Towards an Executable ACSL

Virgile Prevosto

CEA List

July 13^{th}, 2010

outline

Frama-C

Frama-C in a nutshell

- `http://frama-c.com`
- Developed at CEA List and INRIA Saclay-Île de France
- Framework for multiple analyses of C
- a kernel providing basic functionalities, based on CIL
- a set of plug-ins performing various static analyses

Some existing verification plug-ins

- Value analysis
- Jessie plug-in
- Wp
- Sante and link to the PathCrawler tool

Frama-C

Frama-C in a nutshell

- `http://frama-c.com`
- Developed at CEA List and INRIA Saclay-Île de France
- Framework for multiple analyses of C
- a kernel providing basic functionalities, based on CIL
- a set of plug-ins performing various static analyses

Some existing verification plug-ins

- Value analysis
- Jessie plug-in
- Wp
- Sante and link to the PathCrawler tool

ACSL

The ACSL Language

- `http://frama-c.com/acsl.html`
- ANSI/ISO C Specification Language
- Formal specification language dedicated to C programs
- Based on function contracts and code assertions

Main objectives

- Syntax and concepts close to C
- Independence from a particular analysis
- Allow communication between analyses

ACSL

The ACSL Language

- `http://frama-c.com/acsl.html`
- ANSI/ISO C Specification Language
- Formal specification language dedicated to C programs
- Based on function contracts and code assertions

Main objectives

- Syntax and concepts close to C
- Independence from a particular analysis
- Allow communication between analyses

This presentation

A first contact with ACSL

- Presentation of ACSL
- Simple specification examples
- Covering most constructions of the language

Executable subset of ACSL

- Identifying constructions that are relevant to run-time assertion checking
- Discussing how they could be implemented/compiled in C.

This presentation

A first contact with ACSL

- Presentation of ACSL
- Simple specification examples
- Covering most constructions of the language

Executable subset of ACSL

- Identifying constructions that are relevant to run-time assertion checking
- Discussing how they could be implemented/compiled in C.

Specification

```
/*@
requires \valid(root) && finite_list(root);
assigns \nothing;
ensures mem(\result,root);
ensures
  \forall integer n; mem(n,root) ==>
      \result >= n;
*/
int max_list(list* root);
```

Implementation

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

Loop Invariants

```
/*@ loop invariant \valid(root) &&
  reachable(\at(root,Pre),root) &&
  mem(max,\at(root,Pre)) &&
  \forall int n;
     mem_sub(n, \at(root,Pre),root) ==>
       max >= n;
*/
while(root->next) {
  ...
}
```

Pure C expressions

Description

- All pure C expressions are valid ACSL expressions
- Integer operations are performed in $\mathbb{Z}$, real in $\mathbb{R}$.

Executable Setting

- Translation is trivial...
- ... Modulo the representation of mathematical datatypes
  - For integers, we could use some BigNum library (GMP)
  - For reals: ?

Logic Operators

Description

- Propositional connectors
- Quantifiers

Executable Setting

- Propositional level can be mapped directly to booleans
- Possible to support some limited form of quantification over a finite set, *e.g.*

```
\forall integer x; 0 <= x <= 10 ==> a[x] <= a[10]
```

Built-in constructions

Description

- Predicates and logic functions about pointers (**\valid**, **\baseaddr**,...).
- Evaluation of an expression at a given program point (**\at**, **\old**).

Executable Setting

- Would require a dedicated memory management (allocation table).
- Store needed results in intermediate variables as required.

Pre/Post conditions

Description

- **`requires`** supposed to hold when entering the function.
- **`ensures`** supposed to hold at function exit.
- Notion of **`behavior`** with **`assumes`** clause.

Executable Setting

- Use assertions (when the underlying predicate can be evaluated at run time).

Assigns clause

Description

- **assigns** describes the locations that may be modified during one run of the function.
- In other words, every location not in an **assigns** clause must retain the same value between pre and post state.

Executable Setting

- Using allocation table and type information, might be possible to check the exact property.
- Other possibility: Decorate each assignment with an assertion that the lval is in **assigns** (overapproximation but might be easier to check).

Inductive invariant

Description

- Must be true before entering the loop.
- If true when entering, must stay true at the end of a loop step.

Executable Setting

- Use plain assertions (property a bit weaker).

Termination

Description

- `loop variant`: integer expression which:
- is always non negative;
- is strictly decreasing between two loop steps

Executable Setting

- Use assertions
- Store the preceding value of the variant

Direct definition

```
/*@ predicate
      hd_max{L}(list* root, integer m) =
        \valid(root) && root->element <= m;
*/
```

Executable setting

- Inline the definition.
- Use C function.

Axiomatic definition

```
/*@ inductive
      reachable{L}(list* root, list* node) {
      case reachable_hd{L}:
        \forall list* l1; reachable(l1,l1);
      case reachable_next{L}:
        \forall list* l1, *l2;
          \valid(l1) ==>
          reachable(l1->next,l2) ==>
          reachable(l1,l2);
}*/
```

Executable setting

- Not translatable in the general case
- Might identify some patterns in axioms?