

Advanced ACSL and WP tutorial

Virgile Prevosto

April 25th, 2013

long n
for 0 < i < n
C[i] = 0
tmp2 =
of the

tmp2[i] = 0; if (i < (n-1) && ! tmp1[i]) >>= 1; else tmp2[i] = (1 << (n-1) - 1) && tmp2[i] + tmp1[i]; /* Then the second part takes the first one... */
tmp1[i] = 0; k = 0; k <= i; tmp1[i] += mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(M1)*M1 = MC2*M1*M1
i = 1; tmp1[i] >>= 1; /* Final rounding: tmp2[i] is now represented on 9 bits. */ if (tmp1[i] < -256) tmp2[i] = -256; else if (tmp1[i] > 255) tmp2[i] = 255; else tmp2[i] = tmp1[i];



- ▶ Specification by cases
- ▶ the subdomain is defined by **assumes** clause
- ▶ can give additional constraints with local **requires** clauses
- ▶ the behavior's postcondition is defined by **ensures**, **assigns** clauses
- ▶ **complete behaviors** states that given behaviors cover all cases
- ▶ **disjoint behaviors** states that given behaviors do not overlap



long n
for 0 < k
c1[i] = m
tmp2 =
of d

tmp2[0] = 0; for (int i = 0; i < n; i++) tmp2[i] = 0; for (int i = 0; i < n; i++) tmp2[i] = 0; Then the second part takes the first part
tmp1[0] = 0; for (int i = 0; i < n; i++) tmp1[i] = 0; for (int i = 0; i < n; i++) tmp1[i] = 0; The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
= 1; tmp1[0] = 0; Final reading: tmp2[0] is now represented on 3 bits: if (tmp1[0] < 256) tmp2[0] = 256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = 0;

Loop invariants - some hints

How to find a suitable loop invariant? Consider two aspects:

- ▶ identify **locations modified in the loop**
 - ▶ define their possible value intervals (relationships) after k iterations
 - ▶ use **loop assigns** clause to list variables that (might) have been assigned so far after k iterations
- ▶ identify realized actions, or **properties already ensured by the loop**
 - ▶ what **part of the job** already realized after k iterations?
 - ▶ what **part of the expected loop results** already ensured after k iterations?
 - ▶ why the next iteration can proceed as it does? ...

A **stronger property** on each iteration may be required to prove the final result of the loop.



Logic functions and predicates

- ▶ can be defined directly or through **axioms**
- ▶ may be parameterized by one (or more) program states:
predicate `foo{L}(int* a) = \at(*a, L) == 0;`
- ▶ `\at(\cdot, L)` can be omitted if there is no ambiguity (exactly one state in context).
 - ▶ this is what was done until now
- ▶ can be completed by additional **lemmas**



`\separated`(loc_1, loc_2, ..., loc_n) expresses the fact that loc_1, loc_2, loc_n are disjoint blocks of memory.

long n;
for (i = 0; i < n; i++)
C[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; k = (n-1) - i; if (tmp1[i] >= 1) tmp2[i] = (1 << (n-1) - i) + tmp1[i]; else tmp2[i] = tmp1[i]; } Then the second part takes the first part's
tmp1[i][k] = 0; k = k-1; tmp1[i][k] += mc2[i][k] * tmp2[k]; } The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1
i = i - tmp1[i][k] >= 1; } Final rounding: tmp2[i] is now represented on 9 bits: *if (tmp1[i] < -256) tmp2[i] = -256; else if (tmp1[i] > 255) tmp2[i] = 255; else tmp2[i] = tmp1[i];



- ▶ Can be specialized for a given behavior:
for b1, b2: **assert** `\true;`
- ▶ Will be proved under the **assumes** of the behavior(s)
- ▶ But will also only be used for the ensures of the behavior(s)

