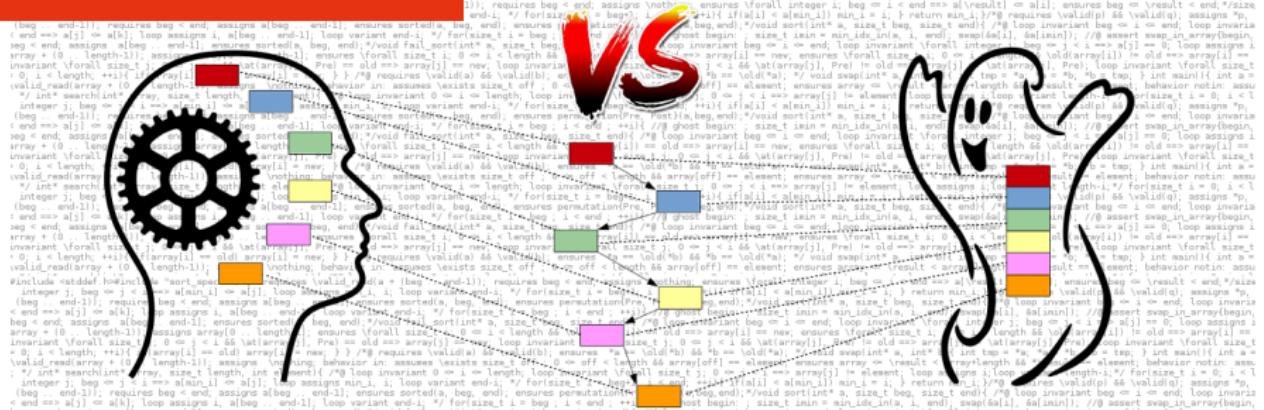


Logic Against Ghosts: Comparison of two Proof Approaches for Linked Lists



Allan Blanchard

Nikolai Kosmatov

Frédéric Loulergue – April 10, 2019 @ SAC-SVT 2019

Formal verification of IoT Software

The VESSEDA project

- make IoT software safer
- mainly by making formal verification more affordable for this field
- How? Make it easier to apply

Use case: the Contiki Operating System

- an operating system for (low power) IoT devices
- critical module: the linked-list module

Frama-C at a glance



Software Analyzers

- a **F**ramework for **M**odular **A**nalysis of **C** code
- developed at CEA List
- ACSL annotation language
- extensible plugin-oriented platform
 - > collaboration of analyses over same code
 - > inter plugin communication through ACSL formulas
 - > adding specialized plugins is easy
- <http://frama-c.com/> [Kirchner et al. FAC 2015]

Deductive verification with Frama-C

The WP plugin

- based on Dijkstra's weakest precondition calculus
- verify that:
 - > every function ensures its postcondition,
 - > each function call respects the precondition about the input,
 - > no runtime error can happen
- input: program annotated with ACSL contracts
 - > precondition, postcondition, loop invariant, ...
- output: a set of verification conditions (VCs)
 - > that are then transmitted to automatic/interactive provers

The LIST module - Overview

Provides a generic list API for linked lists

- about 176 LOC (excl. MACROS)
- required by 32 modules of Contiki
- more than 250 calls in the core part of Contiki

Some special features

- no dynamic allocation
- does not allow cycles
- maintain item unicity

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;  
  
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

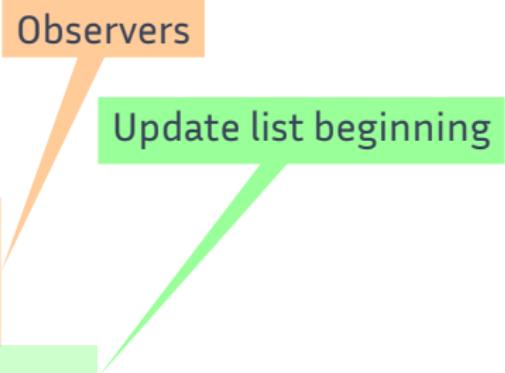
The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;  
  
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

Observers

The LIST module - A rich API

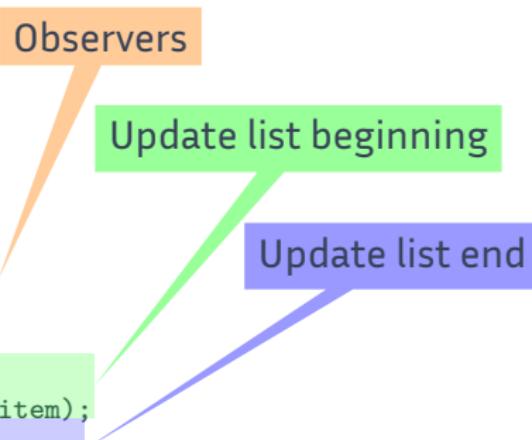
```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;  
  
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```



The diagram illustrates the interaction between the LIST module's API and external observers. An orange callout labeled "Observers" points to the first two lines of the API code: the definition of the list structure and the list_t typedef. A green callout labeled "Update list beginning" points to the list_init function, which is highlighted with a green background.

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;  
  
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```



The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;  
  
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

Observers

Update list beginning

Update list end

Update list anywhere

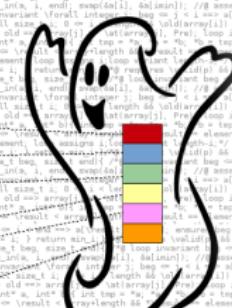
02

Ghosts for lists

The diagram illustrates a complex piece of C code with numerous annotations. The code is primarily concerned with array manipulation, specifically dealing with arrays of pointers and integer arrays. Annotations include:

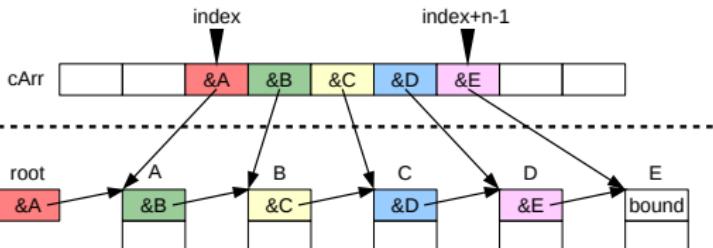
- Red box:** Points to a line involving `for(i=0; i < size; i++)` and `for(j=0; j < size; j++)`.
- Blue box:** Points to a line involving `for(i=0; i < size; i++)` and `for(j=i; j < size; j++)`.
- Green box:** Points to a line involving `for(i=0; i < size; i++)` and `for(j=0; j < size; j++)`.
- Pink box:** Points to a line involving `for(i=0; i < size; i++)` and `for(j=0; j < size; j++)`.
- Orange box:** Points to a line involving `for(i=0; i < size; i++)` and `for(j=0; j < size; j++)`.

The annotations provide detailed descriptions of the code's behavior, such as loop invariant maintenance, pointer validity, and array element access. The code itself is heavily annotated with comments explaining its purpose and the logic behind each step.



Proof using ghost arrays [NFM 2018]

Ghost code



Actual code

Example

```
list* list_pop(list_t pLst) /*@ ghost (list** array, int index, int size) */ {
    list* root = *pLst ;
    if(root){
        /*@ ghost array_pop(pLst, array, index, size) ;
        *pLst = root->next ;
    }
    return root ;
}
```

Limitations

Ghosts are still quite concrete

- we need to show that the array is **spatially separated** from the list elements
- which is **costly** for the verification process
 - > about a third of the annotations are dedicated to this
 - > it pollutes the proof context

Maintain equivalence

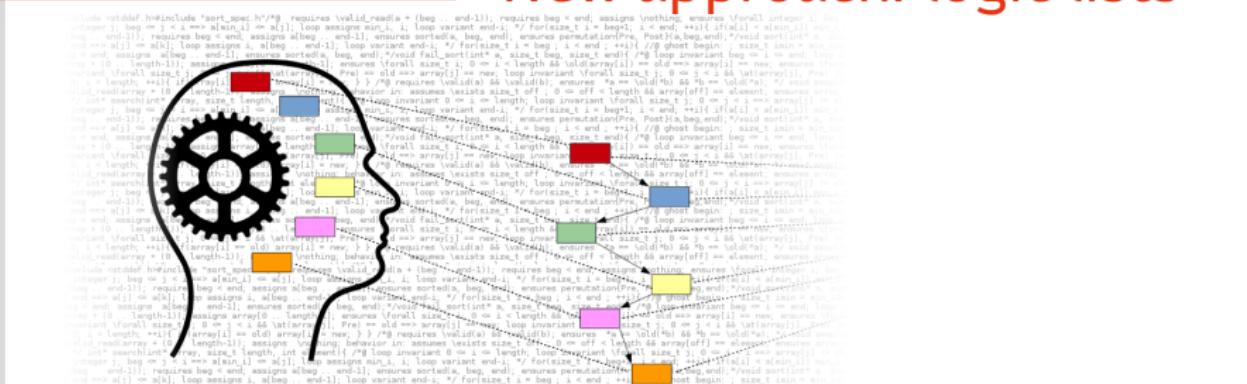
- each time we modify the list, we have to modify the content of the array
- ⇒ **more functions to verify**

The `list_insert` function was not proved

- ... and we expected it to be **really hard** to prove because of memory separation

03

New approach: logic lists



ACSL logic lists

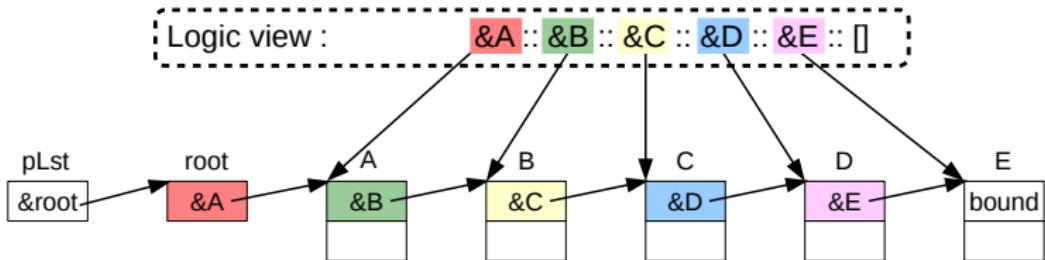
```
\let empty_list = \Nil ;
\let an_elem = \Cons(1, empty_list) ;
\let another_one = [| 2 |] ;
\let multi = [| 3, 4, 5 |] ;
\let concat = (an_elem ^ another_one ^ multi ^ [| 6, 7, 8 |]) ;
/* etc ... */
```

What is the difference with ghost arrays?

Purely logic type:

- easier to handle for SMT solvers,
 - > natively handled type
 - > no encoding of the C semantics (related to arrays)
- no separation property needed,
- leads to more concise specification.

Formalization I



```

/*@
inductive linked_ll{L}(list *bl, list *el, \list<list*> ll) {
  case linked_ll_nil{L}: \forall list *el;
    linked_ll{L}(el, el, \Nil);
  case linked_ll_cons{L}: \forall list *bl, *el, \list<list*> tail;
    \separated(bl, el) \Rightarrow \valid(bl) \Rightarrow
    linked_ll{L}(bl->next, el, tail) \Rightarrow
    separated_from_list(bl, tail) \Rightarrow
    linked_ll{L}(bl, el, \Cons(bl, tail));
}
*/
  
```

An issue: no global \exists in contracts

A classic way to specify a list pop feature:

```
/*@ \exists list* hd, \list<list*> l ;
@ ...
@ ...
@ behavior not_empty:
@   assumes *list != NULL ;
@   requires linked_ll(*list, NULL, \Cons(hd, l));
@   assigns *list ;
@   ensures \result == hd == \old(*list) ;
@   ensures linked_ll(*list, NULL, l);
*/
list * list_pop(list_t list);
```

An issue: no global \exists in contracts

A classic way to specify a list pop feature:

```
/*@ \exists list* hd, \list<list*> l ;
@ ...
@ ...
@ behavior not_empty:
@   assumes *list ≠ NULL ;
@   requires linked_ll(*list, NULL, \Cons(hd, l));
@   assigns *list ;
@   ensures \result == hd == \old(*list) ;
@   ensures linked_ll(*list, NULL, l);
@*/
list * list_pop(list_t list);
```

ACSL does not allow to quantify variables over a contract

An issue: no global \exists in contracts

Introduce quantification in each clause?

```
/*@ ...
@  

@ behavior not_empty:  

@ assumes *list != NULL ;  

@ requires ∃ list* hd, \list<list*> l ;  

@           linked_ll(*list, NULL, \Cons(hd, l));  

@ assigns *list ;  

@ ensures \result == \old(*list) ;  

@ ensures ∃ list* hd, \list<list*> l ;  

@           linked_ll{Old}(*list, NULL, \Cons(hd, l)) ∧  

@           linked_ll(*list, NULL, l);  

@*/  

list * list_pop(list_t list);
```

An issue: no global \exists in contracts

Introduce quantification in each clause?

```
/*@ ...
@ 
@ behavior not_empty:
@   assumes *list != NULL ;
@   requires ∃ list* hd, \list<list*> l ;
@           linked_ll(*list, NULL, \Cons(hd, l));
@   assigns *list ;
@   ensures \result == \old(*list) ;
@   ensures ∃ list* hd, \list<list*> l ;
@           linked_ll{\Old}(*list, NULL, \Cons(hd, l)) ∧
@           linked_ll(*list, NULL, l);
*/
list * list_pop(list_t list);
```

Contract would be far more difficult to use by SMT solvers
(Because of the need to instantiate the existential variables)

Formalization II

Solution: a logic function that provides the right value to solvers

```
/*@ axiomatic To_ll {
    logic \list<list*> to_ll{L}(list* bl, list* el)
        reads { e->next
            | list* e ; \valid(e) ∧ in_list(e, to_ll(bl, el)) } ;

    axiom to_ll_nil{L}: ∀ list *el ;
        to_ll{L}(el, el) == \Nil ;

    axiom to_ll_cons{L}: ∀ list *bl, *el ;
        \separated(bl, el) ⇒
        \valid(bl) ⇒
        ptr_separated_from_list(bl, to_ll{L}(bl->next, el)) ⇒
        to_ll{L}(bl, el) == (\Cons(bl, to_ll{L}(bl->next, el))) ;
}
```

Example of contract

```
/*@ requires linked_ll(*list, NULL, to_ll(*list, NULL));
@ assigns *list ;
@ ensures linked_ll(*list, NULL, to_ll(*list, NULL));
@ behavior empty:
@   assumes *list == NULL ;
@   ensures \result == NULL ;
@ behavior not_empty:
@   assumes *list != NULL ;
@   ensures \result == \old(*list) ;
@   ensures to_ll{Pre}(\at(*list, Pre), NULL) ==
@       ([| \at(*list, Pre) |] ^ to_ll(*list, NULL));
@ complete behaviors; disjoint behaviors;
*/
list * list_pop(list_t list);
```

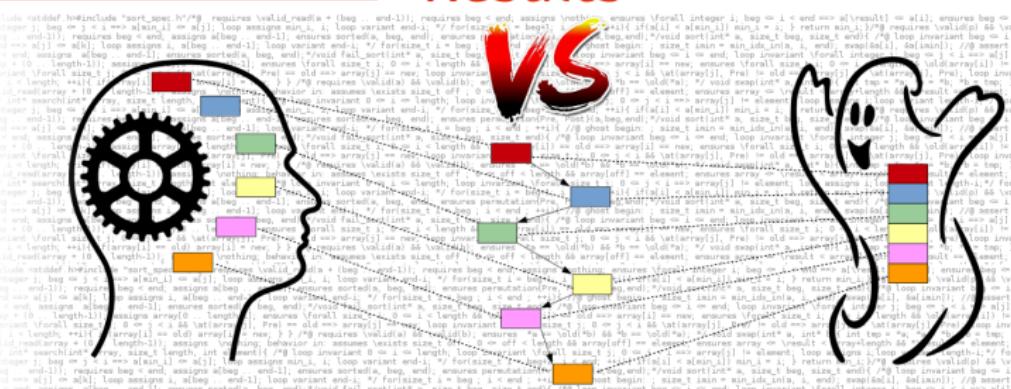
Example of lemma (split a list)

```
/*@
lemma linked_ll_split{L}:
  ∀ struct list *bl, *el, \list<struct list*> l1, l2;
    linked_ll(bl, el, l1 ^ l2) ⇒ l1 ≠ \Nil ⇒
      \let bound = \nth(l1, \length(l1) - 1)->next ;
      linked_ll(bl, bound, l1) ∧ linked_ll(bound, el, l2) ;
*/
/*@
lemma to_logic_list_split{L}:
  ∀ struct list *bl, *el, *sep, \list<struct list*> ll;
  ll ≠ \Nil ⇒ linked_ll(bl, el, ll) ⇒
  ll == to_logic_list(bl, el) ⇒
  in_list(sep, ll) ⇒
    ll == (to_logic_list(bl, sep) ^ to_logic_list(sep, el));
*/

```

04

Results



Inria
inventors for the digital world

The module is now entirely proved

Written annotations

- Total: 1700 LoA (Lines of Annotations)
- 410 LoA for contracts
- 270 LoA for logic definitions and lemmas
- 1020 LoA for guiding annotations

Verification conditions (VCs)

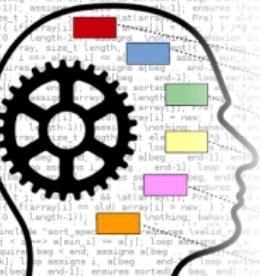
- Total: 757
- 33 for lemmas, proved using Coq
- 2 assertions needed to be proved with Coq
- all other VCs are proved automatically

We also optimized the `list_insert` function

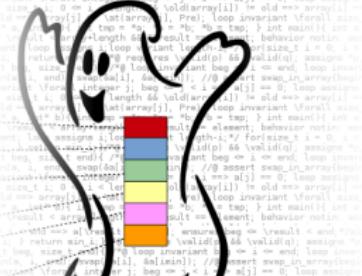
Comparison of approaches

Generated verification conditions:

include <assert.h>
 include <limits.h>
 #include <math.h>
 #include <string.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <stropts.h>
 #include <stropts.h>
 #include <stropts.h>



83	ensures	134
53	assigns	71
16	requires	69
264	guiding	399
54	RTE	108
33	Lemmas	24
503	Total	805
0.7s	Time/VC (automatic)	1.64s



include <assert.h>
 include <limits.h>
 #include <math.h>
 #include <string.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <stropts.h>
 #include <stropts.h>
 #include <stropts.h>

(Excluding `list_insert`, not proved with the ghost approach)

Comparison of approaches

The logic version ...

- allowed us to prove a previously unproven function (`list_insert`)
- requires less annotations (and thus less VCs),
- enables faster VCs proof

Some limitations

- the logic version requires more lemmas (and their Coq proofs are harder)
- it is not clear whether the logic version can be made executable

05

Let's sum up!

Logic lists lead to efficient proof

The results show that with logic lists

- the proof of the functions is easier
 - > it requires less guiding annotations
 - > VCs are discharged faster by SMT solvers
- the approach requires more lemmas
 - > due to the generation of the witness for the logic list
 - > these proofs are harder to write for non-experts
 - > however we do not expect it to be a major problem
- it is not clear whether this approach can be made executable

Ongoing and Future work

Most recent work

- New proof (based on the ghost version) using auto-active verification [NFM 2019]

Ongoing and Future work

Most recent work

- New proof (based on the ghost version) using auto-active verification [NFM 2019]

Improve ghost support

- the problem with ghosts was due to memory separation
- one could take into account the ghost status of variables to assume separation

Ongoing and Future work

Most recent work

- New proof (based on the ghost version) using auto-active verification [NFM 2019]

Improve ghost support

- the problem with ghosts was due to memory separation
- one could take into account the ghost status of variables to assume separation



Here comes a new challenger?

- another approach: pure observational function
- could lead to the same level of abstraction
(no memory separation problem)
- while being directly executable

Ongoing and Future work

Most recent work

- New proof (based on the ghost version) using auto-active verification [NFM 2019]

Improve ghost support

- the problem with ghosts was due to memory separation
- one could take into account the ghost status of variables to assume separation



Here comes a new challenger?

- another approach: pure observational function
- could lead to the same level of abstraction
(no memory separation problem)
- while being directly executable

Thank you for your attention!

Thank you!