FMSoft
Lecture 15 — Annotated programs and verification conditions in separation logic

(pre-lecture version)

Tadeusz Litak

February 5, 2019

Informatik 8, FAU Erlangen-Nürnberg

Annotated (or decorated) dynamic programs ADCom

No need to insist on annotating each command with both pre- and postcondition. Instead, define:

$c_1, c_2 ::=$ SKIP $\{B\}$ | X := $a\{B\}$ | $c1$; $c2$ |
IF b THEN $\{A\}\, c_1$ ELSE $\{A'\}\, c_2$ ENDIF $\{B\}$ |
WHILE b DO $\{A\}\, c_1$ END $\{B\}$ |
X := CONS $\overline{a}$ $\{B\}$ | DISPOSE $a\{B\}$ |
[a] := $a'\{B\}$ | X := [a] $\{B\}$ |
$\{A\}\, c$ | $c\{B\}$

Note that previously I forgot about the last line, this is why we could not write annotations involving the consequence rule!

An annotated/decorated PCA:

$d ::= \{A\}\, c\{B\}$, where $c\{B\} \in$ ADCom (prove by induction: each element of ADCom comes with a postcondition at the end!)

- We have the obvious function $erase : \mathsf{ADCom} \to \mathsf{DCom}$
- For a decorated PCA $\{\!\{A\}\!\}\, c\{\!\{B\}\!\}$, we would like ensure that $c\{\!\{B\}\!\}$ is indeed correctly annotated
- In particular, we want to have that
  $\vdash \{\!\{A\}\!\}\, erase(c\{\!\{B\}\!\})\{\!\{B\}\!\}$

  (and that the same holds for each component of c, e.g., that candidates for loop invariants are indeed loop invariants ...)
- We will do that by defining a function

$$vc : \mathsf{SepAss} \times \mathsf{ADCom} \to \mathsf{SepAss}$$

  s.t. $\vDash vc(A, c\{\!\{B\}\!\})$ implies $\vdash \{\!\{A\}\!\}\, erase(c\{\!\{B\}\!\})\{\!\{B\}\!\}$
- Furthermore, the separation logic assertion obtained this way, while complex, can be broken down into pieces which tend to be manageable for automated tools ...

Verification conditions

$$vc(A,\ \texttt{SKIP}\ \{B\}) = A \to B$$

$$vc(A,\ \texttt{X}\ \texttt{:=}\ a\{B\}) = A \to B[a/\texttt{X}]$$

$$vc(A, c_1\{B\}\ \texttt{;}\ c_2) = vc(A, c_1\{B\}) \wedge vc(B, c_2)$$

$$\begin{aligned} vc(A,\ \texttt{IF}\ b\ \texttt{THEN}\ \{A_1\}\, c_1\{A_1'\} \\ \texttt{ELSE}\ \{A_2\}\, c_2\{A_2'\}\ \texttt{ENDIF}\ \{B\}) = \\ (A \wedge b \to A_1) \wedge (A \wedge \neg b \to A_2) \wedge \\ (A_1' \leftrightarrow B) \wedge (A_2' \leftrightarrow B) \wedge \\ vc(A_1, c_1\{A_1'\}) \wedge vc(A_2, c_2\{A_2'\}) \end{aligned}$$

$$\begin{aligned} vc(A,\ \texttt{WHILE}\ b\ \texttt{DO}\ \{A_1\}\, c_1\{A_1'\}\ \texttt{END}\ \{B\}) = \\ (A \to A_1') \wedge \\ (A_1 \leftrightarrow A_1' \wedge b) \wedge \\ (B \leftrightarrow A_1' \wedge \neg b) \wedge \\ vc(A_1, c_1\{A_1'\}) \end{aligned}$$

Some of these equivalences can be weakened, some not. Which ones?

$$vc(A,\ \mathtt{X}\ \mathtt{:=}\ \mathtt{CONS}\ \overline{a}\ \{B\}) =$$
$$A \to \forall i'.(i' :\to \overline{a}) \mathrel{-\!\!*} B[i'/\mathtt{X}])$$
$$vc(A,\ [a]\ \mathtt{:=}\ a'\{B\}) =$$
$$A \to (a :\to \_) * ((a :\to a') \mathrel{-\!\!*} B)$$
$$vc(A,\ \mathtt{X}\ \mathtt{:=}\ [a]\ \{B\}) =$$
$$A \to \exists i'.((a :\to i') * \mathtt{true}) \wedge B[i'/\mathtt{X}] \qquad i' \# B$$
$$vc(A,\ \mathtt{DISPOSE}\ a\{B\}) = A \to (a :\to \_) * B$$
$$vc(A, \{A'\}\, c) = (A \to A') \wedge vc(A', c)$$
$$vc(A, c\{B'\} \to \{B\}) = vc(A, c\{B'\}) \wedge (B' \to B)$$

Recall:

$$\vdash \{\forall i'.(i' :\to \overline{a}) \mathrel{-\!\!*} A[i'/\mathtt{X}])\}\ \mathtt{X} := \mathtt{CONS}\ \overline{a}\ \{A\}$$

$$\vdash \{(a :\to \_) * ((a :\to a') \mathrel{-\!\!*} A)\}\ [a] := a'\ \{A\}$$

$$\vdash \{\exists i'.((a :\to i') * \mathtt{true}) \wedge A[i'/\mathtt{X}]\}\ \mathtt{X} := [a]\ \{A\}\ \ (i' \text{ fresh for } A)$$

$$\vdash \{(a :\to \_) * B\}\ \mathtt{DISPOSE}\ a\{B\}$$

- We can now prove the desired result:

Theorem
$\vDash vc(A, c\{B\})$ *implies* $\vdash \{A\}\, erase(c\{B\})\{B\}$.

Proof sketch.
Straightforward induction on c, using the definition of vc, (global backward) rules for program constructs and the consequence rule for preconditions. □

- We just need to ensure that the tool has as many tactics/heuristics for verification conditions as possible ...
- ...and/or that our annotations are helpful enough
- Aside question: is the reverse implication true?

```
{ emp }
X := CONS 42
{ X :-> 42 };
Y := CONS 23
{ X :-> 42 * Y :-> 23 };
DISPOSE X
{ Y :-> 23 } ;
DISPOSE Y
{ emp }
```

Note that if we wanted to treat annotated programs as derivations in disguise, we would not only need more annotations, but also to use some additional rules. E.g., the small axiom for allocation would yield

$\vdash$ {(X == i)$\wedge$emp} X := CONS 42 {X :-> 42 * emp}

in the first step, and we would need VPrEl' to get rid of (X == i) ...

- For more sophisticated examples, we would need inductive definitions/predicates: lists, trees etc.
- Easy to encode in our infinitary language

  in fact, we already did it with factorial
- In real-life systems, can be non-trivial
- Earliest tools based on SL like Smallfoot provided support for lists or trees, but not for general inductive definitions
- As you have seen in tutorials, there is no problem with these in IDF-based VeriFast
- And, of course, also in formalizations of separation logic in general purpose proof assistants like Coq (see SemProg next semester)

Linked lists

Let us define a sequence of predicates

$$list : \mathbb{N} \to \mathbb{Z} \to \mathsf{SepAss}$$

"$list_n(z, h)$: The heaplet is a linked list of length n and its first address is z"

Recall that `z == nil` is encoded by `z < 0` ...

$$\texttt{list(0,z)} := \texttt{z == nil} \wedge \texttt{emp}$$
$$\texttt{list(n+1,z)} := \exists i'.(\texttt{z :-> } i') \texttt{ * list(n,}i'\texttt{)}$$
$$\texttt{list(z)} := \bigvee_{n \in \mathbb{N}} \texttt{list(n,z)}$$

Of course, this is a rather farcical notion of a list. We'd need to extend our language somewhat to do better.

```
{ emp }
Z := -1
{ list(0,z) } ;
X := 0
{ X >= 0 ∧ list(X,Z) } ;
WHILE (not (X == n))
DO
    { X >= 0 ∧ list(X,Z) ∧ not (X == n)}
    Z := CONS Z
    { X >= 0 ∧ list(X + 1,Z) } ;
    X := X + 1
    { X >= 0 ∧ list(X,Z) } ;
END
{ X >= 0 ∧ list(X,Z) ∧ X == n} →
{ list(n,Z) }
```

How one could improve this example?

Extend the language with products

First suggestion of C. Rauch

```
Z := <0, 0>
X := 0
WHILE (not (X == n))
DO
  X := X + 1
  Z := CONS < X , Z >
END
```

Christoph: would require extending the assertion language too

An alternative: do it in VeriFast

Christoph's VeriFast code, part I

```
#include "stdlib.h"

struct node {
  struct node * next;
  int value;
};

/*@
  inductive ilist = inil | icons (int, ilist);

  predicate range(int n, struct node *x, ilist l) =
  n <= 0
    ? x == 0 &*& l == inil
    :    x->next |-> ?next
      &*& x->value |-> n-1 &*& l == icons (n, ?ls)
      &*& malloc_block_node(x) &*& range(n-1,next,ls);

@*/
```

Christoph's VeriFast code, part II

```
struct node * f(int n)
//@ requires emp;
//@ ensures range(n,result,?l);
{
 struct node *z = 0;
 int x = 0;
 //@ close range(0,z,inil);
 while(!(x==n))
 //@ invariant x >= 0 &*& range(x,z,?l);
 {
   struct node * w = malloc(sizeof(struct node));
   if(w == 0) abort();
   w->next = z;
   w->value = x;
   z = w;
   x = x + 1;
   //@close range(x,z,icons(x,l));
 }
 return z;
}
```

Another SL-based tool ...

- ...we won't be able to talk about
- Infer: presently developed @Facebook but available open source
  `https://github.com/facebook/infer`
  Calcagno and Distefano, *Infer: An Automatic Program Verifier for Memory Safety of C Programs*, NFM 2011
- See Calcagno et al. *Moving Fast with Software Verification*, NFM 2015
  `https://research.fb.com/publications/moving-fast-with-software-verification/`
- ...and blog entries:
  `https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/`
  `https://research.fb.com/four-facebook-employees-win-the-prestigious-cav-award/`