ser definida. Em Dafny definiremos uma função recursiva que exprima esta noção, como se segue:

```
function countn(v:array<int>, n:int, k:int) :int
   requires 0 <= n <= v.Length
   reads v
{
   if (n==0) then 0
   else if (v[n-1] == k) then 1+countn(v, n-1, k)
        else countn(v, n-1, k)
}
```

Note-se que, apesar de não se tratar de código, mas sim de uma definição ao nível lógico, esta função necessita de ser anotada com uma pré-condição que delimita o valor de n entre 0 e o limite alocado para o array (`v.Length`). É também necessária a cláusula `reads v`, que autoriza o acesso da função ao *array* (a discussão desta anotação está fora do nosso âmbito aqui).

Dispondo da função lógica countn, não teremos dificuldade em escrever uma especificação para o programa acima, a que chamaremos $\mathbf{Countn}$:

$$\{0 \leq n\} \ \ \mathbf{Countn} \ \ \{result = countn(vector, n, k)\}$$

É fácil exprimir um invariante apropriado para este programa. No início de uma iteração, foram contadas as ocorrências entre os índices 0 e $i - 1$. Logo:

$$I \equiv (0 <= i <= n) \ \wedge \ result = countn(vector, i, k)$$

É bastante imediato entender que se trata de facto de um invariante do ciclo, sendo bem inicializado ($result = 0$ e $countn(vector, 0, k) = 0$ por definição) e preservado pelo corpo do ciclo, que actualiza a contagem olhando para a o índice i, que é depois incrementado.

A utilidade do invariante também é evidente, uma vez que, terminando o ciclo com $i = n$, a pós-condição é consequência de I.

Para verificar o programa escrevemos o seguinte método Dafny. Note-se que é necessário fortalecer a pré-condição, delimitando n com o tamanho alocado para o *array*.

```
method Countn(vector:array<int>, n:int, k:int) returns (result:int)
    requires 0 <= n <= vector.Length
    ensures result == countn(vector, n, k)
{
    result := 0;
    var i := 0;
    while (i<n)
      invariant 0 <= i <= n
```

```
      invariant result == countn(vector, i, k)
    {
      if (vector[i] == k) { result := 1+result; }
      i := i+1;
    }
}
```

A execução da ferramenta não identifica quaisquer erros, comprovando que o programa é correcto face à especificação:

```
Dafny program verifier finished with 3 verified, 0 errors
```

[Permalink para este exemplo]

Exercício
Considere-se o problema de somar todos os elementos de um *array* com N posições

$$\{0 \leq n\} \ \ \mathbf{Sum} \ \ \{result = \sum_{k=0}^{n-1} vector[k]\}$$

Sendo Sum o seguinte programa:

```
result := 0;
i := 0;
while (i < n)  {
  result := vector[i] + result;
  i := i+1;
}
```

1. Defina um invariante para o programa acima
2. Escreva os triplos de Hoare correspondes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verifique o programa recorrendo ao Dafny. Para isso:
   a. Escreva uma função lógica recursiva em Dafny que corresponda à noção de somatório entre dois índices de um *array*

b. Defina um método que lhe permita provar que o programa é correcto face à especificação acima.

Repita depois o exercício para as seguintes versões alternativas do programa (ambas correctas):

```
result := 0;
i := -1;
while (i < n-1)  {
  i := i+1;
  result := vector[i] + result;
}
```

```
result := 0;
i := n;
while (i > 0)  {
  i := i-1;
  result := vector[i] + result;
}
```

Exercícios

Multiplicação de números inteiros

Considere os três algoritmos seguintes de multiplicação de números inteiros, com a seguinte especificação:

$$\{y \geq 0\}\ \ \mathbf{mult}\ \ \{r = x.y\}$$

Para cada algoritmo,

1. Defina um invariante apropriado para o ciclo

2. Escreva os triplos de Hoare correspondes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verifique o programa recorrendo ao Dafny, definindo métodos com as anotações que entender.

```
r := 0;
i := 0;
while (i<y)
{
    r := r+x;
    i := i+1;
}
```

```
r := 0;
i := y;
while (i>0)
{
    r := r+x;
    i := i-1;
}
```

```
r := 0;
xx := x;
yy := y;
while (yy>0)
{
    if (yy%2 != 0) {
         yy := yy-1;
         r := r+xx;
    }
    xx := xx*2;
    yy := yy/2;
}
```