Preuve de code bas niveau avec J^3

Journée ProofInUse

Guillaume Cluzel Raphaël Rieu-Helft

TrustInSoft

21 Novembre 2022

TRUST IN SOFT

Objectif du projet

Rendre la preuve de code C bas niveau accessible à des développeurs non experts des méthodes formelles.
Pour ça, on veut être capable de diagnostiquer la raison pour laquelle un but est non prouvé :

- Les spécifications sont fausses.
- Certaines fonctions/boucles sont sous-spécifiées.

Support de constructions bas niveau :

- Casts de pointeurs
- Aliasing
- Unions
- Pointeurs mal alignés
- Bitfields
- Addresses absolues

Utilisation des bitvecteurs

Le code bas-niveau fait beaucoup appel aux opérateurs `&`, `|`, `^` et `~`.

- Mal supporté par la théorie des entiers mathématiques.
- Support natif dans la théorie des bitvecteurs.

Mais aussi : extraction et recollement de bytes nécessaire aux casts de pointeurs.
Démo : `abs.i`.

Contraintes sur les types grâce aux bitvecteurs

Les bitvecteurs contiennent nativement des contraintes de types sur leur taille.

```
struct S {
    unsigned int x : 4;
    unsigned int y : 3;
};

/*@ ensures \result != 2; */
int f(struct S s) {
    s.y += 84;
    s.x -= 56;
    return s.x + s.y;
}
```

Avec des entiers mathématiques :

```
CVC5 1.0.0 (Timeout):
  Values at the entry point:
    s.x IN {56}
     .y IN {6}
     .[bits 7 to 31] IN UNINITIALIZED
```

Et les pointeurs ?

Jusqu'ici nos exemples ne contenaient pas de pointeurs. Les types C étaient simplement mappés sur les types Why3.

- `int32_t` → `Int32.t`
- `struct` → record.
- Tableau C → Tableau why3.

Il faut complexifier le modèle pour modéliser la mémoire.

```
/*@ requires \valid(p);
    ensures \result == 42; */
int f(int* p){
    int* q = p;
    q = 42;
    return *p;
}
```

Deux pointeurs peuvent pointer sur la même zone mémoire (ici p et q).

Problème d'aliasing.

Modélisation de la mémoire I

Au plus proche de la description données par le standard C.
Inspiré du modèle mémoire de CompCert.
La norme C parle d'"Array object". Ils sont tous séparés deux à deux.

Chacun de ces objets est identifié par sa base.

```
type base_t =
    | Absolute
    | Address int
    | Allocated int
```

Et chaque object a une permission et une taille.

```
type permission =
    | Freeable
    | Writable
    | Readable
    | Nonempty
    | None
```

Modélisation de la mémoire II

```
type memory_layout = {
    block_size : base_t -> size_t;
    address_perm : int -> permission;
    allocated_perm : int -> permission
    }
```

Et toute position dans la mémoire peut être identifiée de manière unique par sa base et son décalage par rapport l'offset initial de la base.

```
type loc = { base_addr : base_t ; offset : offset_t }
```

Types abstraits : modèle paramétrisé par les tailles des différents types (`size_t`, `ptrdiff_t`), l'endianness...

Modélisation de la mémoire III

Le contenu de la mémoire est simplement représentée comme une map des addresses vers des octets.

```
type mem_val = Undef | Byte BV8.t | Pointer ptr_slice

type value_map = offset_t -> mem_val

type memory_content = base_t -> value_map
```

mémoire non-initialisée→`Undef`

Granularité

```
function store_int32 (m : memory_content) (p : loc) (v : BV32.t) : memory_content =
  let v3 = BV8_BV32_conv.extract_byte0 v in
  let v2 = BV8_BV32_conv.extract_byte1 v in
  let v1 = BV8_BV32_conv.extract_byte2 v in
  let v0 = BV8_BV32_conv.extract_byte3 v in
  fun b ->
    if b <> p.base_addr then
      m b
    else
      fun o ->
        if o = p.offset then Byte v0
        else if o = add_offset p.offset offset_one then Byte v1
        else if o = add_offset p.offset offset_two then Byte v2
        else if o = add_offset p.offset offset_three then Byte v3
        else m b o
```

Granularité au niveau de l'octet : les lectures hétérogènes ne sont pas spécialement compliquées, support naturel des unions... mais les lectures homogènes ne sont pas spécialement simples

Assigns ACSL

```
type loc_set
clone export algebra.CommutativeMonoid with
  type t = loc_set,
  function op = union,
  constant unit = empty,
  axiom .

predicate assign (mc : memory_content) (mp : memory_perm) (mc' : memory_content) (mp' :
  forall l.
    readable mp l /\ readable mp' l /\ not (mem l ls) ->
    mc l.base_addr l.offset = mc' l.base_addr l.offset
```

Allocation dynamique gérée de manière très similaire

Problème : comment faire en sorte que les prouveurs instancient des règles de transitivité ?

Aliasing

Pas d'hypothèses de non-aliasing particulière (y compris pour des pointeurs vers des types différents)

- support naturel des casts de pointeur : les pointeurs sont non-typés
- projet : faire une analyse d'aliasing à part, assigner les pointeurs à des mémoires différentes lors de la traduction

Améliorer les contre-exemples

La mémoire est un gros objet compliqué, comment éviter que les prouveurs génèrent des valeurs triviales ?

```
\valid(m) /\ *m = { .st = UNINITIALIZED, .x = UNINITIALIZED };
```

Bitvecteurs

- Avantages :
  - taille fixée : forcément respectée par les contre-exemples
  - support naturel des opérations bitwise
  - encodage naturel des options : `option BV8.t` encodé par `BV9.t`
- Inconvénients :
  - la présence de `bv2nat` dans le contexte rend les contre-exemples et les preuves très difficiles
  - parti-pris : aucun entier mathématique dans les tâches, les entiers ACSL sont représentés par des BV256

Polymorphisme

polymorphisme dans la tâche $\rightarrow$ pas de contre-exemple pertinent

- versions monomorphisées de plusieurs modules de la librairie standard
- la paramétrisation se fait avec des déclaration `clone`

Éviter la sous-spécification

Exemple typique : un lemme pour décrire la lecture dans de la mémoire initialisée, lecture non-initialisée non spécifiée
Problème : rien n'empêche le prouveur de proposer un contre-exemple où rien n'est initialisé

```
function load_int8_opt (m : memory_content) (p : loc) : option_
  match (m p.base_addr p.offset) with
  | Undef -> none8ml m p
  | Byte b -> some8 b
  | Pointer ps ->
    if ps.l.base_addr = Absolute
    then some8 (offset_extract_slice ps.l.offset ps.slice)
    else some8 (ptr_to_bv ps)
  end
```