Funções e Especificações

Para demonstrar que uma função f verifica a sua especificação R é necessário:

- definir $f : A \longrightarrow B$
- provar que $\forall\ a : A,\ R\ a\ f(a)$ se verifica

Uma axiomatização da função factorial

```
/*@ axiomatic factorial {
  @
  @ predicate isfact(integer n, integer r);
  @ axiom isfact0:
  @    isfact(0,1);
  @ axiom isfactn:
  @    \forall integer n, integer f;
  @            n>0 ==> isfact (n-1,f) ==> isfact(n,f*n);
  @
  @ logic integer fact (integer n);
  @ axiom fact1:
  @    \forall integer n; isfact (n,fact(n));
  @ axiom fact2:
  @    \forall integer n, integer f;
  @            isfact (n,f) ==> f==fact(n);
  @ }
 @*/
```

Tabulação de factoriais

As anotações de pré e pós condições são:

```
/*@ requires
  @   size >=0 &&
  @   \valid_range(inp,0,size-1) &&
  @   \valid_range(outp,0,size-1) &&
  @   \forall int a; 0<=a<size ==> inp[a] >= 0;
  @ assigns outp[0..size-1];
  @ ensures
  @   \forall int a;
  @      0<=a<size ==> outp[a] == fact (inp[a]);
  @*/
```

Tabulação de factoriais

```
void factab (int inp[], int outp[], int size)
{
  int k = 0 ;
/*@ loop invariant 0<=k<=size &&
  @   \forall int a; 0<=a<k ==> outp[a] == fact (inp[a]);
  @ loop variant size-k;
  @*/
  while (k < size) {
    int f = 1, i = 1, n=inp[k];
    /*@ loop invariant 1<=i<=n+1 && f == fact(i-1);
      @ loop variant n+1-i;
      @*/
    while (i <= n) {
      f *= i;
      i += 1;
    }
    outp[k++]=f;
  }
}
```

Tabulação de factoriais

- O uso do tipo lógico `integer`, em vez do tipo `int` do C, torna a axiomatização mais abstracta:
  - podemos usar a axiomatização com qualquer tipo inteiro concreto usado no programa.
  - à função lógica `fact` que ocorre na pós-condição e invariantes, são passados argumentos do tipo `int`.
- O uso de condições `frame` é ilustrado pela cláusula:

$$\texttt{assigns outp}[0..\texttt{size} - 1];$$

- A presença de uma cláusula `assigns`, leva à geração de condições de verificação específicas que garantem que a função não altera o valor de posições de memória não listadas na cláusula.

Chamada de funções

- Vamos considerar uma versão da função de tabelação de factoriais, que usa uma funcão auxiliar `factf`.
- Ilustra a modularidade permitida pela utilização de contractos na definição dos programas.

  - É dado um protótipo da função `factf`, com um contracto
  - Não queremos saber qual a implementação de `factf`
  - Verificamos a correcção de `factab` independentemente da implementação de `factf`.

- Consideramos para `factf`:

```
/*@ requires n >= 0;
  @ assigns \empty;
  @ ensures \result == fact(n);
  @*/
int factf (int n);
```

Chamada de funções - `factab`

```
#define LENGTH 1000
int inp[LENGHT], outp[LENGHT];

/*@ requires
  @   0 <= size <= LENGHT &&
  @   \forall int a; 0<=a<size ==> inp[a] >= 0;
  @ assigns outp[0..size-1];
  @ ensures
  @   \forall int a;
  @      0<=a<size ==> outp[a] == fact (inp[a]);
  @*/
void factab (int size) {
  int k = 0 ;
  /*@ loop invariant 0<=k<=size &&
    @   \forall int a; 0<=a<k ==> outp[a] == fact (inp[a]);
    @ loop variant size-k;
    @*/
  while (k < size) {
    outp[k++]=factf(inp[k]);
  }
}
```

Chamada de funções - `factab`

- O uso de `\result` na pós-condição refere o valor de retorno na função
- A cláusula `assigns \empty` garante que nenhuma posição de memória visível externamente é alterada pela função.
- Note-se que esta versão de `factab` funciona sobre arrays de input e output globais, sobre os quais é conhecido o tamanho.

- É suficiente garantir que o tamanho (`size`) considerado pela função `factab` não é maior do que `LENGHT` - o tamanho alocado para os arrays.

ACSL- Axiomatizações

Definem tipos, funções e predicados declarando os respectivos axiomas.

```
/*@ axiomatic IntList {
@ type int_list;
@ logic int_list nil;
@ logic int_list cons(integer n,int_list l);
@ logic int_list append(int_list l1,int_list l2);
@ axiom append_nil:
@         \forall  int_list l; append(nil,l) == l;
@ axiom append_cons:
@        \forall integer n; int_list l1,l2;
@          append(cons(n,l1),l2) == cons(n,append(l1,l2));
@ }
@*/
```

A consistência tem de ser assegurada pelo utilizador.

ACSL - definições recursivas

```
/*@ logic integer max_index{L} (int t[],integer n)=
@ (n==0)? 0: (t[n-1]==0) ? n: max_index(t,n-1);
@*/
```

Retorna o maior indice $0 \leq 1 \leq n-1$ tal que $t[i] = 0$

ACSL- Construções de ordem superior

$$\backslash lambda\ \tau_1\, x_1, \ldots \tau_n\, x_n\, ;\, t$$

$f(k)$ inteira ou real (ou booleana) *kinteiro*

$$\begin{aligned}
\backslash max(i,j,f) &= max\{f(i),\ldots,f(j)\} \\
\backslash min(i,j,f) &= min\{f(i),\ldots,f(j)\} \\
\backslash sum(i,j,f) &= f(i)+\cdots+f(j)\} \\
\backslash product(i,j,f) &= f(i)\times\cdots\times f(j)\} \\
\backslash numof(i,j,f) &= \backslash sum(i,j,\backslash lambda\ integer\, k\, ;\, f(k)?1:0)
\end{aligned}$$

ACSL- Construções de ordem superior

Função para somar n números em precisão dupla:

```
/*@ requires n >= 0 && \valid(t+(0..n-1)) ;
@ ensures \result == \sum(0,n-1,\lambda int k; t[k]);
@*/
double array_sum(double t[],int n) {
int i;
double s = 0.0;
/*@ loop invariant 0 <= i <= n;
@ loop invariant s == \sum(0,i-1,\lambda int k; t[k]);
@ loop variant n-i;
*/
for(i=0; i < n; i++) s += t[i];
return s;
}
```

Predicados e funções lógicas

É possivel definir predicados e funções lógicas. Os predicados podem ser indutivos. Por exemplo, dada uma lista ligada `typedef struct _list { int element; struct _list* next; } list;`

Podemos definir propriedades, por exemplo de acessibilidade (desde da raiz por uma sucessão de campos `next`:

```
/*@
inductive reachable{L} (list* root, list* node) {
case root_reachable: \forall list* root; reachable(root,root);
case next_reachable: \forall list* root, *node;
\valid(root) ==> reachable(root->next, node) ==>
reachable(root,node);
}
*/
```

L representa o estado da memória mas é inferido, normalmente...

Predicados e funções lógicas

O predicado anterior pode ser usado para definir se uma lista é finita ou não. Se for finita chega a `\null`

```
/*@ predicate finite{L}(list* root) = reachable(root,\null); */
```

Invariante de tipo Algo que se verifica para um dado tipo

```
/*@ type invariant finite_list(list* root) = finite(root); @/
```

Pode tambem haver invariantes globais: válidos antes e depois de chamadas a funções.

Invariantes de dados

Invariantes de tipo : Permitem indicar que todos os elementos duma dada estrutura de dados verificam um invariante.

```
/*@ type invariant finite_list(list* root)
                 = finite(root);
 */
```

Invariantes globais: propriedades de variáveis globais

```
int a;
//@ global invariant a_is_positive: a >= 0 ;
```

Podem ainda ser

fortes

fracos podem ser temporariamente não satisfeitos, mas têm de o ser à entrada e à saída.

Axiomatizações

Podemos também definir o comprimento duma lista (finita!).

```
/*@ axiomatic Length {
logic integer length{L}(list* l);

axiom length_nil{L}: length(\null) == 0;

axiom length_cons{L}:
     \forall list* l, integer n; finite(l) && \valid(l) ==>
           length(l) == length(l->next) + 1;
}
*/
```

Máximo duma lista

```
/*@
@ requires \valid(root);
@ assigns \nothing;
@ terminates finite(root);
```

```
@ ensures
@ \forall list* l; \valid(l) && reachable(root,l)
@               ==> \result >= l->element;
@ ensures \exists list* l; \valid(l) && reachable(root,l) &&
@           \result == l->element;
*/
int max_list(list* root);
```

Máximo duma lista

```
int max_list(list* root) {
int max = root->element;
while(root->next) {
root = root -> next;
if (root ->element > max) max = root->element;
}
return max;
}
```

Comportamentos - ordenação

```
/*@ predicate Sorted{L}(int t[],integer i, integer j)=
  @   \forall int k; i <= k < j ==>
  @    \at(t[k],L) <= \at(t[k+1],L);
  @*/

/*@ requires N >= 1 && \valid_range(A,1,N);
  @ assigns A[1..N];
  @ ensures Sorted{Here}(A,1,N);
  @*/

void insertion_sort(int A[], int N) {
  int i, j, key;

  /*@ loop invariant
    @   2 <= j <= N+1 && Sorted{Here} (A,1,j-1);
    @ loop variant (N-j);
    @*/

  for (j=2; j<= N; j++){
    key = A[j];
    i = j-1;
    /*@ loop invariant
      @ ...
      @ loop variant
```

```
      @ ...
      @*/
  while (i>0 && A[i]>key) {
    A[i+1] = A[i];
    i--;
  }
}
```

Comportamentos e Labels

- O contracto da função definida deve ser poder ser utilizado em qualquer algoritmo de ordenação, uma vez que apenas descreve o que o algoritmo faz.
- Por outro lado, os invariantes de ciclo descrevem como o algoritmo funciona, portanto são específicos de cada algoritmo.
- O predicado `Sorted` recebe como argumento um array e dois índices, que definem os limites do array entre os quais o algoritmo está ordenado
- O predicado também recebe uma label `L`, com o significado de que o programa está ordenado num determinado estado
- O operador `\at` define o valor de uma expressão num determinado estado.

Comportamentos e Labels

- O uso de labels permite referir dentro da mesma asserção ao valor de uma expressão em dois estados diferentes.
- Duas labels especiais:
  - `Here`: denota o estado actual;
  - `Old`: denota o pré-estado da função.
- A especificação do algoritmo está incompleta: não específica a propriedade de que um algoritmo de ordenação deve preservar o multiconjunto dos elementos.
- Podemos considerar a seguinte formalização:

$$\begin{array}{l} \forall k.p \leq k \leq r \rightarrow (\exists l.p \leq l \leq \rightarrow A[k] = B[l]) \quad \wedge \\ \forall k.p \leq k \leq r \rightarrow (\exists l.p \leq l \leq \rightarrow B[k] = A[l]) \end{array}$$

- Podemos tratar permutações como sequências de trocas duas a duas.

Axiomática de permutações

```
/*@  predicate Swap{L1,L2}(int a[],integer i, integer j)=
  @    \at(a[i],L1) == \at(a[j],L2) &&
  @    \at(a[j],L1) == \at(a[i],L2) &&
  @    \forall int k; k!=i && k!=j ==>
  @    \at(a[k],L1) == \at(a[k],L2);
  @
  @  axiomatic permutation {
  @
  @  predicate Permuta{L1,L2} (int a[], integer l, integer h);
  @
  @  axiom Permuta_refl{L}:
  @    \forall int a[], interger l, h; Permuta{L,L} (a,l,h);
  @  axiom Permuta_sim{L1,L2}:
  @    \forall int a[], interger l, h;
  @    Permuta{L1,L2} (a,l,h) ==>  Permuta{L2,L1} (a,l,h);
  @  axiom Permuta_trans{L1,L2,L3}:
  @    \forall int a[], interger l, h;
  @    Permuta{L1,L2} (a,l,h) && Permuta{L2,L3} (a,l,h) ==>
  @    Permuta{L1,L3} (a,l,h);
  @  axiom Permuta_swap{L1,L2}:
  @    \forall int a[], interger l, h, i, j;
  @    l <= i <= h && l <= j <= h && Swap{L1,L2} (a,i,j) ==>
  @    Permuta{L1,L2} (a,l,h);
  @  }
```

Axiomatização de permutações

- `Swap {L1,L2}(a,i,j)`, tem o significado de que o conteúdo de `a` nos estados `L1` e `L2` é mesmo excepto para os índices `i` e `j`, para os quais estão trocados
- `Permuta {L1,L2}(a,l,h)`, tem o significado de que o conteúdo de `a` entre os índices `l` a `h` no estado `L2` é uma permutação do seu conteúdo no estado `L1`.
- `Permuta_swap`, tem o significado de que uma troca entre dois índices válidos é uma permutação (corresponde à permutação mais elementar).
- Definimos agora a especificação de um algoritmo de ordenação,
- Estruturamos os contractos num conjunto de comportamentos (assinalados com `behavior`).

Especificação da ordenação

```
/*@  requires 0 <= p <= r && \valid_range(A,p,r);
  @  assigns A[p..r];
  @  behavior sorted:
  @    ensures
  @      Sorted{Here}(A,p,r);
  @  behavior permutation
  @    ensures
  @      Permuta{Old,Here}(A,p,r);
  @*/
void sort(int A[]), int p, int r);
```