Verification of C programs ACSL and Frama-c

ACSL

ANSI-C specification language.

Inspired by, but different from, JML (C $\neq$ Java!)

- object-oriented features absent (inheritance...)
- C has no support for memory safety or exceptions (for instance two C arrays may *overlap*!)
- Dynamic checking in C is hard to implement. ACSL is tailored for static checking and deductive verification

Maximum of an *Array*

```
int size, u[], max;
```

```
/*@ requires size >= 1;
  @ ensures 0 <= max < size &&
  @ (\forall int a; 0<=a<size ==> u[a]<=u[max]);
  @*/
```

Something missing?

```
void maxarray() {
  int i = 1;
  max = 0;

  /*@ loop invariant
    @ 1<=i<=size && 0<=max<i &&
    @   (\forall int a; 0<=a<i ==> u[a]<=u[max]);
    @ loop variant size-i;
    @*/
  while (i < size) {
    if (u[i] > u[max]) max = i;
    i = i+1;
  }
}
```

Safety-aware Version

```
int size, u[], max;

/*@ requires size >= 1
  @           && \valid_range(u,0,size-1);
  @ ensures 0 <= max < size &&
  @ (\forall int a; 0<=a<size ==> u[a]<=u[max]);
  @*/
```

Factorial: Axiomatization

```
/*@ 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);
  @} */
```

Factorial: Tabulation (spec)

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

Factorial: Tabulation (altern.)

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


/*@ requires 0<=size<=LENGTH &&
  @    \forall int a; 0<=a<size ==> inp[a] >= 0;
  @ ensures
  @    \forall int a;
  @       0<=a<size ==> outp[a] == fact (inp[a]);
  @*/
```

```
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++;
    }
    outp[k++] = f ;
  }
}
```

Factorial: Function

```
/*@ requires n >= 0;
  @ ensures \result == fact(n);
  @*/
int factf (int n)
{
  int f = 1, i = 1 ;

  /*@ loop invariant 1<=i<=n+1 && f == fact(i-1);
    @ loop variant n+1-i;
    @*/
  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```

Contracts and Modularity!

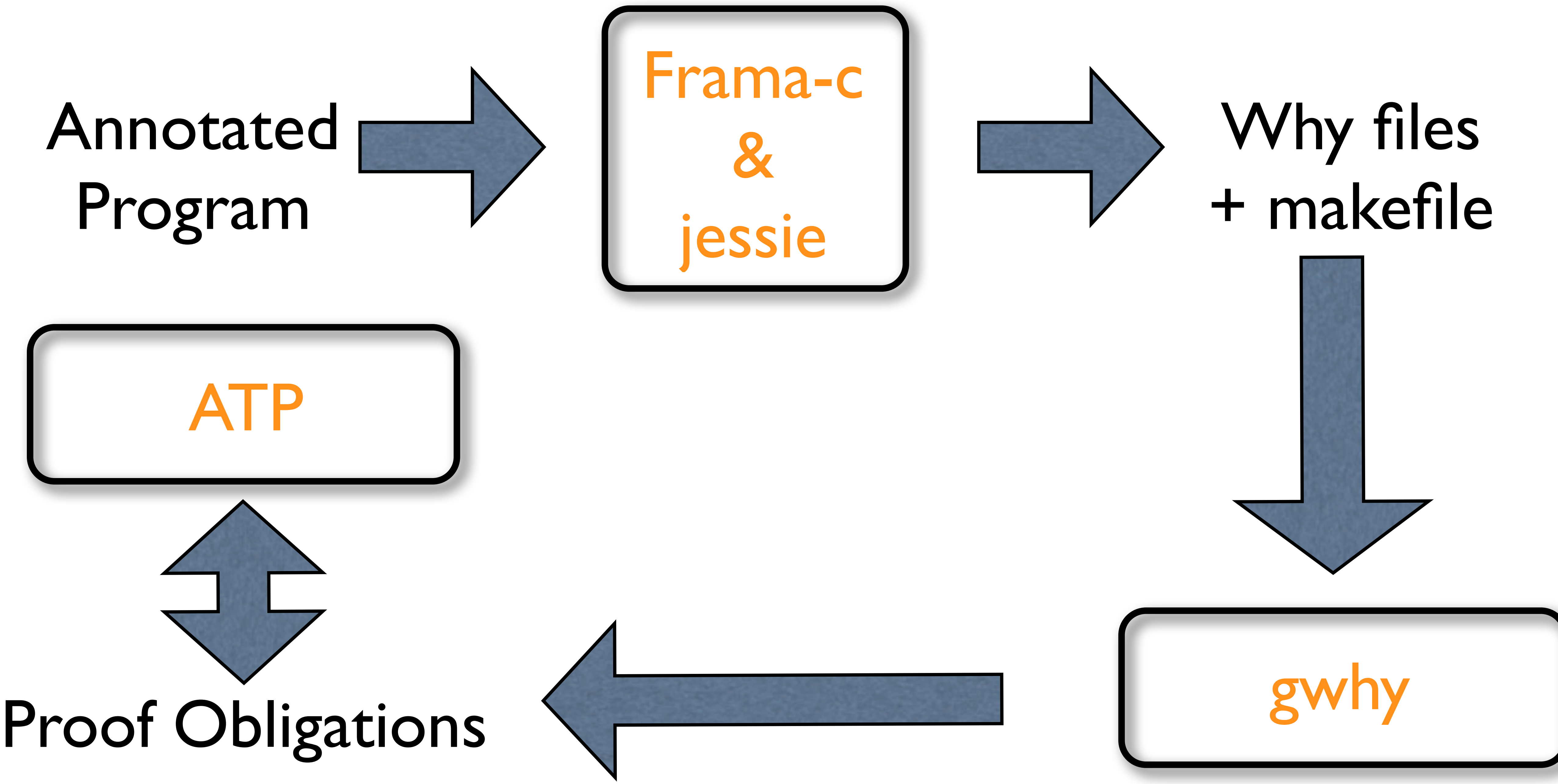
```
void factab (int size)
{
  int k = 0 ;

  /*@ 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]) ;
    k++;
  }
}
```

Frama-c

- A multi-purpose tool for the analysis of C programs, joint effort of CEA and INRIA
- Includes PV module (VCGen) based on the *Caduceus* tool, developed at LRI
- Multi-prover; initially meant for the *Coq* proof assistant
- Builds on a more general verification tool called *Why*, also from LRI

Frama-c with ATP



Exercise 1

```
void swap(int X[], int a, int b)
{ aux = X[a]; X[a] = X[b]; X[b] = aux; }
```

1. Write specification
2. Prove correctness of function

Exercise 2

Recall the *partition* function used by the quicksort algorithm.   Verify informally:

1. Write a Specification
2. Examine suggested implementation
3. Identify loop invariant
4. Check initial conditions and preservation
5. Identify loop variant
6. Check final conditions

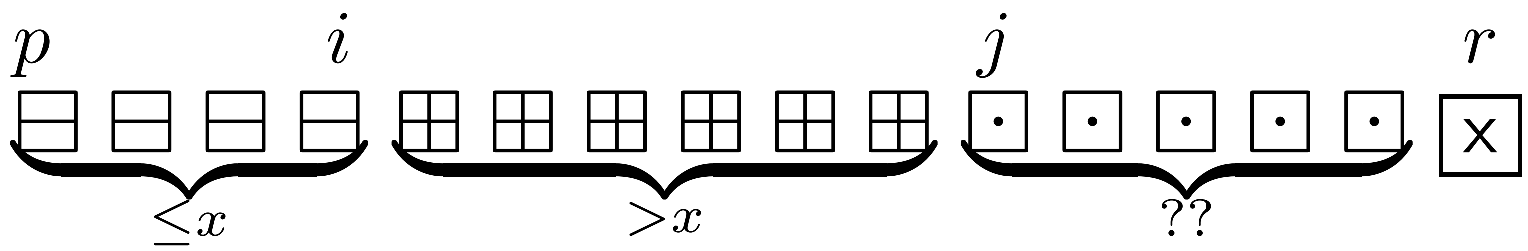
Exercise 2

```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}
```

Análise de Correcção – Invariante

No início de cada iteração do ciclo `for` tem-se para qualquer posição k do vector:

1. Se $p \leq k \leq i$ então $A[k] \leq x$;
2. Se $i + 1 \leq k \leq j - 1$ então $A[k] > x$;
3. Se $k = r$ então $A[k] = x$.



⇒ Verificar as propriedades de *inicialização* ($j = p$, $i = p - 1$), *preservação*, e *terminação* ($j = r$)

⇒ o que fazem as duas últimas instruções?

Something Missing!

- It is still required to check that the elements are the same in the input and in the output arrays!
- A particular case of the problem of specifying that two arrays contain the same elements
- *And* same number of occurences: *multiset equality*, rather than *set equality*

A first attempt

$$\forall k : p \leq k \leq r : (\,\exists l : p \leq l \leq r : A[k] = B[l] \wedge A[l] = B[k]\,)$$

What's wrong with it?

Second attempt

$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : A[k] = B[l] \,)$$
$$\wedge$$
$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : B[k] = A[l] \,)$$

What's wrong with it?

Third attempt

Define a notion of permutation

```
inductive Permut{L1,L2}(int a[], integer l, integer h) {
    case Permut_refl{L}:
     \forall int a[], integer l, h; Permut{L,L}(a, l, h) ;
    case Permut_sym{L1,L2}:
      \forall int a[], integer l, h;
        Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
    case Permut_trans{L1,L2,L3}:
      \forall int a[], integer l, h;
        Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
          Permut{L1,L3}(a, l, h) ;
    case Permut_swap{L1,L2}:
      \forall int a[], integer l, h, i, j;
         l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j)
           ==> Permut{L1,L2}(a, l, h) ;
```