Here be wyverns!
Verifying LLVM bitcode with llStar

Jules Villard

University College London

Abstract. We present the `llStar` tool for the verification of programs written in bitcode, the intermediate language of the LLVM compiler infrastructure. The low-level nature of bitcode makes it a challenging target for automatic formal verification. Thanks to a novel adaptation of separation logic to bitcode's memory model, and using the symbolic execution engine `coreStar` together with `z3` as a backend, `llStar` is able to automatically analyse bitcode programs. Simple programs enjoy simple proofs and specifications, and the compositionality of the analysis allows it to scale to larger programs. Additional theories can easily be added to `llStar`, which we showcase with shape domains for linked lists and directed acyclic graphs.

1 Introduction

Developing new verification techniques applicable to realistic programs is often hampered by the intricacies of mainstream programming languages. At the same time, the way to deal with these intricacies is often the same for verification techniques that share similar formalisms. It is thus desirable to develop tools for *intermediate languages*, and with builtin support for extensions to their theories. In this paper, we focus on automatic compositional techniques to analyse programs written in the "bitcode" language, and base the reasoning about symbolic program states on an easily extensible set of *rewrite rules*. Bitcode, the intermediate representation of the LLVM compiler infrastructure [20], is a RISC-like assembly language with typed values and an infinite number of virtual registers. Many languages can be compiled to bitcode, such as C, C++, Fortran, Ada, and Java bytecode. The semantics and memory model of bitcode, close to that of C, are well-understood (and have been formalised inside the Coq proof assistant [27]), yet its low-level nature presents challenges for automatic verification, namely reasoning about low-level, untyped dynamic accesses to the memory that rely heavily on pointer arithmetic, combined with arrays and user-defined C-like structures.

This paper presents `llStar`, a tool for verifying bitcode programs. The theory of `llStar` happily deals with the low-level aspects of bitcode's memory model and operations. The analysis performed by `llStar`, the first of its kind for bitcode, is based on separation logic [24,19], which allows one to reason concisely and compositionally about heap-manipulating programs and has already given rise to several tools [12,1,17,6]. New analyses can be developed on top of `llStar` simply by adding new logic rules to its flexible proof engine, based on the `coreStar` symbolic execution tool [3]. The analysis is compositional, hence has the potential to scale to large programs, and the tool has been used to automatically verify a range of examples. Functions are annotated with pre and postconditions, and `llStar` attempts to discover loop invariants automatically. We make the following contributions:

- We give a simple symbolic state model of LLVM's memory model which, while precise enough to guarantee memory safety, enables local reasoning about the heap and a rich connection to an SMT solver to deal with values.
- We present the symbolic execution mechanism of `llStar`. In particular, we define separation logic style specifications for bitcode instructions, proof rules for entailments between symbolic states, and a translation from bitcode programs to the more abstract language accepted by `coreStar`.
- We demonstrate the extensible nature of `llStar` by proving a range of examples manipulating linked lists and directed acyclic graphs. The logic rules that deal with such inductive predicates can been added to `llStar` as separate, program-independent rules.
- The `llStar` tool and its source code are released under a BSD license. [26]

Limitations At the moment, `llStar` ignores certain features of LLVM such as exceptions, vector instructions, and concurrency. Moreover, it only considers partial correctness hence cannot guarantee termination. The theory of `llStar` is incomplete and in general cannot prove the *presence* of bugs. The price to pay for the ability to easily add new theories is one of performance compared to tools built with native support for these theories. Finally, `llStar` currently does not check soundness or termination of user-added proof rules.

Related Work A large number of tools attack the problem of the formal verification of low-level imperative languages, and in particular of C, such as Frama-C [16], VCC [10], Havoc [8], and Predator [14]. Closer cousins to `llStar` are other tools based on separation logic such as Slayer [1], Space Invader [12], Infer [6], and HIP/SLEEK [17], which perform compositional and automatic analyses, usually based on specific shape domains. Some of these domains can be imported into `llStar` as additional logic rules, as shown here for the logic of linked lists, borrowed from Space Invader [12]. Two tools in particular are also based on `coreStar` and thus closely related to `llStar`, although they analyse different languages: Botinčan et al.'s [2], for C programs, and `jStar` [13], for Java bytecode.

Other tools build directly on LLVM. `KLEE` [5] mixes static and dynamic symbolic execution to find bugs in bitcode programs. `LLBMC` [25] is a bounded model-checker for LLVM. `KITTeL` [15] is a termination prover for bitcode. While successful in their own rights, the treatment of the heap in each of these tools is somewhat unsatisfactory, either assuming safe memory accesses or sacrificing compositionality and scalability by representing it explicitly as an array of cells.

Recent works have aimed at integrating separation logic and SMT solvers. Botinčan et al. [2] use `z3` to discharge pointer arithmetic facts, but do not integrate support for other theories such as floats and records. Piskac et al. [23] and Navarro Pérez and Rybalchenko [21], on the other hand, both propose decision procedures for separation logic entailments involving arbitrary satisfaction theories. Our own integration with SMT is shallower, and we only consider theories directly relevant to LLVM values: bitvectors, floats, records, and arrays.

Outline In §2 we introduce the bitcode language and `llStar`'s architecture. In §3 we detail our symbolic execution of bitcode. We describe our experiments in §4, in particular how to extend the basic theory of `llStar` with inductive predicates for linked lists and directed acyclic graphs. We then conclude.

2 Overview

In this section, we briefly describe bitcode and the architecture of `llStar`.

2.1 Bitcode modules

Bitcode modules are LLVM's translation units: each module is compiled separately and defines its own functions, datatypes, and global variables. Functions are written in a RISC-like assembly language with an infinite number of registers of any sizes. A bitcode program is tied to a particular architecture; in particular, the sizes of datatypes and the layouts of structures in memory are fixed (but for instance the precise calling convention of functions is not). Finally, the code is always presented in Single Static Assignment form (SSA): the definition of a variable must dominate all of its uses.

Values appearing in bitcode instructions are always associated with their types, described by the following grammar, where N is a natural number:

$t ::= \texttt{i}N$	integer type
$\mid \texttt{half} \mid \texttt{float} \mid \texttt{double} \mid \ldots$	floating-point types
$\mid t\texttt{*} \mid t\ (t_1, \ldots, t_n) \mid \texttt{[}N\ \texttt{x}\ t\texttt{]} \mid \texttt{\{}t_1, \ldots, t_n\texttt{\}}$	derived types

Base types are bitvectors $\texttt{i}N$ of any size N and various kinds of floats. Derived types are, in order, pointers, functions, arrays (of size N, although the size may not actually be enforced by LLVM, in particular when it is not statically known), and structures. LLVM values follow a similar grammar (see §3.1). For instance, a 32 bit integer with value 10 is represented as `i32 10`, and a structure containing two 16 bit values 32 and 52 as `{ i16 32, i16 52 }`.

Instructions are grouped into basic blocks inside function definitions and can be categorised as follows, where `x` represents a variable name, v a value, l a label of a basic block, and f a function name:

$I ::= \texttt{unreachable}$	error
$\mid \texttt{ret}\ t\ v \mid \texttt{ret void}$ $\mid \texttt{br i1}\ v\texttt{, label}\ l_{true}\texttt{, label}\ l_{false} \mid \texttt{br label}\ l \mid \ldots$	control flow
$\mid \texttt{x =}\ bop\ t\ v_1\texttt{,}\ v_2$	binary operation
$\mid \texttt{x =}\ convop\ t_1\ v\ \texttt{to}\ t_2$	conversions
$\mid \texttt{x = icmp}\ cond\texttt{,}\ t\ v_1\texttt{,}\ v_2 \mid \texttt{x = fcmp}\ cond\texttt{,}\ t\ v_1\texttt{,}\ v_2$	comparisons
$\mid \texttt{x = alloca}\ t \mid \texttt{x = load}\ t\texttt{*}\ v \mid \texttt{store}\ t\ v\texttt{,}\ t\texttt{*}\ v_p$	memory manipulation
$\mid \texttt{x = getelementptr}\ t\texttt{*}\ v\texttt{,}\ t_1\ v_1\texttt{, ...,}\ t_n\ v_n$	address calculation
$\mid \texttt{x = phi}\ t\ \texttt{[}v_1\texttt{,}\ l_1\texttt{], ..., [}v_n\texttt{,}\ l_n\texttt{]}$	phi node
$\mid \texttt{x = call}\ t\ f\texttt{(}v_1\texttt{, ...,}v_n\texttt{)}$	function call

Attempting to execute `unreachable` in any context results in an error. Returns, conditional and unconditional branching are standard. A binary operation *bop* can be an operation between integers (`add`, `sub`, ...), between floats (`fadd`, `fsub`, ...) or a bitwise logical operation (`and`, `or`, ...). Conversions can occur between bitvectors or floats of different sizes, between floats and bitvectors, or between pointer values and bitvectors (thus allowing pointer arithmetic). Comparisons between bitvectors or floats return a boolean (a value of type `i1`, *i.e.*, a single bit) representing the truth value of the comparison (`eq` for equality, `ugt` for unsigned greater than, ...). Stack memory is allocated with `alloca`, and `load` and `store` are the usual memory operations. Functions calls are also standard. Let us now focus our attention on the address calculation and the phi node instructions.

The `getelementptr` instruction computes offsets from a base pointer inside an aggregate type (a structure or array type) according to a sequence of “hops” starting from that address. Each hop adds an offset to the pointer based on the type reached at the previous hop. For instance, if the type is that of a structure, then the hop number corresponds to one of its fields, and the offset is that of this field; in the case of an array (or, equivalently for LLVM, a pointer), the hop is an index into the array and the additional offset is the offset of the element at that index in the array. See Fig. 4 for rules to evaluate these values.

A `phi` instruction of the form “`x = phi` t `[`v_1`,` l_1`], ..., [`v_n`,` l_n`]`” sets `x` to the value v_i if the control arrives at the current basic block from a previous block labelled by l_i (where l_1 to l_n are the labels of all the predecessors of the current basic block). Phi nodes are a standard ingredient of programs in SSA form.

We have omitted from this presentation a few datatypes and instructions of LLVM that are currently not supported by `llStar`, namely vectors, function and label pointers, `undef` values, variable argument functions, exception handling, concurrency, inline assembly, and many LLVM intrinsics.

Example 1. In the module of Fig. 1, we define the (recursive) type `%nd` with two fields: a 32 bits integer and a pointer to a `%nd`, and a function `list_dispose` that assumes a linked list starting at `%l` and disposes its elements one by one. Compiling the C program on the right-hand side of Fig. 1 typically results in this bitcode module (*e.g.* using `clang -O2`). If `%l` is nil then we jump straight to the exit point, else we enter the loop. The current head of the list is pointed to by `%hd`. The tail of the list is pointed to by the second field of `%hd`, whose address is computed using `getelementptr` (on typical 64 bit architectures we would get `%p` = `%hd` + 8), and its value loaded into `%tl`. The current head is then cast into the type expected by `free` and disposed of. Finally, the new `%tl` pointer is compared to nil and we branch accordingly. If the loop is taken again, `%hd` gets assigned to `%tl`.

2.2 Architecture of `llStar`

The architecture of `llStar`, shown in Fig. 2, comprises the following elements:

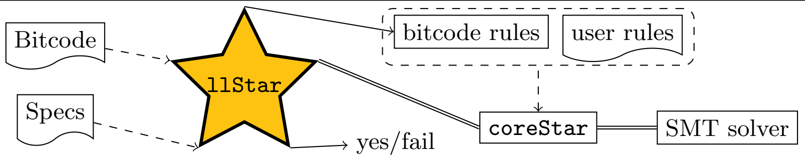
Bitcode This is the bitcode module to analyse. `llStar` uses LLVM to parse this file and to retrieve the memory layout of the data structures defined in

Figure 1 A bitcode module and some equivalent C code.

```
%nd = type { i32, %nd* }
define void @list_dispose(%nd* %l){
go: %c = icmp eq %nd* %l, null
 br i1 %c, label %fin, label %loop
loop: %hd = phi %nd* [%tl,%loop],[%l,%go]
 %p = getelementptr %nd* %hd,i64 0,i32 1
 %tl = load %nd** %p
 %hd8 = bitcast %nd* %hd to i8*
 call void @free(i8* %hd8)
 %d = icmp eq %nd* %tl, null
 br i1 %d, label %fin, label %loop
fin: ret void }
declare void @free(i8*)
```

```
#include <stdlib.h>
typedef struct s {
 int data;
 struct s* next; }
 nd;
void list_dispose
          (nd* l){
 nd *hd=l, *tl=l;
 while(tl) {
   hd = tl;
   tl = tl->next;
   free(hd); }
}
```

Figure 2 The Architecture of `llStar`. Wavy boxes represent user input; double lines represent tool interaction, dashed lines inputs and solid lines outputs.



the file (*e.g.* the offsets and paddings of fields inside structures). Currently, `llStar` is based on the latest stable version of LLVM (version 3.3).

Specs This contains specs for functions defined and used in the input program.

`coreStar` `coreStar` is an open-source symbolic execution and frame inference engine for separation logic. Bitcode programs are translated into `coreStar`'s generic language of assignments (defined by their specs), gotos and labels (see §3.2). We include the latest development version of `coreStar` in `llStar`.

bitcode rules When performing symbolic execution of programs, `coreStar` needs to manipulate the formula describing the current state. The base theory of `coreStar` is agnostic in the predicates used to describe symbolic states. To cope with user-defined predicates (such as our pointsto and malloced above), additional *logic rules* have to be given to `coreStar`. `llStar` generates the rules needed to analyse a bitcode module automatically, for instance to unfold pointers to a structure into pointers to each field of the structure (see §3.3).

user rules Users of `llStar` may define their own set of rules on top of `llStar`. In §4.1, we give examples of such rules to reason about singly-linked lists and directed acyclic graphs. The soundness and termination of the system of rules are currently the responsibility of the user.

SMT solver At many points during the symbolic execution, queries are made to an SMT solver by `coreStar` to solve entailments between values. We have added support for translating the datatypes needed by `llStar` into SMT queries to `coreStar` (see §3.4). Our SMT solver of choice is `z3` [11] (version 4.3.2), which supports all the needed theories.

Usage Given a bitcode program and function specifications (pairs of pre and postconditions) as input, `llStar` uses `coreStar` to perform symbolic execution and prove that the program is memory safe (there are no null pointer dereferences, invalid accesses to memory, double frees, or memory leaks) and *partially correct* with respect to the given spec: from all states satisfying the given precondition, *if* the program terminates then its final states satisfy the postcondition. The final output of the tool is either a proof of the program properties or an error (due either to a bug in the program or to an incompleteness in `llStar`).

We can for instance give the following spec to Ex. 1, which states that it will consume a single pointer to a `nd` whose second field is nil:

`list_dispose(l):` $\{\mathtt{l} \overset{\mathtt{nd}}{\longmapsto} \{v', \mathtt{null}\}\}\{\mathsf{emp}\}$

By default, `llStar` will be able to generate enough logic rules from the bitcode of Ex. 1 to understand *e.g.* that $\mathtt{l} + 8$ points to the second field of `l`, and thus will be able to prove `list_dispose` correct w.r.t. this spec[1]. However, it will not be able to prove that the program works on nil-terminated linked lists of any size, and the reason is that `llStar` has no built-in support for such inductive predicates. Instead, the user has to supply `llStar` with additional rules for reasoning about linked lists (see §4.1). With appropriate rules, `llStar` can prove the richer specification $\{\mathsf{lseg}(\mathtt{nd}, \mathtt{l}, \mathtt{null})\}\{\mathsf{emp}\}$, *i.e.* `dispose_list` takes any linked list of nodes of type `nd` terminated by `null` and consumes it. Omitting the call to `free`, calling `free` twice, or lowering the `load` after `free` would violate any of the two specifications (resp. it would leak memory, free the same pointer twice, or dereference after `free`), which `llStar` would report as errors.

3 Symbolic execution of bitcode

In this section, we describe how `llStar` performs symbolic execution on bitcode programs. We begin with a description of the symbolic states used to represent the current state of the execution.

3.1 Symbolic States

Value and type expressions handled by `llStar` include those of LLVM, with the addition of existentially quantified variables, and expressions that correspond to sizes of types or to the evaluation of bitcode operations on values, such as additions, conversions, comparisons, and address calculations. Formally, `llStar` types and expressions can be described by the grammars below, where N, n and i are integer constants, primed variables x' represent existentials, fpt is a floating point type, fp a floating point number, and cond, $\mathit{conv}_t^{t'}$ and bop are respectively condition evaluations, conversions (from type t to t') and binary operations as described in §2.1:

[1] In fact, as explained at the end of §3.2, the precondition must also require that `l` was allocated on the heap by `malloc` – and thus can be `free`'d.

$$\begin{aligned}e &::= x' \mid \mathtt{i}N\ \ n \mid \mathit{fpt}\ \mathit{fp} \mid [e_1, \ldots, e_n] \mid e_a[e_i] \mid e_a[e_i := e_v] \mid \{e_1, \ldots, e_n\} \mid \mathit{field}^i_t(e)\\ &\mid \mathit{eltptr}(e, t, [e_1; \ldots; e_n]) \mid \mathit{sizeof}(t) \mid \mathit{cond}(e_1, e_2) \mid \mathit{conv}^{t'}_t(e) \mid \mathit{bop}(e_1, e_2)\\ t &::= \mathtt{i}N \mid \mathtt{half} \mid \ldots \mid \mathtt{double} \mid t\mathtt{*} \mid \mathtt{[}e\ \mathtt{x}\ t\mathtt{]} \mid \{t_1, \ldots, t_n\}\end{aligned}$$

Symbolic states consist of separation logic formulas of a particular shape, as supported by `coreStar`, described by the following grammar:

$$\begin{aligned}A ::= &\ \mathsf{emp} \mid e_p \overset{t}{\mapsto} e_v \mid \mathsf{malloced}(e_p, e_s) \mid \ldots && \text{spatial predicates}\\ &\mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 >_{bv} e_2 \mid e_1 \leqslant_{fp} e_2 \mid \ldots && \text{pure predicates}\\ &\mid A_1 * A_2 \mid A_1 \vee A_2 && \text{connectives}\end{aligned}$$

The emp predicate denotes the empty heap (where nothing is allocated); $e_p \overset{t}{\mapsto} e_v$ denotes a heap such that only cells at addresses $e_p, \ldots, e_p + \mathit{sizeof}(t) - 1$ are allocated and they form a value e_v of type t; $\mathsf{malloced}(e_p, e_s)$ indicates that cells $e_p, \ldots, e_p + e_s - 1$ form a heap-allocated block. Pure predicates implicitly denote an empty heap and are equalities, disequalities, and (typed) comparisons between expressions. The connectives are standard; the separating conjunction $*$ is that of separation logic: a memory state satisfies $A_1 * A_2$ if it can be split into two *disjoint* (with respect to their domains) heaps h_1 and h_2 such that $h_1 \vDash A_1$ and $h_2 \vDash A_2$. On *pure* formulas (those made from pure predicates and connectives only), $*$ coincides with the classical conjunction $\wedge$. While the base fragment of `llStar` above does not include, *e.g.*, inductive predicates, users of `llStar` are encouraged to define their own (see §4.1), using `coreStar`'s support for arbitrary *abstract predicates* [22].

To illustrate our assertion language, in particular the use of malloced, let us present the specs of `malloc` and `free`, as defined in `llStar`'s distribution:

$\mathtt{malloc(s):}\ \ \{\mathsf{emp}\}\{(\mathsf{malloced}(\mathtt{ret},\mathtt{s}) * \mathtt{ret} \xrightarrow{\mathtt{[s\ x\ i8]}} v') \vee \mathtt{ret} = \mathtt{null}\}$

$\mathtt{free(x):}\ \ \{\mathsf{malloced}(\mathtt{x},\mathtt{s'}) * \mathtt{x} \xrightarrow{\mathtt{[s'\ x\ i8]}} v'\}\{\mathsf{emp}\}$

Nothing is required in the precondition of `malloc` (emp), and it returns either a pointer to an uninitialised array of bytes of the appropriate size and a `malloced` predicate, or the `null` pointer. The spec of `free` does the opposite.

3.2 Bitcode axiomatisation

Target language Let us now define the axiomatic semantics of bitcode instructions over symbolic states. We use `coreStar` as our symbolic execution engine, which supports three types of instructions: *specification assignments*, labels, and non-deterministic gotos between several labels, according to the following grammar:

$$\mathit{CI} ::= \mathtt{x} = \{A\}\{B\}(e_1, \ldots, e_n) \mid \mathtt{label}(l) \mid \mathtt{goto}(l_1, \ldots, l_n)$$

where l, $l_1, \ldots, l_n$ are labels (represented by strings), `x` is a program variable, A and B are formulas respectively representing the pre and post-condition of the assignment, and e_1 to e_n are expressions to be assigned to the parameters of the specification. Special variables `ret` and $\mathtt{p}_1$ to $\mathtt{p}_n$ can appear free in A and B and represent the return value that will be assigned to `x` and the parameters of the assignment (in our case, parameters will only be used to interpret function calls).

Figure 3 Axioms for bitcode instructions.

$\textsc{Unreachable} : \Gamma, \texttt{unreachable} \rightsquigarrow \{\mathit{false}\}\{\mathsf{emp}\}$

$\textsc{BinaryOps} : \Gamma, \texttt{x = } \mathit{bop}\ t\ v_1\texttt{,}\ v_2 \rightsquigarrow \texttt{x} = \{\mathsf{emp}\}\{\texttt{ret} = \mathit{bop}(v_1, v_2)\}$

$\textsc{ConvOps} : \Gamma, \texttt{x = } \mathit{convop}\ t_1\ v\ \texttt{to}\ t_2 \rightsquigarrow \texttt{x} = \{\mathsf{emp}\}\{\texttt{ret} = \mathit{conv}_{t_1}^{t_2}(v)\}$

$\textsc{Cond} : \Gamma, \texttt{x = }\{\texttt{i},\texttt{f}\}\texttt{cmp}\ \mathit{cond}\texttt{,}\ t\ v_1\texttt{,}\ v_2 \rightsquigarrow \texttt{x} = \{\mathsf{emp}\}\{\texttt{ret} = \mathit{cond}(v_1, v_2)\}$

$\textsc{GetEltPtr} : \Gamma, \texttt{x = getelementptr}\ t\texttt{*}\ v\texttt{,}\ t_1\ v_1\texttt{, ..., }t_n\ v_n$
$\rightsquigarrow \texttt{x} = \{\mathsf{emp}\}\{\texttt{ret} = \mathit{eltptr}(v, t*, [v_1; \ldots; v_n])\}$

$\textsc{Alloca} : \Gamma, \texttt{x = alloca}(t) \rightsquigarrow \texttt{x} = \{\mathsf{emp}\}\{\texttt{ret} \overset{t}{\mapsto} x'\}$

$\textsc{Ret} : \Gamma, \texttt{ret}\ t\ v \rightsquigarrow \texttt{ret} = \{\mathit{ALLOCA}\}\{\texttt{ret} = v\}$

$\textsc{Load} : \Gamma, \texttt{x = load}\ t\texttt{*}\ v \rightsquigarrow \texttt{x} = \{v \overset{t}{\mapsto} e\}\{\texttt{ret} = e * v \overset{t}{\mapsto} e\}$

$\textsc{Store} : \Gamma, \texttt{store}\ t\texttt{*}\ v_1\ v_2 \rightsquigarrow \{v_1 \overset{t}{\mapsto} e\}\{v_1 \overset{t}{\mapsto} v_2\}$

$\textsc{Call} : \Gamma, \texttt{x = call}\ t\ f\texttt{(}v_1\texttt{, ...,}v_n\texttt{)} \rightsquigarrow \texttt{x} = \{A\}\{B\}(v_1, \ldots, v_n)$

$\textsc{UncondBr} : \Gamma, \texttt{br label}\ l \rightsquigarrow \mathtt{goto}(l)$

When `ret` does not appear in the specification, we omit $\texttt{x} =$ from the syntax, and similarly for parameters $\mathtt{p}_i$. Specification assignments may also $n > 1$ return values, in which case they are written $\texttt{ret}_1$ to $\texttt{ret}_n$. Let us briefly review how `coreStar` interprets this language. See [3] for a more in-depth description.

Instructions are arranged in a control flow graph (CFG) that accounts for sequential composition between instructions and connects gotos to their corresponding labels. Executing $\mathtt{goto}(l_1, \ldots, l_n)$ non-deterministically jumps to any of the labels l_1 to l_n.

To execute an assignment $\texttt{x} = \{A\}\{B\}(e_1, \ldots, e_n)$ in a given symbolic state C, `coreStar` looks for a solution to the *frame problem* between C and A, *i.e.* for a formula F such that the following implication holds:

$$C \vdash A[\mathtt{p}_1, \ldots, \mathtt{p}_n \leftarrow e_1, \ldots, e_n] * F$$

If a frame F is found, the state resulting from the execution of the command is

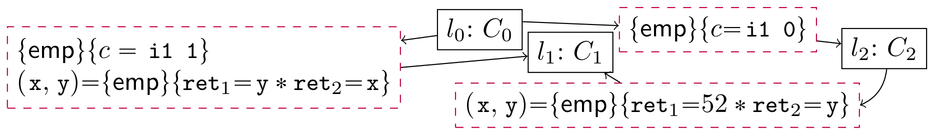
$$(B[\mathtt{p}_1, \ldots, \mathtt{p}_n \leftarrow e_1, \ldots, e_n] * F)[\texttt{x}, \texttt{ret} \leftarrow x', \texttt{x}]$$

for some fresh existential variable x' (to account for the update to the value of `x`). Absence of a solution to the frame problem can be due either to an incompleteness in the proof rules or to a bug in the program, and is reported as an error.

Most of the bitcode instructions are directly translated by `llStar` into specification assignments, as shown in Fig. 3. The translation assumes a context Γ mapping function names to their specifications. The error instruction `unreachable` is given an inconsistent precondition *false*. Rules BinaryOps, ConvOps, Cond and GetEltPtr convert value manipulations into the corresponding expressions. Rule Alloca allocates a pointer of the corresponding type on the stack. Such pointers have the same status as pointers allocated on the heap (as returned by `malloc`) but lack the additional resource that would allow them to be `free`'d. At the exit points of a function, the current memory state typically contains stack-allocated pointers that have to be reclaimed. When translating a function, `llStar` accumulates resources allocated by `alloca` into a formula ALLOCA that is consumed by `ret` instructions. The remaining rules are straightforward.

The remaining instructions: conditional `br` and `phi`, are implemented as extra nodes in the CFG, placed between source and target blocks. Between the source block and the first (resp. second) destination block of a `br` instruction, the condition is assumed to be true (resp. false). Coming from a predecessor block, variables set by `phi` instructions get the appropriate values. We illustrate these transformations on the three blocks below and their associated CFG:

l_0: C_0;	l_1: `x = phi i32 [%y,`l_0`] [52,`l_2`];`	l_2: C_2
`br i1` c`,label` l_1`,`l_2	`y = phi i32 [%x,`l_0`] [%y,`l_2`];` C_1	


$\{\mathsf{emp}\}\{c = \texttt{i1 1}\}$
$(\texttt{x}, \texttt{y}) = \{\mathsf{emp}\}\{\texttt{ret}_1 = \texttt{y} * \texttt{ret}_2 = \texttt{x}\}$
l_0: C_0
l_1: C_1
$\{\mathsf{emp}\}\{c = \texttt{i1 0}\}$
l_2: C_2
$(\texttt{x}, \texttt{y}) = \{\mathsf{emp}\}\{\texttt{ret}_1 = 52 * \texttt{ret}_2 = \texttt{y}\}$

3.3 Frame inference for symbolic states

As mentioned in §2.2, `coreStar` does not know how to manipulate predicates used to describe symbolic states on its own. When faced with a frame inference problem between formulas A and B, `coreStar` repeatedly rewrites the sequent $A \vdash B$ using *logic rules* given by `llStar` (and the user, see §4.1), in order, until it can be reduced to $F \vdash \mathsf{emp}$ (at which point F is a solution to the frame problem [3]), or until no further rule can be applied (in which case the proof fails). In this section, we describe the set of rules handled by `llStar` to analyse bitcode modules. Some of these rules will be the same for every module, while others will be generated depending on the parameters of the module (in particular the size and layout of data types and the structures it defines).

Each logic rule accepted by `coreStar` describes patterns to be syntactically matched by `coreStar` in the current formula or sequent. Patterns can use *pattern variables*, prefixed by a question mark (*e.g.* `?x`), which match any expression. Logic rules can be of one of the following three forms:

– *Rewrite rules* $e_1 = e_2$: If an expression in a formula matches e_1 then the equality is added to it.
– *Equivalence rules* $A \Leftrightarrow B$: Every sub-formula matching A is changed into B.
– *Sequent rules*
$$\frac{A_2 \vdash B_2}{A_1 \vdash B_1} \qquad \text{or} \qquad \frac{A_2 \vdash B_2}{A_1 \vdash B_1}\, C$$
They allow `coreStar` to rewrite the current goal if it is of the form $AF * A_1' \vdash B_1' * F$ where A_1' and B_1' match A_1 and B_1 respectively. The extra bits of state are implicitly carried over, unchanged, to the premise. The optional side-condition C is a *pure* formula (one without spatial predicates) that is passed to the SMT solver; if present, the rule can only trigger if C is satisfied.

Logic rules work towards matching spatial predicates from the right-hand side formula (RHS) of a sequent with predicates on the left-hand side formula (LHS). If this can be done for the whole spatial part of the RHS, then all that remains is to check implication between two pure formulas, which can be discharged by the SMT solver. The spatial logic rules of `llStar` are of three sorts: (1) rules that unfold aggregates (arrays and structures) on the LHS into field-wise pointers when trying to match a sub-part of that aggregate on the RHS, so as to expose a matching pointer on the LHS; (2) rules that match pointers of possibly different types (*e.g.* a byte array returned by `malloc` and the corresponding structure pointer); (3) rules that fold fragmented aggregates into a single pointer. Rules of the first two kinds are always tried before rules of the third kind, so as to avoid infinite proof attempts where aggregates get repeatedly folded and unfolded. Other choices of rules are of course possible, for instance choosing to systematically unfold instead of folding structures. In practice, we found the current set of rules to strike a good balance between precision and performance. For instance, arriving at the `load` instruction in Ex. 1 from the first specification (given in §2.2), we must prove *e.g.* $A = \texttt{\%hd} + 8 \xmapsto{\texttt{nd*}} v'$ from a heap containing $\texttt{\%hd} \xmapsto{\texttt{nd}} \{i', \mathsf{null}\}$. This will be (1) unfolded into the two fields of `nd`, at which point (2) A will be matched (instantiating $v' = \texttt{null}$). After the `load`, (3) the two fields are collated back into the original structure pointer.

Fig. 4 presents the main rewrite rules used by `llStar`. Size expressions are replaced by concrete values given by LLVM, and `null` is replaced by a nil bitvector of the appropriate size. Similarly, *eltptr* is rewritten using the concrete offsets inside structures and arrays; since array sizes do not matter for address calculation, array-based *eltptr* values are rewritten into pointer-based ones. Equivalence rules convert comparison operations whose truth values are known into the corresponding predicates. An allocated `null` pointer predicate is unsatisfiable (rule NullPtr). Spatial predicates on the right-hand side of a sequent can be removed if they are matched on the left-hand side according to RemoveMalloc, RemovePtr and RemovePtrEq. The latter asks the SMT solver if the roots are equal (in which case the equality is added to the LHS for later reuse), while RemovePtr triggers when `coreStar` can immediately deduce root equality from the current state. Both rules then require the values pointed to to be equal. The second rule is useful when pointer equality stems from bitvector arithmetic or record or array theory facts. It is given to `coreStar` in a lower priority order so that it is seldom called unduly. Similarly, two kinds of unfolding rules for structures are generated: UnfoldStruct handles common cases where the RHS requires exactly one of the fields of the structure, while UnfoldStructInner is more general (and costly) and matches pointers to anywhere contained in a field (for instance to match parts of a nested sub-structure). Rules ByteArrayToStruct and TypeFromByteArray are examples of reinterpretations of the types of values stored in memory, in these cases between arrays of bytes and pointers to types of the same size. Both give a more precise type to the byte array, guided by the current goal pointer (and in the case of the former, by the fact that its address was derived using a jump inside type *st*). The last rule folds back the pointers forming a structure into a single structure pointer.

3.4 Pure entailments

Entailments between pure formulas, and side conditions of sequent rules are translated by `coreStar` into SMT formulas passed to the `z3` SMT solver. We have

Figure 4 Selected builtin rules. Given an LLVM type t, $SZ(t)$ is the concrete size of t. Given an LLVM structure type st, N_{st} is the number of fields in the structure, t_i is the type of the i-th field, and $O(st, i)$ its offset. Rules parameterised by a type are generated for all types in a module. Rules parameterised by indices in a structure are generated for all fields in that structure.

$$\mathit{sizeof}(t) = \mathit{SZ}(t) \qquad \mathit{sizeof}(\texttt{[?n x ?t]}) = \texttt{?n} * \mathit{sizeof}(\texttt{?t}) \qquad \mathit{eltptr}(\texttt{?x}, \texttt{?t}, []) = \texttt{?x}$$

$$\begin{array}{ll} \mathit{eltptr}(\texttt{?x}, \texttt{[?n x ?t]}, \texttt{?j}) & \mathit{eltptr}(\texttt{?x}, \texttt{?t*}, \texttt{?n} :: \texttt{?j}) \\ = \mathit{eltptr}(\texttt{?x}, \texttt{?t*}, \texttt{?j}) & = \mathit{eltptr}(\texttt{?x} + \texttt{?n} * \mathit{sizeof}(\texttt{?t}), \texttt{?t}, \texttt{?j}) \end{array}$$

$$\mathit{eltptr}(\texttt{?x}, st, i :: \texttt{?j}) = \mathit{eltptr}(\texttt{?x} + O(st, i), t_i, \texttt{?j}) \qquad \texttt{null} = \texttt{i}(\mathit{SZ}(\texttt{i8*}))\ \texttt{0}$$

$$\mathit{eq}(\texttt{?x}, \texttt{?y}) = \texttt{i1}\ 0 \Leftrightarrow \texttt{?x} \neq \texttt{?y} \qquad \mathit{bvugt}(\texttt{?x}, \texttt{?y}) = \texttt{i1}\ 1 \Leftrightarrow \texttt{?x} >_{bv} \texttt{?y}$$

$$\textsc{NullPtr}\ \frac{}{\texttt{null} \xmapsto{\texttt{?t}} \texttt{?v} \vdash} \qquad \textsc{RemoveMalloc}\ \frac{\vdash \texttt{?n} = \texttt{?m}}{\mathsf{malloced}(\texttt{?x}, \texttt{?n}) \vdash \mathsf{malloced}(\texttt{?x}, \texttt{?m})} \qquad \textsc{RemovePtr}\ \frac{\vdash \texttt{?v} = \texttt{?w}}{\texttt{?x} \xmapsto{\texttt{?t}} \texttt{?v} \vdash \texttt{?x} \xmapsto{\texttt{?t}} \texttt{?w}}$$

$$\textsc{UnfoldStruct}_j(st)\ \frac{\mathop{\ast}_{i=0}^{N_{st}} \texttt{?x} + O(st, i) \xmapsto{t_i} \mathit{field}^i_{st}(\texttt{?v}) \vdash \texttt{?x} + O(st, j) \xmapsto{t_j} \texttt{?w}}{\texttt{?x} \xmapsto{st} \texttt{?v} \vdash \texttt{?x} + O(st, j) \xmapsto{t_j} \texttt{?w}}$$

$$\textsc{UnfoldStructInner}_j(st)\ \frac{\mathop{\ast}_{i=0}^{N_{st}} \texttt{?x} + O(st, i) \xmapsto{t_i} \mathit{field}^i_{st}(\texttt{?v}) \vdash \texttt{?y} \xmapsto{\texttt{?t}} \texttt{?w}}{\texttt{?x} \xmapsto{st} \texttt{?v} \vdash \texttt{?y} \xmapsto{\texttt{?t}} \texttt{?w}} \begin{array}{l} \texttt{?x} + O(st, j) \leqslant_{bv} \texttt{?y} \ast \\ \texttt{?y} + \mathit{sizeof}(\texttt{?t}) \leqslant_{bv} \texttt{?x} + O(st, j+1) \end{array}$$

$$\textsc{ByteArrayToStruct}(st)\ \frac{\texttt{?x} \xmapsto{st} \mathit{conv}^{st}_{\texttt{[}SZ(st)\texttt{ x i8]}}(\texttt{?v}) \vdash \texttt{?y} \xmapsto{\texttt{?t}} \texttt{?w}}{\texttt{?x} \xmapsto{\texttt{[}SZ(st)\texttt{x i8]}} \texttt{?v} \ast \texttt{?y} = \mathit{eltptr}(\texttt{?x}, st, \texttt{?j}) \vdash \texttt{?y} \xmapsto{\texttt{?t}} \texttt{?w}} \qquad \textsc{RemoveEqPtr}\ \frac{\texttt{?x} = \texttt{?y} \vdash \texttt{?v} = \texttt{?w}}{\texttt{?x} \xmapsto{\texttt{?t}} \texttt{?v} \vdash \texttt{?y} \xmapsto{\texttt{?t}} \texttt{?w}}\ \texttt{?x} = \texttt{?y}$$

$$\textsc{TypeFromByteArray}(t)\ \frac{\texttt{?x} \xmapsto{t} \mathit{conv}^{t}_{\texttt{[}SZ(t)\texttt{ x i8]}}(\texttt{?v}) \vdash \texttt{?x} \xmapsto{t} \texttt{?w}}{\texttt{?x} \xmapsto{\texttt{[}SZ(t)\texttt{ x i8]}} \texttt{?v} \vdash \texttt{?x} \xmapsto{t} \texttt{?w}} \qquad \textsc{FoldStructL}(st)\ \frac{\texttt{?x} \xmapsto{st} \texttt{\{} \texttt{?v}_0, \ldots, \texttt{?v}_{N_{st}} \texttt{\}} \vdash}{\mathop{\ast}_{i=0}^{N_{st}} \texttt{?x} + O(st, i) \xmapsto{t_i} \texttt{?v}_i \vdash}$$

extended this translation, which previously only dealt with mathematical integer arithmetic, to handle the theories of bitvector and floating point arithmetic, records, and arrays, which respectively correspond to LLVM integers and floats, structure values and array values. Let us detail the translation of structure values. To each structure type corresponds a particular record type whose size matches the number of fields of the structure. A structure value `{`e_1, …, e_n`}` of type st is then translated into a constructor for the corresponding record type, applied to the translations of e_1 to e_n, and $\mathit{field}^i_{st}(e)$ is translated into the corresponding record selector. For instance, at the beginning of the analysis of Ex. 1, `llStar` declares a new record type `nd` with constructor `mk_nd` and two selectors `nd_fld0` and `nd_fld1` to represent values of the `%nd` type, which gets sent to `z3`:

```
(declare-datatypes () ((nd (mk_nd (nd_fld0 (_ BitVec 32))
                                  (nd_fld1 (_ BitVec 64))))))
```

The translations of other values are straightforward, with one exception: we have to account for the fact that pointer arithmetic operations do not overflow as long as the computed addresses fall within allocated parts of the memory. This is expressed by the following rule:

$$\frac{\texttt{?x} \xmapsto{\texttt{?t}} \texttt{?v} * \texttt{?x} +_{bv} \mathit{sizeof}(\texttt{?t}) >_{bv} \texttt{?x} \vdash}{\texttt{?x} \xmapsto{\texttt{?t}} \texttt{?v} \vdash}$$

Finally, to minimise clutter, some types can be omitted in `llStar` formulas. We built a type inference engine to guess missing types before formulas are passed to the SMT solver, which expects constants to be declared with their types.

4 Analysing bitcode programs with `llStar`

In this section, we show how `llStar` can be easily extended with additional theories and list further experiments.

4.1 Extending `llStar` with new theories

Data structures are often axiomatised by logic rules that are independent from the exact structures used to form each node. For instance, the theory of singly-linked list can be written using only $\mathsf{lseg}(s, x, y)$ and $\mathsf{sllnode}(s, x, n)$ predicates, which denote respectively a list segment from location x to location y and an individual element of the list at address x, abstracting for instance from which field of x contains the value n to the next pointer. The variable s is used to distinguish between lists of different types. Rules of this form are desirable because they are module-independent and thus can be written once and for all for each theory.

To facilitate this style `llStar` provides a $t(i_1, \ldots, i_n) = \mathsf{node}$ directive, indicating that structures of type t correspond to predicates node with $n + 2$ arguments: the structure type t, the address of the structure and the values of fields i_1 to i_n. The logic rules to link pointers to abstract nodes are then generated by `llStar`. For instance, to prove the list specification of Ex. 1, we declare $\texttt{nd}(1) = \mathsf{sllnode}$. Users can also write their own "glue", *e.g.* for nodes of a special shape.

We were able to directly import the standard shape domain for singly-linked list segments of Distefano et al. [12] as logic rules for `llStar` in this fashion (similarly to `jStar`). We then proved a range of examples on linked lists (see §4.2).

We have also developed a minimalistic theory (which does not include abstraction rules used to normalise formulas when trying to find a loop invariant) for binary directed acyclic graphs (dags), based on ramification [18], which we used to prove programs similar to the recursive dag marking of Fig. 5.[2] Let us describe its main rules briefly. The current theory reasons about $\mathsf{dag}(s, x, d)$ and

[2] For now, the theory is only able to handle programs that do not modify the link structure of the dag. Lifting this restriction would involve tracking the memory footprint of predicates, to be able to apply more general theorems from [18].

Figure 5 Marking a dag.

```
typedef struct ds{int m;
 struct ds *l, *r;} dn;
void mark(dn *x) {
 if (!x || x->m) return;
 x->m = 1;
 mark(x->l);mark(x->r);}
```

$\mathsf{dagnode}(s, x, l, r)$ predicates, where x is the root of the dag or dag node, d is an abstract value recording the link structure of the dag (defined by a relation $mdag(d, x) = (l, r)$), and l and r are the left and right children of the node x. We are able to prove for instance that `mark` satisfies the spec $\{\mathsf{dag}(\mathtt{dn}, \mathtt{x}, d')\}\{\mathsf{dag}(\mathtt{dn}, \mathtt{x}, d')\}$, which expresses that the link structure of the dag rooted at `x` does not change. Crucially, a non-empty dag unfolds into a node and an *overlapping conjunction* $\circledast$ of its two subdags, which denotes that they occupy potentially overlapping regions of the heap. This is used in the rule below, triggered when dereferencing `x` in Fig. 5:

$$\frac{\begin{matrix}\mathsf{dagnode}(\mathtt{?s},\, \mathtt{?x}, l', r') * (\mathsf{dag}(\mathtt{?s}, l',\, \mathtt{?d}) \circledast \mathsf{dag}(\mathtt{?s}, r',\, \mathtt{?d})) \\ *mdag(\mathtt{?d},\, \mathtt{?x}) = pair(l', r')\end{matrix} \vdash \mathtt{?x} \xmapsto{\mathtt{?t}} \mathtt{?v}}{\mathsf{dag}(\mathtt{?s},\, \mathtt{?x},\, \mathtt{?d}) * \mathtt{?x} \neq \mathtt{null} \vdash \mathtt{?x} \xmapsto{\mathtt{?t}} \mathtt{?v}}$$

Isolating one of the subdags from an overlapping conjunction, *e.g.* to mutate the values of its nodes, or to make recursive calls to `mark`, leaves an incomplete dag structure expressed using the existential magic wand $-\!\circledast$ of separation logic[3]:

$$\frac{\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) -\!\circledast (\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) \circledast \mathsf{dag}(\mathtt{?s},\, \mathtt{?r},\, \mathtt{?dy})) \vdash}{\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) \circledast \mathsf{dag}(\mathtt{?s},\, \mathtt{?r},\, \mathtt{?dy}) \vdash \mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?d})}$$

Another rule eliminates $-\!\circledast$ to recreate the original overlapping conjunction, triggered *e.g.* after each recursive call to `mark` in Fig. 5:

$$\frac{\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) \circledast \mathsf{dag}(\mathtt{?s},\, \mathtt{?r},\, \mathtt{?dy}) \vdash}{\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) * (\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) -\!\circledast (\mathsf{dag}(\mathtt{?s},\, \mathtt{?l},\, \mathtt{?dx}) \circledast \mathsf{dag}(\mathtt{?s},\, \mathtt{?r},\, \mathtt{?dy}))) \vdash}$$

The soundness of these rules is based on general facts about dag [18]. At the time of this writing, `coreStar` only allows $*$ and $\vee$ to compose formulas, so $\circledast$ and $-\!\circledast$ are currently represented by abstract predicates whose arguments are formulas rewritten as "punned" terms. This may change in future versions.

4.2 Experiments

Tab. 1 lists some of our experimental results. The programs analysed come from several languages: C, C++ (compiled using `clang`) and Fortran (compiled using `gcc` with the `dragonegg` plugin), illustrating the advantage of working at the level of an intermediate language like bitcode. The first example is a collection of simple functions that access individual fields in dynamically allocated structures, sometimes disposing them at the end. The second example showcases pointer arithmetic in C. The list examples consist of one loop each, which respectively traverses a linked list, disposes it node by node, and reverses a list in place. Their specs are all straightforward to express (*e.g.* `list_reverse(x)`: $\{\mathsf{lseg}(\mathtt{nd}, \mathtt{x}, \mathtt{null})\}\{\mathsf{lseg}(\mathtt{nd}, \mathtt{ret}, \mathtt{null})\}$),

[3] A state satisfies $A -\!\circledast B$ if there is a state satisfying A that can be added to it so that the resulting state satisfies B.

program	**LoC**	`-O`	**LoC.ll**	**time (s)**	**z3.t (s)**	**z3.%**	**theory**
`field_access.ll`	124	-	124	2.00	0.57	28.5%	-
`pointer_arith.c`	29	`-O0`	113	6.43	1.50	23.3%	-
`list_traverse.cpp`	10	`-O0`	24	3.15	1.46	46.3%	lists
`list_dispose.c`	20	`-O0`	20	2.59	1.08	41.7%	lists
`list_dispose.c` (Fig. 1)	20	`-O2`	17	2.18	0.88	40.4%	lists
`list_dispose.f`	20	`-O1`	20	2.59	1.08	41.7%	lists
`list_ip_reverse.c`	16	`-O0`	40	19.84	4.32	21.8%	lists
`list_ip_reverse.c`	16	`-O2`	19	3.64	2.09	57.4%	lists
`mark_dag.c` (Fig. 5)	11	`-O0`	35	0.82	0.26	32.1%	dag
`mark_dag_obfuscated.c`	17	`-O0`	65	7.20	0.81	11.2%	dag
`mark_dag_obfuscated.c`	17	`-O2`	31	0.93	0.21	22.6%	dag

Table 1. Experimental results. The first column lists test cases; the second their numbers of lines; the third is the optimisation level of the compiler; the fourth is the number of bitcode lines. The next three columns respectively list the total time of the analysis, and the time spent in `z3` both in seconds and as a percentage of the total time. The last column lists theories passed to `llStar`. The experiments were run on a 3.3GHz dual-core i3-3220 desktop machine.

and the loop invariants are discovered automatically. The last example is a modification (preserved by `-O2`) of `mark` (Fig. 5) that swaps pointers to the left and right children before and after each recursive call. These recursive functions are easier to verify than iterative functions, since there is no loop invariant to discover. All these test cases (and more) can be found in the `llStar` distribution [26].

5 Conclusion

We have described the `llStar` tool for automatic, compositional verification of LLVM bitcode. `llStar` balances several requirements to reason concisely about intricate, low-level heap manipulation. Verification of basic LLVM programs is reasonably efficient, while `llStar`'s flexible proof engine allows for easy prototyping of new theories for heap structures.

A few lines of work are envisioned for the future. Some have to do with increasing the level of automation in `llStar`. One could for instance add bi-abduction [7] support to `llStar` in order to discover function preconditions automatically, or implement cyclic abduction techniques to discover the inductive predicates used in a program, together with their logic rules [4]. Finally, connecting `llStar` to an abstract interpreter [9] would allow for discovering richer loop invariants with respect to values. In fact, this would solve one of the main current limitations of `llStar` when reasoning about array programs. Some of these features are present in `coreStar` to some degree, but would require better support and integration to be used in practice by `llStar`. It would also be invaluable to have `llStar` generate machine-checkable proofs of bitcode programs. This could for instance be achieved by drawing a connection between `llStar` proof rules and the certified semantics of bitcode defined by VellVM [27].

References

1. J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.
2. M. Botincan, M. J. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *Electr. Notes Theor. Comput. Sci.*, 2009.
3. M. Botinčan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson. coreStar: The core of jStar. In *BOOGIE*, 2011.
4. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. Technical Report RN/13/14, University College London, 2013.
5. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
6. C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods*, 2011.
7. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 2011.
8. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, 2009.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
10. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, 2009.
11. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
12. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
13. D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
14. K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *SAS*, 2013.
15. S. Falke, D. Kapur, and C. Sinz. Termination analysis of c programs using compiler intermediate languages. In *RTA*, 2011.
16. The Frama-C platform for static analysis of C programs, 2008.
17. G. He, S. Qin, C. Luo, and W.-N. Chin. Memory usage verification using Hip/Sleek. In *ATVA*, 2009.
18. A. Hobor and J. Villard. The ramifications of sharing in data structures. In *POPL*, 2013.
19. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
20. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
21. J. A. Navarro Pérez and A. Rybalchenko. Separation logic modulo theories. In *APLAS*, 2013.
22. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, 2005.
23. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV*, 2013.
24. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
25. C. Sinz, F. Merz, and S. Falke. LLBMC: A bounded model checker for LLVM's intermediate representation - (competition contribution). In *TACAS*, 2012.
26. J. Villard. `llStar` website. `http://bitbucket.org/jvillard/llstar/`.
27. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.