Multi-Prover Verification of C Programs

Jean-Christophe Filliâtre, Claude Marché

Université Paris Sud

Introduction and Motivations

We propose

- a new language for specifying C programs (annotations into comments, à la JML)
- a new approach for the verification of such annotated programs
- an implementation: the Caduceus tool

Caduceus is currently under experimentation on industrial embedded programs:

- Axalto company (smart cards)
- Dassault Aviation company (avionics)

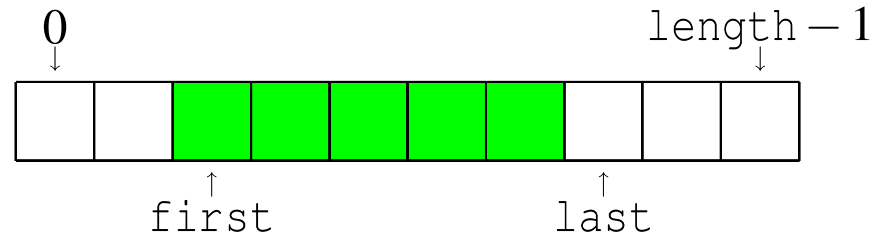
Overview of the talk

1. Usage of our tool illustrated on examples
   (a) a simple character queue
   (b) in-place linked-list reversal
   (c) the Schorr-Waite graph marking algorithm

2. Underlying verification technique

3. Limitations, perspectives

Example: character queue as circular array




```
struct queue {
  char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Example continued: specifying functions

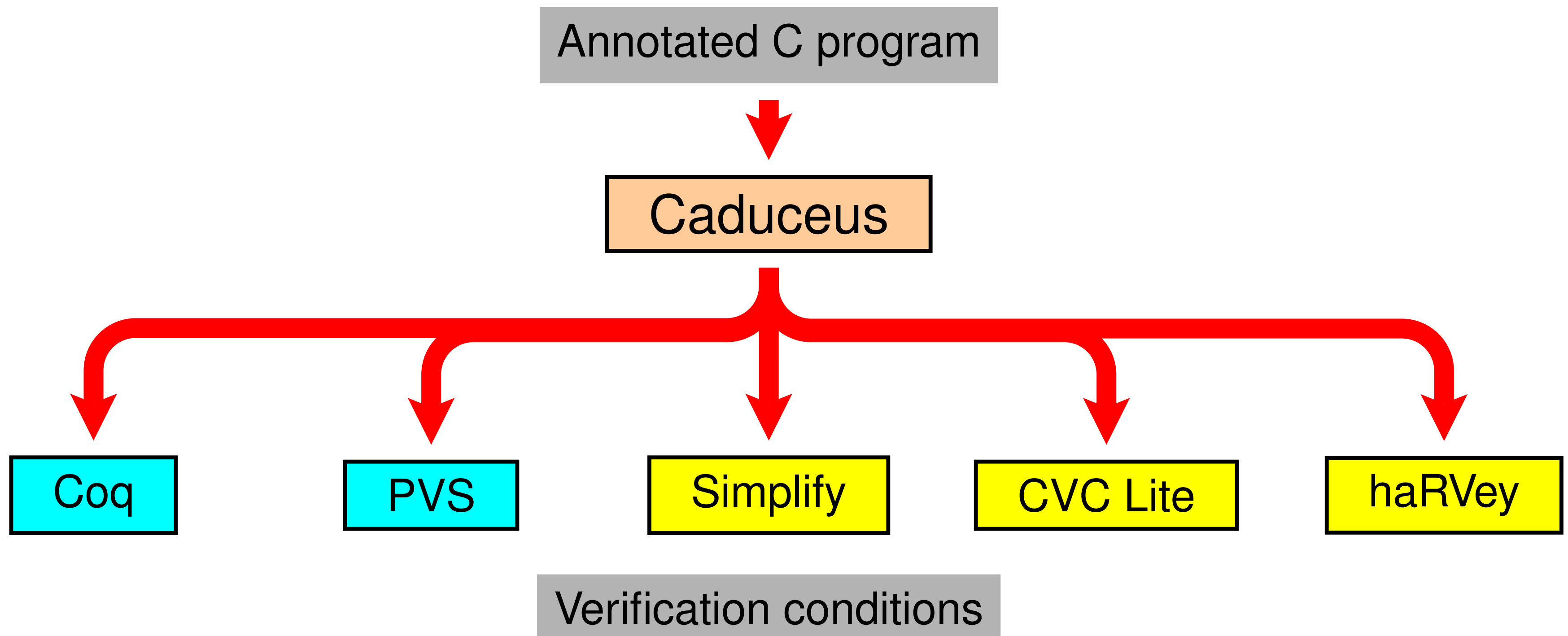
```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c);

/*@ requires !q.empty
  @ assigns q.empty, q.full, q.first
  @ ensures !q.full && \result == q.contents[\old(q.first)]
  @*/
char pop();
```

Example continued: body for `push` function

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c) {
  q.contents[q.last++] = c;     // insert 'c' in the queue
  if (q.last == q.length)
        q.last = 0;             // wrap if needed
  q.empty = 0;                  // queue is not empty
  q.full = (q.first == q.last); // queue is full if
                                // 'last' reaches 'first'
}
```

Multi-prover architecture



Example continued: certification of `push` function

Caduceus produces 3 verification conditions expressing that

- the code of `push` contains no unallocated pointer dereference (e.g. assignment of `q.contents[q.last++]` is valid).
- the postcondition and the `assigns` clause of `push` are established
- the invariant `q_invariant` is preserved by `push`

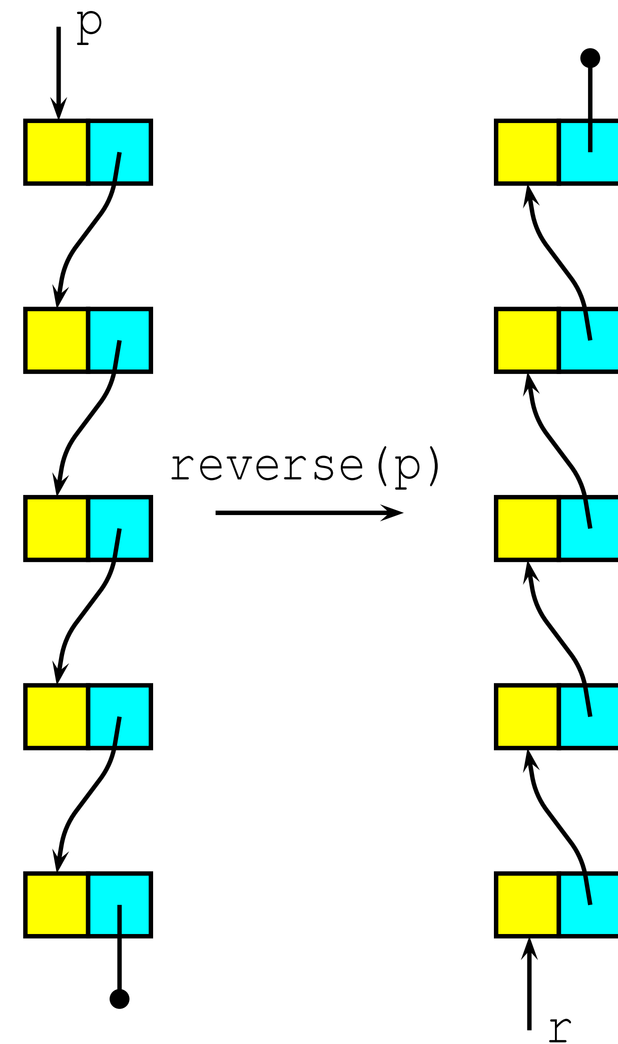
Proofs of these obligations

- with Simplify (100%) and CVC Lite (67%)
- with Coq (100%), very easy (6 lines of tactics)

Example: in-place list reversal

```
typedef struct struct_list {
  int hd;
  struct struct_list *tl;
} *list;

list reverse(list p) {
  list r = NULL;
  while (p != NULL) {
    list q = p ;
    p = p->tl;
    q->tl = r;
    r = q;
  }
  return r;
}
```



Introduction of new logical types and functions

- Assertions are first-order logic formulas with a C-like syntax
- New predicates and functions can be introduced

```
// logical finite list of pointers
//@ logic plist nil()
//@ logic plist cons(list p, plist l)
```

```
// concatenation and reversal
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- Axioms may be given, e.g.

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```

Specification of list reversal

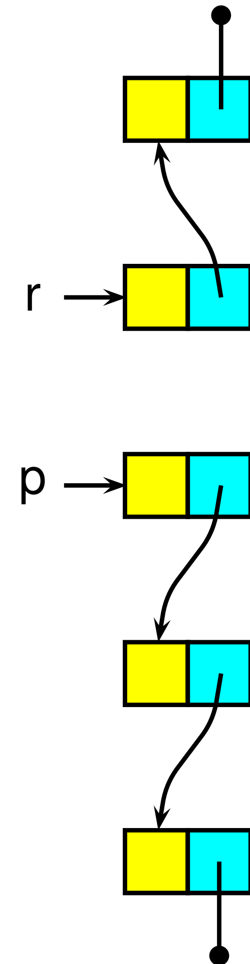
```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }
```

```
/*@ requires is_list(p)
  @ ensures \forall plist l;
  @     \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Annotating the code of list reversal

```
list reverse(list p) {
  list r = NULL;
/*@ invariant
    \exists plist lp; \exists plist lr;
      llist(p, lp) && llist(r, lr) &&
      disjoint(lp, lr) &&
      \forall plist l; \old(llist(p, l)) =>
        app(rev(lp), lr) == rev(l)
   @ variant length(p) for length_order */
  while (p != NULL) {
    list q = p;
    p = p->tl; q->tl = r; r = q;
  }
  return r;
}
```



Certification of list reversal

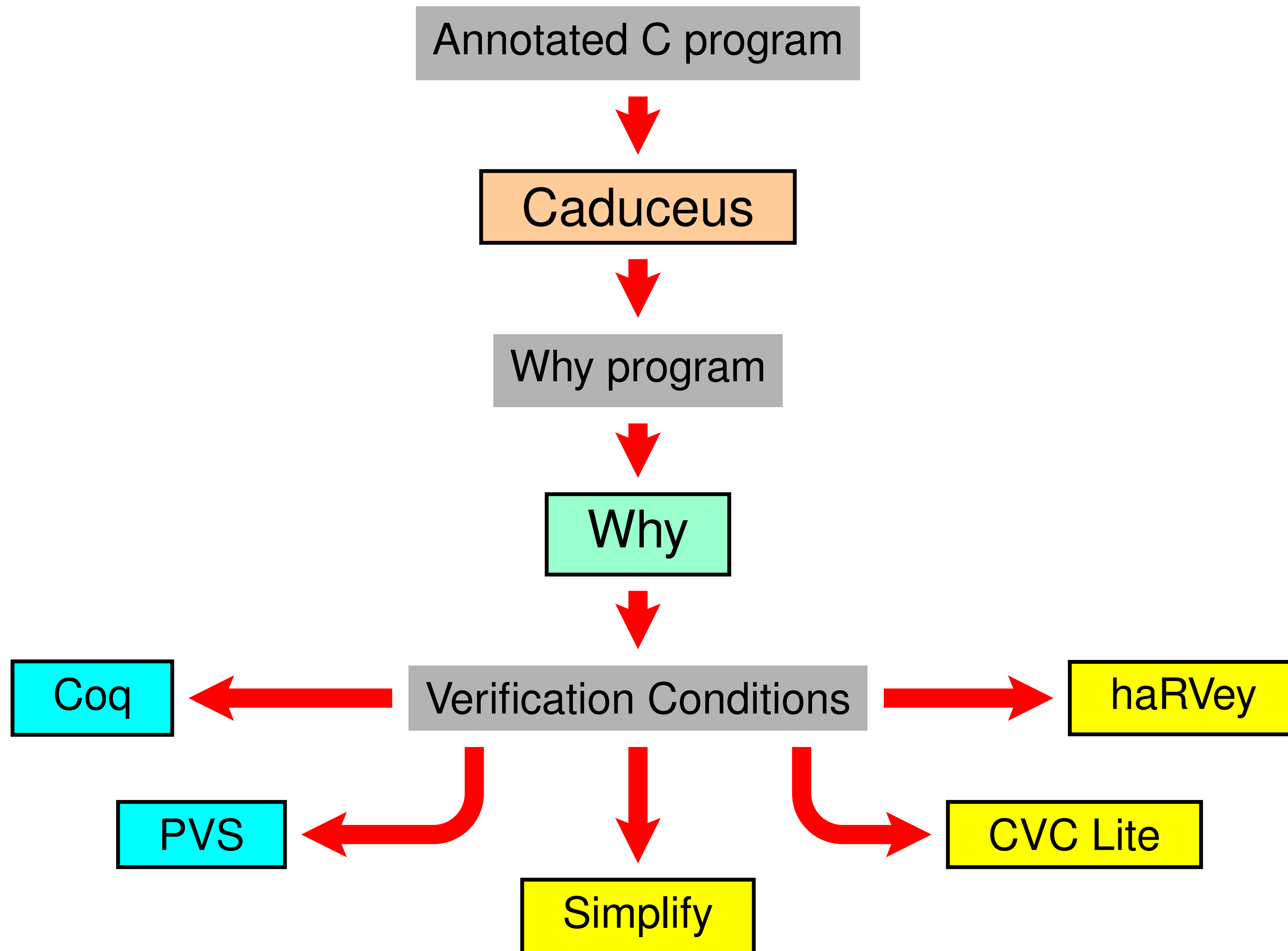
- 7 verification conditions
- With Simplify: 71%
- With Coq: 100%, with 661 lines of tactics

Example: Schorr-Waite algorithm

- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

Underlying Technique

Intermediate language: Why



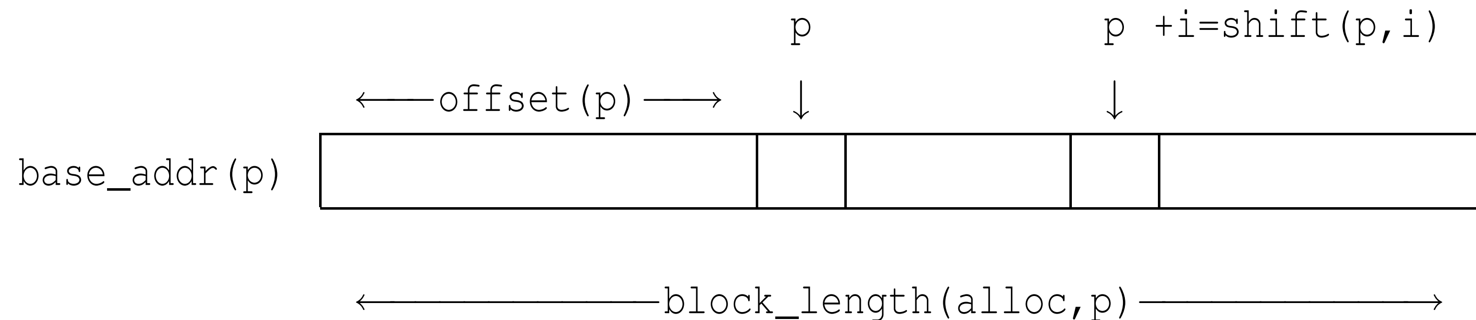
The Why tool

- Language *dedicated* to program certification
  - purely functional types + *references* on such types
  - Hoare-like annotations (pre/post-conditions, loop invariants/variants)
  - explicit side effects: read/assigned references, exceptions raised
  - aliases are forbidden
- Annotations are written in first-order predicate logic

  Adaptation to a new prover only requires a new *pretty-printer*
- Allows abstract data types and functions

Modeling C memory heap

- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



- Each structure field is a map from addresses to memory blocks

Burstall-Bornat model: example of character queue

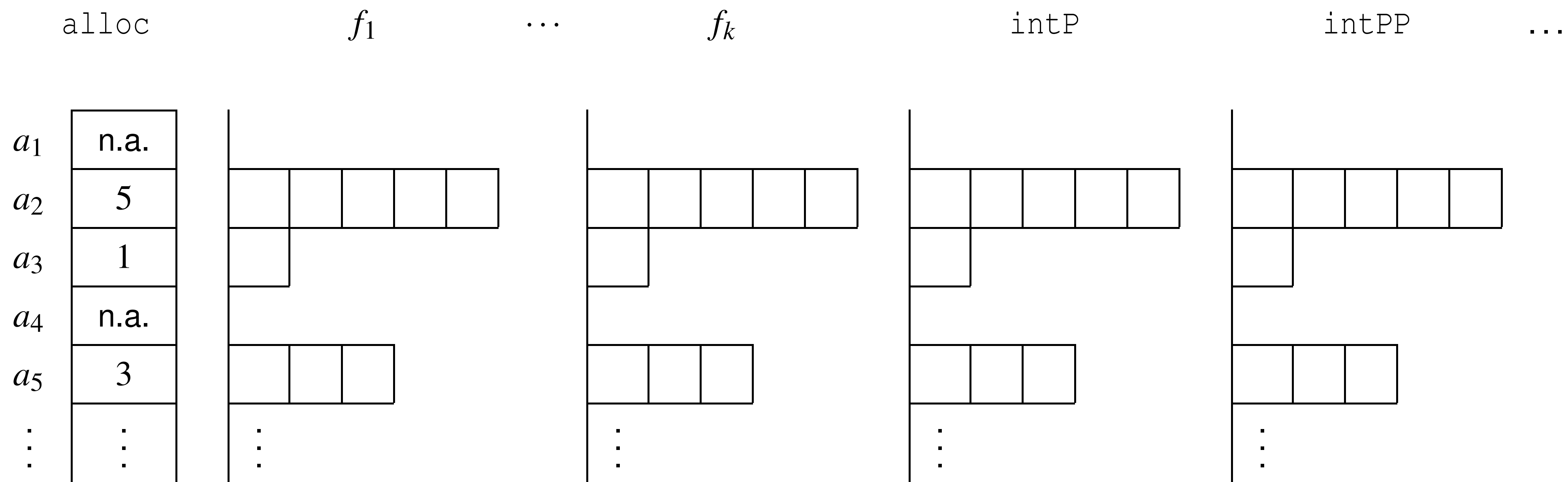
adresses *structure fields* *arrays of* `int`


```
q.contents[q.last++] = c

q.last <- 4 + 1

*(q.contents+4) <- c
```

General structure of C memory heap



Translation of C statements into Why

The C statement

```
q.contents[q.last++] = c
```

becomes in Why:

```
assert valid(alloc,q);          // proof obligation
let tmp1 = acc(last,q) in       // tmp1 <- q.last
upd(last,q,tmp1+1);             // q.last <- tmp1+1
let tmp2 = shift(acc(contents,q),tmp1) in
                                // tmp2 <- q.contents + tmp1
assert valid(alloc,tmp2);       // proof obligation
upd(intP,tmp2,c)                // *tmp2 <- c
```

Axiomatization

- The abstract Why functions `acc`, `upd`, `shift`, etc. are specified by axioms: the *background theory*.
- Excerpt from this theory:

```
acc(upd(t,i,v),i) = v
i <> j -> acc(upd(t,i,v),j) = acc(t,j)
shift(p,0) = p
shift(shift(p,i),j) = shift(p,i+j)
...
```

- An important part of this theory is dedicated to assigns clauses

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C
- Prototype freely available http://why.lri.fr/caduceus (Demo on demand)

But scaling up issues show up on large programs:

- Generated proof obligations can get large
- Clear need for assistance to write specifications
- Need for more automation of proofs, cooperation of provers

Limitations

Current limitations of the tool

- Non-terminating recursive functions
- Initialization of global variables not satisfying invariants
- Arithmetic overflow
- Use of `&` operator on stack-allocated variables
- Dynamic memory allocation
- `union`

Limitations of the model:

- pointer cast
- non ANSI features (i.e. compiler dependent)

About soundness of the approach

1. Why method
   - Why's verification condition generator has been proved
     [Filliâtre, 2003, J. of Functional Programming]
   - A *validation* is produced: a certificate, checkable by Coq
2. Caduceus translation from C to Why
   - Soundness of the translation not formally proved but
   - The modeling is simple enough to be trusted
     its limitations are clearly exposed
   - Ultimately, consistency of axioms can be proved by a Coq realization