Lemma Functions for Frama-C: C Programs as Proofs

Grigoriy Volkov, NRU HSE
Mikhail Mandrykin, ISP RAS
Denis Efremov, NRU HSE

Deductive Verification

One of the most “heavyweight” static techniques of formal reasoning about software properties, deductive verification makes it possible to mathematically prove **conformance between source code and a formal specification** of its requirements.

Formal specifications for imperative programming languages are usually based on the Hoare logic, so each function has **pre- and postconditions**.

Function Specification Example

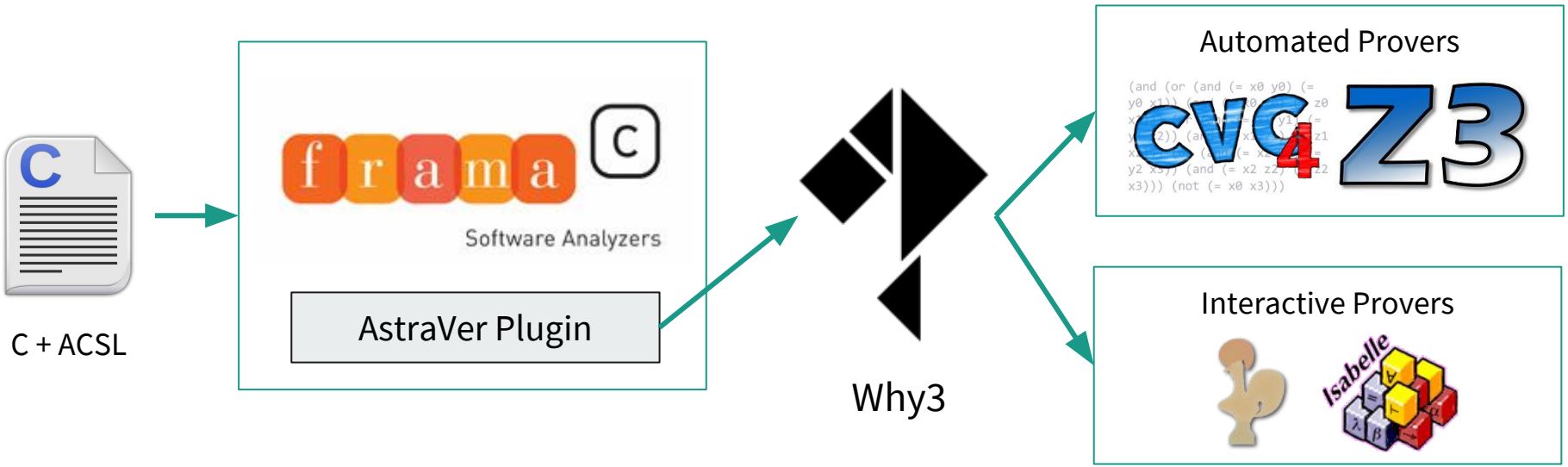
```
/* http://toccata.lri.fr/gallery/BinarySearchACSL.en.html */

int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;

  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

Function Specification Example

```
/* http://toccata.lri.fr/gallery/BinarySearchACSL.en.html */
/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ ensures -1 <= \result < n;
  @ behavior success:
  @   ensures \result >= 0 ==> t[\result] == v;
  @ behavior failure:
  @   assumes sorted(t,0,n-1);
  @   ensures \result == -1 ==>
  @     \forall integer k; 0 <= k < n ==> t[k] != v; */
int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;
  /*@ loop invariant 0 <= l && u <= n-1;
    @ for failure:
    @   loop invariant
    @   \forall integer k; 0 <= k < n && t[k] == v ==> l <= k <= u;
    @ loop variant u-l; */
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

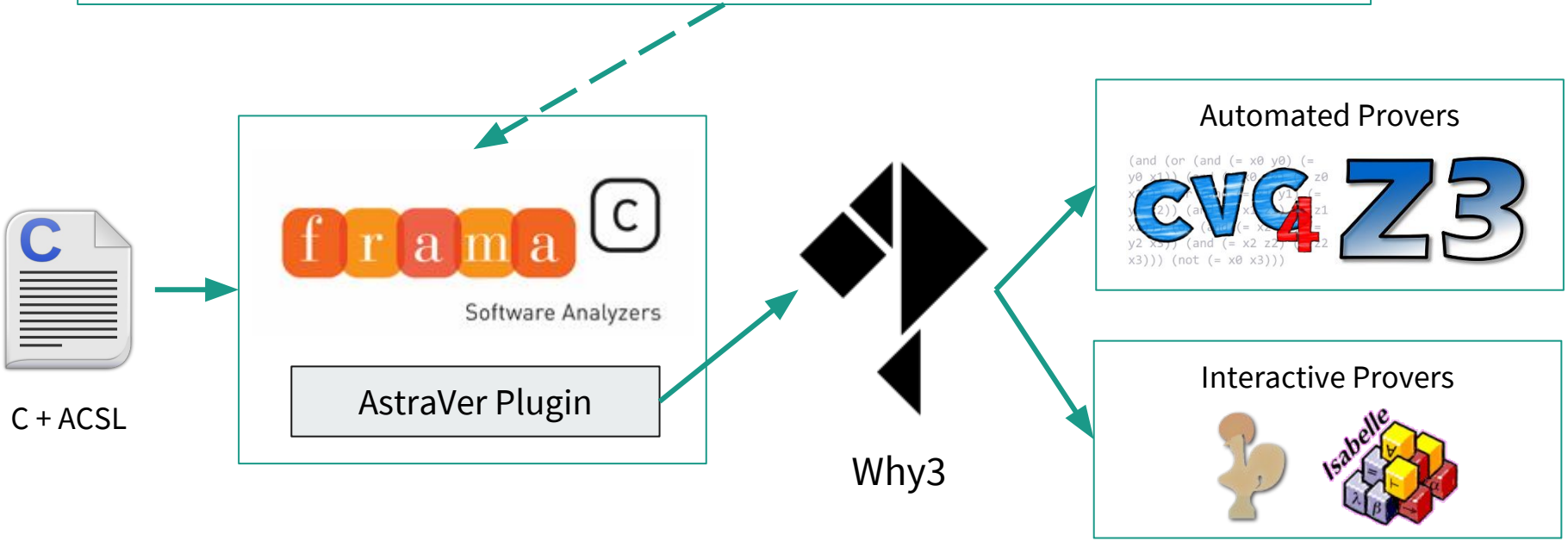
AstraVer Toolset

Open source toolchain for practical deductive verification of ACSL-annotated C programs, targeted at Linux kernel module source code.



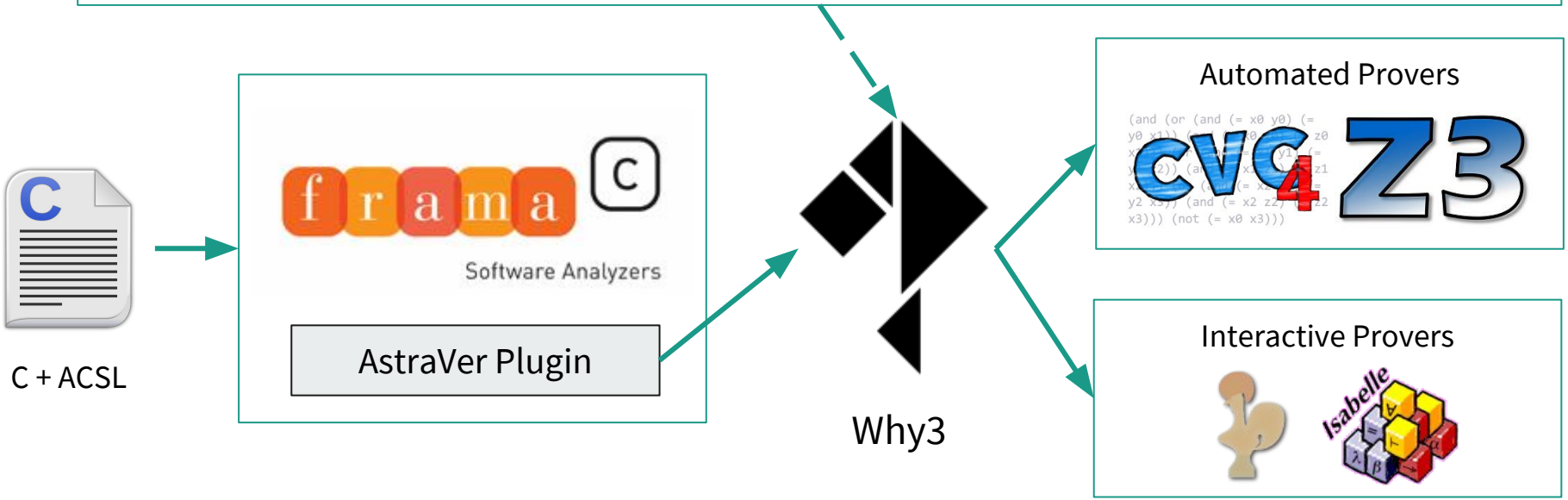
Frama-C

A platform for C code analysis that lets plugins implement various verification techniques, analyses, metrics, etc.



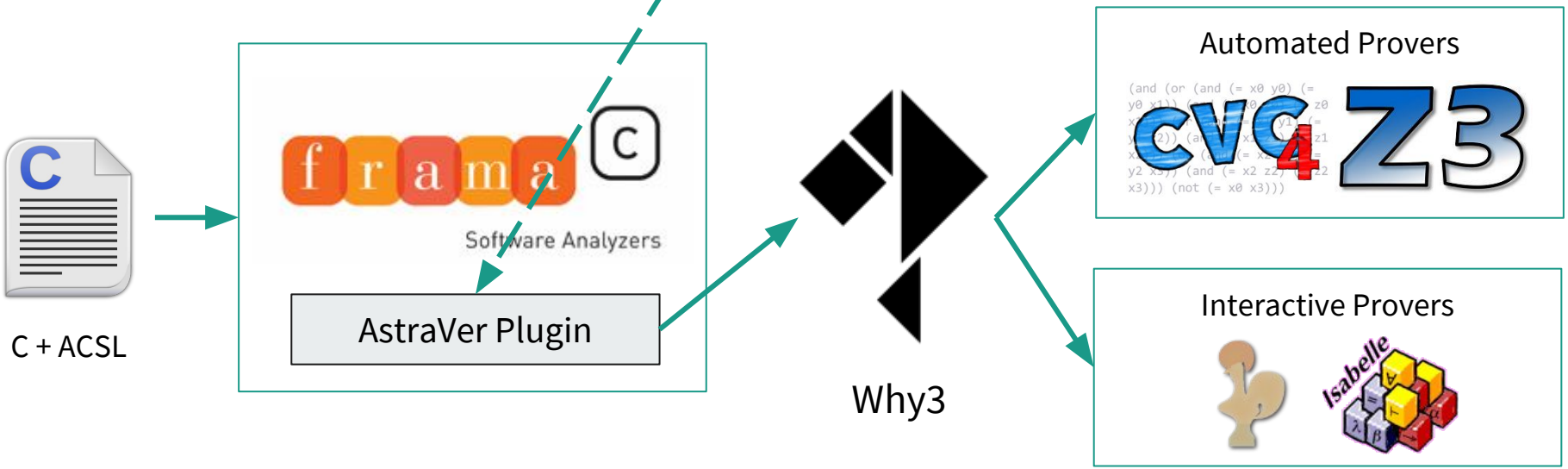
Why3

A deductive verification platform with its own unified programming and specification language, WhyML. Generates verification conditions and discharges them using external provers.
Offers a GUI IDE for interactive goal transformation and prover management.



AstraVer Plugin

A plugin for deductive verification of C code — our fork of the Jessie plugin.
Translates ACSL-annotated C code to WhyML.



From Lemmas to Lemma Functions

Lemmas

The most complex specification properties are usually factored out into **lemmas** - independent logical statements that must be proved separately.

```
/*@ axiomatic Strchr {
  logic char *strchr(char *str, char c) =
    *str == c ? str : ((*str == '\0') ? (char *) \null : strchr(str+1, c));

  lemma strchr_at_end_zero:
    \forall char *str, c; \valid(str) && *str == '\0' ==>
      strchr(str, c) == \null;
 … } */

/*@ requires valid_str(s);
    ensures \result == strchr(s, c); … */
char *strchr(const char *s, int c);
```

Proving Lemmas

Sometimes lemmas can be proved automatically, but usually they are too difficult for SMT solvers (e.g. lemmas that require proof by induction).

Several ways to deal with that:

- Leave them unproved
- Use interactive proof assistants like Coq
  - New (difficult) languages and tools to learn
  - Instability: irrelevant changes in the source code often cause the proof to fail, requiring the proof to be adjusted again
  - Proofs are kept separately from the specification
    - Not visible when reading the source code (harder to estimate the effort required for various code modifications)
    - Problems with proof sharing (using one lemma in multiple files)
- Our approach: auto-active verification with **lemma functions** written in C

Auto-Active Proofs

Auto-active verification is any verification technique where user input is supplied before verification condition generation.

Lemma functions are a way to express proofs of user-defined auxiliary lemmas as *pure* (no side effects) *total* (always terminating) imperative functions.

In a lemma function the function body serves as a guide for automatic provers (SMT solvers) that efficiently indicates the relevant parts of the search space.

Example

We have a recursive logical function:

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s :
    (*s ≡ '\0') ? s :
      strchrnul(s + 1, c);
```

And a lemma describing a property about it:

```
lemma strchrnul_in_range:
  ∀ char *s, char c;
    valid_str(s) ⇒
      s ≤ strchrnul(s, c) ≤ s + strlen(s);
```

SMT solvers are not capable of proving the lemma…

Example

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s : (*s ≡ '\0') ? s : strchrnul(s + 1, c);
```

Let's rewrite the lemma as a function!

```
lemma strchrnul_in_range:
  ∀ char *s, char c; valid_str(s) ⇒ s ≤ strchrnul(s, c) ≤ s + strlen(s);
```

\forall quantified variables become function arguments

```
ghost
/@ lemma
 @ requires valid_str(s);
 @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s);
 @/
void strchrnul_in_range(char *s, char c) {
  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
}
```

Example

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s : (*s ≡ '\0') ? s : strchrnul(s + 1, c);
```

Let's rewrite the lemma as a function!

```
lemma strchrnul_in_range:
  ∀ char *s, char c; valid_str(s) ⇒ s ≤ strchrnul(s, c) ≤ s + strlen(s);

ghost
/@ lemma
 @ requires valid_str(s);
 @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s);
 @/
void strchrnul_in_range(char *s, char c) {
  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
}
```

the logical property is rewritten as pre- and postconditions

Example

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s : (*s ≡ '\0') ? s : strchrnul(s + 1, c);

ghost
/@ lemma
 @ requires valid_str(s);
 @
 @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s);
 @/
void strchrnul_in_range(char *s, char c) {
  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
}
```

The if statement splits between the base and inductive cases!
Base case: *s is equal to '\0' or c - the postcondition trivially follows from the definition of strchrnul

Example

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s : (*s ≡ '\0') ? s : strchrnul(s + 1, c);

ghost
/@ lemma
 @ requires valid_str(s);
 @ decreases strlen(s);
 @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s);
 @/
void strchrnul_in_range(char *s, char c) {
  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
}
```

we add a variant to prove that recursion is well-founded

Inductive case: the postcondition follows from the inductive assumption for strings of a smaller length.
The body of the if statement provides **an instance of the lemma** for length s + 1

Example

```
logic char *strchrnul(char *s, char c) =
  (*s ≡ c) ? s : (*s ≡ '\0') ? s : strchrnul(s + 1, c);

ghost
/@ lemma
 @ requires valid_str(s);
 @ decreases strlen(s);
 @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s);
 @/
void strchrnul_in_range(char *s, char c) {
  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
}
```

This is easily verified automatically by discharging verification conditions using an SMT solver such as CVC4 or Alt-Ergo!

Implementation Details

Integrating Lemma Functions into ACSL/Frama-C

We implemented support for lemma functions as a special case of **total pure ghost functions**.

ACSL has a syntax for ghost functions - C functions only used for verification (written in specification comments, so not visible to a compiler).

Meanwhile the AstraVer plugin is able to generate VCs to check ACSL `assigns`, `allocates`, `decreases` and `terminates` clauses of function contracts.

The missing part was to **make lemma functions usable like regular lemmas**, i.e. automatically available in the contexts following the definition. We accomplished this by implementing a relatively simple source code transformation that is invoked by adding the `lemma` keyword to the specification of a ghost function.

Axiom Generation

For a function contract of the form

function arguments

global variables

return value

```
/@ requires R(p_1, …, p_n, g_1, …, g_n);
 @ ensures E(p_1, …, p_n, g_1, …, g_n, \result);
 @/
```

An axiom of the following form is generated:

```
∀ typeof(p_1) p_1, …, typeof(p_n) p_n, typeof(g_1) g_1, …,typeof(g_n) g_n;
  ∃ typeof(\result) \result;
    R(p_1, …, p_n, g_1, …, g_n) ⇒
      E(p_1, …, p_n, g_1, …, g_n, \result)
```

In order to make the property specified by the lemma available globally.

Axiomatic Block Wrapping

To prevent the generated axiom from trivially proving the function's verification conditions, we place the axiom into **an axiomatic block**:

```
axiomatic LF__Axiomatic__strchrnul_in_range {
  axiom LF__Lemma__strchrnul_in_range: ∀ char *s …;
  predicate LF__Pred__strchrnul_in_range(ℤ x) = \true;
}
```

AstraVer implements **on-demand import** of all definitions from axiomatic blocks based on the occurrence of any symbol defined in the block, so a dummy predicate is added to the block, and its usage is inserted (as a precondition) into the function contracts following the lemma function's definition.

Complete Generated Code Example

```
/*@ logic char *strchrnul(char *s, char c) =
      (*s ≡ c) ? s : (*s ≡ '\0') ? s :
        strchrnul(s + 1, c); */

/*@ ghost
  @ /@ lemma
  @  @ requires valid_str(s);
  @  @ decreases strlen(s);
  @  @ ensures s ≤ strchrnul(s, c)
  @  @    ≤ s + strlen(s);
  @  @/
  @ void strchrnul_in_range(
  @      char *s, char c) {
  @  if (*s != '\0' && *s != c)
  @    strchrnul_in_range(s + 1, c);
  @ }
  @*/
```

```
/*@ ghost
  @ /@ requires valid_str(s);
  @  @ decreases strlen(s);
  @  @ assigns \nothing;
  @  @ allocates \nothing;
  @  @ terminates \true;
  @  @ ensures s ≤ strchrnul(s, c) ≤ s + strlen(s); @/
  @ void strchrnul_in_range(char *s, char c) {
  @  if (*s != '\0' && *s != c) strchrnul_in_range(s + 1, c);
  @ } */

/*@ axiomatic LF__Axiomatic__strchrnul_in_range {
  axiom LF__Lemma__strchrnul_in_range:
    ∀ char *s, char c;
    valid_str(s) ⇒
    s ≤ strchrnul(s, c) ≤ s + strlen(s);

  predicate
    LF__Pred__strchrnul_in_range(ℤ x) = \true;
} */
```

Results

The VerKer Project

VerKer is a collection of ACSL specifications for various standard library functions (string and memory manipulation) taken from the Linux kernel. We use VerKer as the primary code base for evaluating our toolset (Frama-C + AstraVer Plugin + Why3 + Solvers) on real-world code.

Rewriting lemmas as lemma functions has allowed VerKer to **fully prove** the correctness of string manipulation functions!

Each lemma function was proved automatically in a few seconds using SMT solvers.

VerKer Summary

Before / After Conversion

Function	# of Lemmas	Auto Proved	Unproved	# of Ghost Fn	# of Lemma Fn
check_bytes8	3	3	0	0	0
strlen	10	4	6	4	2
skip_spaces	7	1	6	1	3
strchr	7	4	3	0	5
strchrnul	7	5	2	0	5
strspn	8	5	3	1	5
strcspn	5	2	3	0	3
strnlen	17	11	6	3	6
strpbrk	5	2	3	0	2
Total	**69**	**37**	**32**	**9**	**31**

Future Work

Logic Types and Values in Lemma Functions

Logical functions/predicates are not C code, they are not supported during C code parsing and do not have C semantics. However we would like to express e.g. storing logical values as C variables or branching on logical predicates in lemma functions. A possible workaround is ghost proxy functions:

```
//@ ensures \result != 0 <==> pred(arr, i);
int pred(int *arr, int i);
```

Similarly, we can use proxy types for logical types:

```
/*@ axiomatic Inj {
  logic integer project(char *p);
  logic char *inject(integer i);
  axiom Inj: \forall integer i;
    project(inject(i)) == i;
} */
```

Thanks!

<gdvolkov@edu.hse.ru> <mandrykin@ispras.ru> <defremov@hse.ru>

https://forge.ispras.ru/projects/astraver