Why3: Verificação de programas imperativos - Documento original

Simão Cunha

5 de dezembro de 2022

Recorda o algoritmo imperativo *insertion sort* escrito em C:

```
1  void insertionSort(int A[], int N) {
2      int i, j, key;
3      for (j=1 ; j<N ; j++) {
4          key = A[j];
5          i = j-1;
6          while (i>=0 && A[i] > key) {
7              A[i+1] = A[i];
8              i--;
9          }
10     A[i+1] = key;
11     }
12 }
```

Why3 permite a verificação dos tais algoritmos imperativos, abstraindo-se de detalhes a nível programacional, como tipos de dados ou alocação de memória. WhyML contém funcionalidades imperativas, incluindo variáveis mutáveis e arrays.

A biblioteca de array Why3, disponível em `http://why3.lri.fr/stdlib/array.html`, contém, em particular, os módulos `array.IntArraySorted` e `array.ArrayPermut` que tanto definem as noções relevantes para a especificação de noção de ordenação. como também contém lemas que facilitam provas automáticas.

É importante relembrar a dicotomia entre duas diferentes abordagens para deduzir a verificação de programas:

- De um lado encontrámos verificadores que sinalizam linguagens de programação do mundo-real. Os exemplos incluem Frama-C/WP, Verifast (para programas C e Java), KeY (para Java), ou SPARK;
- Por outro lado, ferramentas como Why3 e ferramentas da Microsoft como Dafny/Boogie oferecem as suas próprias linguagens de programação, sinalizando a verificação ao nível algorítmico. Why3 não é uma linguagem de programação com um propósito genérico: foi desenhado para a verificação ao nível algoritmico do que a nível programacional.

  Os programas de verificação podem ser obtidos com ferramentas mais recentes, como:

  - por extração automática
    - Why3 oferece a geração automática de código C ou OCaml de desenvolvimentos (verificados) de WhyML

– ao codificar programas na linguagem do verificador, junto com o modo de memória
  * Boogie foi feito especificamente para ser usado desta forma como um verificador intermédio;
  * O conjunto de ferramentas SPARK toolset usa Why3 desta forma;
  * FrontEnds Micro-C e Python também estão disponíveis em Why3

De forma a verificar o algoritmo com Why3, criamos um módulo que começa por importar os módulos de bibliotecas necessários e escrevemos o algoritmo em WhyML: a linguagem de programação do Why3. WhyML pertence à família de linguagens ML (`https://en.wikipedia.org/wiki/ML_(programming_language)`).

```
1  module InsertionSort
2      use int.Int
3      use ref.Ref
4      use array.Array
5      use array.IntArraySorted
6      use array.ArrayPermut
7      let insertion_sort (a: array int)
8          ensures { sorted a }
9          ensures { permut_all (old a) a }
10     =
11         . . .
12 end
```

Iremos escrever o algoritmo completo seguindo passo a passo a versão escrita em C, mas iremos adicionar anotações que são necessárias para provar a sua correção: invariantes de ciclo e variantes (permalink: `https://why3.lri.fr/try/?name=test.mlw&lang=whyml`).

```
let insertion_sort (a: array int)
  ensures { sorted a }
  ensures { permut_all (old a) a }
=
  for j = 1 to length a - 1 do
    invariant { sorted_sub a 0 j }
    invariant { permut_all (old a) a }
    let key = a[j] in
    let i = ref (j-1) in

    while !i >= 0 && a[!i] > key do
        invariant { -1 <= !i <= j-1 }
        invariant { sorted_sub a 0 j }
        invariant { !i = j-1 \/ (a[j-1] <= a[j] /\ key < a[!i+2]) }
        invariant { permut_all (old a) a[!i+1 <- key] }
        variant { !i }
        a[!i+1] <- a[!i];
        i := !i - 1
    done;
    a[!i+1] <- key
  done
```

Fazemos as seguintes observações sobre a linguagem de programação:

- Tal como em muitas linguagens (mas não em C), os arrays contêm informação do comprimento: `length a` refere-se ao tamanho do array `a`. Os acessos ao array são válidos desde que sejam dentro do seu tamanho;
- Os ciclos `for` são iterações com fronteiras (tal como em Python). Logo, não existe a necessidade de dar variantes para estabelecer a terminação, ou para incluir invariantes triviais que se referem a variáveis de controlo (`j` no exemplo abaixo);
- A distinção é feita entre variáveis normais como `key` e `j` abaixo (immutáveis, tal como nas linguagens de programação funcionais) e com referências, que oferecem mutabilidade. As variáveis de controlo de ciclo `for` não são referências e não podem ser atribuídas;
- A instrução `let i = ref (j-1) in ...` cria a referência `i` e inicializa o seu conteúdo com o valor atual da expressão `j-1`. De forma a aceder os conteúdos da referência, é usado o símbolo `!`;
- São usados 3 tipos de operadores de atribuição:
  - `=` é mais uma ligação do que propriamente uma atribuição. É usado para variáveis imutáveis e também para inicializar referências (note-se que a referência da variável em si é imutável; tal como um apontador em C, o seu conteúdo pode ser modificado);
  - `:=` é a instrução de atribuição de referência. Pense em `i := e` como a instrução `!i = e`, semelhante a `*i = e` em C. Além disso, `i := !i - 1` incrementa o valor de `i`;
  - `←` é o operador de atribuição de array: a instrução `a[!i+1] <- key` armazena o valor de `key` na posição `!i+1` no array `a`
- E quanto à especificação e anotações:
  - O predicado `sorted` refere-se a todo o array;

- `sorted_sub a x y` significa que o *range* entre os índices `x` e `y-1` está ordenado;
- A expressão `old a` refere-se ao array `a` no seu estado inicial;
- `a[k <- e]` refere-se à atualização do array `a` com a atribuição de `e` ao valor do índice `k`. Esta notação é usada no exemplo abaixo para expressar o invariante de ciclo de acordo com a propriedade de permutação: tal como está, o array atual não é uma permutação do array inicial porque não contém o elemento `key`; o invariante menciona que o array obteve-o ao escrevê-lo de volta