

# Proving Termination and Memory Safety for Programs with Pointer Arithmetic<sup>\*</sup>

Thomas Ströder<sup>1</sup>, Jürgen Giesl<sup>1</sup>, Marc Brockschmidt<sup>2</sup>, Florian Frohn<sup>1</sup>,  
Carsten Fuhs<sup>3</sup>, Jera Hensel<sup>1</sup>, and Peter Schneider-Kamp<sup>4</sup>

<sup>1</sup> LuFG Informatik 2, RWTH Aachen University, Germany

<sup>2</sup> Microsoft Research Cambridge, UK

<sup>3</sup> Dept. of Computer Science, University College London, UK

<sup>4</sup> IMADA, University of Southern Denmark, Denmark

**Abstract.** Proving termination automatically for programs with explicit pointer arithmetic is still an open problem. To close this gap, we introduce a novel abstract domain that can track allocated memory in detail. We use it to automatically construct a *symbolic execution graph* that represents all possible runs of the program and that can be used to prove memory safety. This graph is then transformed into an *integer transition system*, whose termination can be proved by standard techniques. We implemented this approach in the automated termination prover AProVE and demonstrate its capability of analyzing C programs with pointer arithmetic that existing tools cannot handle.

## 1 Introduction

Consider the following standard C implementation of `strlen` [23,30], computing the length of the string at pointer `str`. In C, strings are usually represented as a pointer `str` to the heap, where all following memory cells up to the first one that contains the value 0 are allocated memory and form the value of the string.

```
int strlen(char* str) {char* s = str; while(*s) s++; return s-str;}
```

To analyze algorithms on such data, one has to handle the interplay between addresses and the values they point to. In C, a violation of *memory safety* (e.g., dereferencing NULL, accessing an array outside its bounds, etc.) leads to undefined behavior, which may also include non-termination. Thus, to prove termination of C programs with low-level memory access, one must also ensure memory safety. The `strlen` algorithm is memory safe and terminates because there is some address `end`  $\geq$  `str` (an *integer property* of `end` and `str`) such that `*end` is 0 (a *pointer property* of `end`) and all addresses `str`  $\leq$  `s`  $\leq$  `end` are allocated. Other typical programs with pointer arithmetic operate on arrays (which are just sequences of memory cells in C). In this paper, we present a novel approach to prove memory safety and termination of algorithms on integers and pointers automatically. To avoid handling the intricacies of C, we analyze programs in the platform-independent intermediate representation (IR) of the LLVM compilation framework [17]. Our approach works in three steps: First, a *symbolic execution graph* is created

---

<sup>\*</sup> Supported by DFG grant GI 274/6-1 and Research Training Group 1298 (*AlgoSyn*).

that represents an over-approximation of all possible program runs. We present our abstract domain based on *separation logic* [22] and the automated construction of such graphs in Sect. 2. In this step, we handle all issues related to memory, and in particular prove memory safety of our input program. In Sect. 3, we describe the second step of our approach, in which we generate an *integer transition system* (ITS) from the symbolic execution graph, encoding the essential information needed to show termination. In the last step, existing techniques for integer programs are used to prove termination of the resulting ITS. In Sect. 4, we compare our approach with related work and show that our implementation in the termination prover AProVE proves memory safety and termination of typical pointer algorithms that could not be handled by other tools before.

## 2 From LLVM to Symbolic Execution Graphs

In Sect. 2.1, we introduce concrete LLVM states and *abstract* states that represent *sets* of concrete states, cf. [9]. Based on this, Sect. 2.2 shows how to construct symbolic execution graphs automatically. Sect. 2.3 presents our algorithm to *generalize* states, needed to always obtain *finite* symbolic execution graphs.

To simplify the presentation, we restrict ourselves to a single LLVM function without function calls and to types of the form *in* (for *n*-bit integers), *in\** (for pointers to values of type *in*), *in\*\**, *in\*\*\**, etc. Like many other approaches to termination analysis, we disregard integer overflows and assume that variables are only instantiated with signed integers appropriate for their type. Moreover, we assume a 1 byte data alignment (i.e., values may be stored at any address).

### 2.1 Abstract Domain

Consider the `strlen` function from Sect. 1. In the corresponding LLVM code,<sup>1</sup> `str` has the type `i8*`, since it is a pointer to the string's first character (of type `i8`). The program is split into the *basic blocks* `entry`, `loop`, and `done`. We will explain this LLVM code in detail when constructing the symbolic execution graph in Sect. 2.2.

```
define i32 @strlen(i8* str) {
entry: 0: c0 = load i8* str
      1: c0zero = icmp eq i8 c0, 0
      2: br i1 c0zero, label done, label loop

loop:  0: olds = phi i8* [str,entry],[s,loop]
      1: s = getelementptr i8* olds, i32 1
      2: c = load i8* s
      3: czero = icmp eq i8 c, 0
      4: br i1 czero, label done, label loop

done:  0: sfin = phi i8* [str,entry],[s,loop]
      1: sfinint = ptrtoint i8* sfin to i32
      2: strint = ptrtoint i8* str to i32
      3: size = sub i32 sfinint, strint
      4: ret i32 size }
```

Concrete LLVM states consist of the program counter, the values of local variables, and the state of the memory. The program counter is a 3-tuple  $(b_{prev}, b, i)$ , where  $b$  is the name of the current basic block,  $b_{prev}$  is the previously executed

<sup>1</sup> This LLVM program corresponds to the code obtained from `strlen` with the Clang compiler [8]. To ease readability, we wrote variables without “%” in front (i.e., we wrote “`str`” instead of “`%str`” as in proper LLVM) and added line numbers.