Exploiting pointer analysis in memory models for deductive verification

Quentin Bouillaguet, Francois Bobot, Mihaela Sighireanu, Boris Yakobowski

Exploiting Pointer Analysis in Memory Models for Deductive Verification*

Quentin Bouillaguet[1,2]✉, François Bobot[1], Mihaela Sighireanu[2], and Boris Yakobowski[1,3]

[1] CEA, LIST, Software Reliability Laboratory, France
firstname.lastname@cea.fr
[2] IRIF, University Paris Diderot and CNRS, France
[3] AdaCore, Paris, France

Abstract. Cooperation between verification methods is crucial to tackle the challenging problem of software verification. The paper focuses on the verification of C programs using pointers and it formalizes a cooperation between static analyzers doing pointer analysis and a deductive verification tool based on first order logic. We propose a framework based on memory models that captures the partitioning of memory inferred by pointer analyses, and complies with the memory models used to generate verification conditions. The framework guided us to propose a pointer analysis that accommodates to various low-level operations on pointers while providing precise information about memory partitioning to the deductive verification. We implemented this cooperation inside the Frama-C platform and we show its effectiveness in reducing the task of deductive verification on a complex case study.

1 Introduction

Software verification is a challenging problem for which different solutions have been proposed. Two of these solutions are deductive verification (DV) and static analysis (SA). Deductive verification is interested in checking precise and expressive properties of the input code. It requires efforts from the user that has to specify the properties to be checked, plus other annotations – e.g., loop invariants. Using these specifications, DV tools build verification conditions which are formulas in various logic theories and send them to specialized solvers. For C programs with pointers, DV has been boosted by the usage of Separation Logic [29], which leads to compact proofs due to the local reasoning allowed by the separating conjunction operator. However, for programs with low-level operations on pointers (e.g., pointer arithmetics and casting), this approach is actually limited by the theoretical results on the fragment of separation logic employed [7] and on the availability of solvers. Therefore, this class of programs is most commonly dealt using classic approaches based on memory models *à la* Burstall-Bornat [9,6], which may be adapted to be sound in presence of low-level operations [31] and dynamic allocation [36]. The memory model is chosen

* This work was partially supported by grant ANR-14-CE28-0018-03.

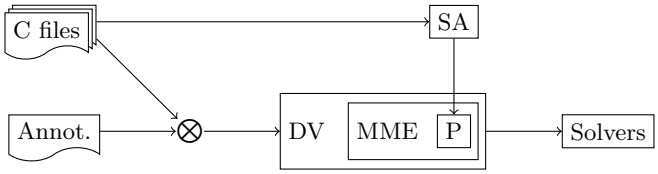

Fig. 1: Verification using memory partitioning inferred by pointer analysis

in general by the DV engine which may employ some heuristics to guide the choice [20]. Indeed, changing the memory model may result in an increase of the number of proofs discharged automatically [35]. However, annotations on non aliasing between pointers and memory partitioning complicates the task of users and of underlying solvers.

On the other hand, static analysis targets checking a fixed class of properties. This loss in the expressivity of properties is counterbalanced by a high degree of automation. For example, static pointer analysis for C programs usually computes over-approximations of the set of values (addresses) for each pointer expression at each control point. These abstractions do not speak about concrete memory addresses, but refer to symbolic memory regions provided by the memory allocated to program variables and in heap by dynamic allocation methods.

The information obtained by static analysis may help to infer partitioning of the memory in disjoint regions which can then be used by DV tools. The success of this collaboration between SA and DV tool strongly depends on the coarseness of the abstraction used by SA to keep track of the locations scanned by a pointer inside each memory region. For example, consider `p` a pointer to integer and a variable `s` of type record with five integer fields, `struct {int m,n,o,p,q;}`, such that `p` scans locations of all fields of `s` except `o` (i.e., `&s.m`, `&s.n`, `&s.p` and `&s.q`). Pointer analyses (e.g.,Section 5.2 of [28]) over-approximate the location of `p` to any location in the memory region of `s` which is multiple of an integer, thus including the spurious `o` field. Therefore, it is important to be able to try several SA algorithms to gather precise information about the memory partitioning.

Our contribution targets this specific cooperation of SA and DV methods in the context of first-order logic solvers. The verification process we propose is summarized by the flow diagram in Figure 1. The code to be verified is first given to the static analyzer to produce state invariants including a sound partitioning P of the program's memory. The partitioning P is exploited by a functor M which produces a memory model environment MME used by the DV tool to generate verification conditions into a logic theory supported by automatic solvers. Our first contribution is the formalization of the functor M and of the information it needs from the static analysis. Secondly, we demonstrate that several existing pointer analyses may be used in this general framework. Thirdly, we implemented this functor in the Frama-C platform [22] between the plug-ins Eva for static analysis and WP for deductive verification. Finally, we propose a new pointer analysis exploiting a value analysis based on abstract interpretation; this analysis is able to produce the memory model that reduces the verification effort of a relevant benchmark.

```
1  typedef int32_t data_t;
2  typedef uint8_t pos_t;
3  typedef struct {
4    data_t *in1, *in2, *in3, *in4;
5    data_t *out1,*out2,*out3,*out4;
6    pos_t  *pos1,*pos2,*pos3,*pos4;
7                } intf4_t;
8  /*@ requires:
9   *    sep({args->in1,...,args->in4},
10  *        args->out1,...,args->out4,
11  *        args->pos1,...,args->pos4);
12  *   ensures:
13  *    sorted_vals(&(args->out1),4);
14  *   ensures:
15  *    perm(&(args->in1),&(args->out1),
16  *         &(args->pos1),4); */
17 void sort4(intf4_t *args) {
18   data_t **inArr   =
19     (data_t **) &(args->in1);
20   data_t **outArr  =
21     (data_t **) &(args->out1);
22   pos_t **posArr =
23     (pos_t **) &(args->pos1);
24   /** init arrays from inputs */
25   int32_t sortArr[4]; // values
26   uint8_t permArr[4]; // permutation
27   /*@ loop invariant: ... */
28   for (int i = 0; i < 4; i++) {
29     sortArr[i] = *inArr[i];
30     permArr[i] = i;
31   }
32
33   /* sorting algorithm on sortArr
34    * with permutation in permArr */
35
36   /** copy results to outputs */
37   /*@ loop invariant: ... */
38   for (int i = 0; i < 4; i++) {
39     (*outArr[i]) = sortArr[i];
40     (*posArr[i]) = permArr[i];
41   }
42 }
```

Fig. 2: Sorting function for $\mathbf{N} = 4$ inputs and outputs

2 A Motivating Example

We overview the issues targeted and the solution proposed in this work using the C code given in Figure 2. This code is extracted from the C code generated by the compiler of a high level data flow language. It combines at least three complex features of pointers in C.

The first feature is the duality of records and arrays, which is used here to interpret the (large) list of arguments for a function as individual fields in a compound (record) type or as cells of an array. Thus, the read of the k-th field ($k \geq 0$) named `fk` of a record stored at location `s` and using only fields of type τ may be written `s->fk` or `*(&(s->f0)+k)`, where `f0` is the first field. It is debatable whether the C standard actually permits this form of dual indexing between records with fields of the same type and arrays [34], but some programs, including this one, use this feature with success. In our example, this duality is used in function `sort4` to ease the extraction of numerical values from the inputs and the storage of the sorted values in the outputs. This first feature makes our running example more challenging, but the technique we propose is also effective when the parameters are encapsulated in arrays of pointers, e.g., when inputs and outputs are declared as a field of type array by `data_t* in[4]`. The second feature is precisely the usage of arrays of pointers which is notoriously difficult to be dealt precisely by pointer analyses. The third feature is the complex separation constraints between pointers stored in arrays, which leads to a quadratic number of constraints on the size of the array and complicates the task of DV tools. In the following, we discuss in detail these issues and our approach to deal with them.

Inputs and outputs of `sort4` have the same type, `data_t`, which shall encapsulate a numerical value to be sorted. For simplicity, we consider only one field of `int32_t` type for `data_t`. Type `pos_t` models an element of the permutation

and denotes the destination position (an unsigned integer) of the value sorted. The parameters of `sort4` are collected by type `intf4_t`: four pointers to `data_t` for input values, four pointers to `data_t` for output values, and four pointers to `pos_t` for the new positions of input values.

The function is annotated with pre/post conditions and with loop invariants. The pre-condition requires (predicate `sep`) that (1) all pointers in `*args` are valid, i.e. point to valid memory locations, (2) the pointers in fields `in` are disjoint from any pointer in fields `out` and `pos`, and (3) pointers in fields `out` and `pos` are pairwise disjoint. Notice that the `in` fields may alias. The post-condition states that the values pointed by the fields `out` are sorted (predicate `sorted_vals`) and, for each output i, the value of this output is equal to the value of the input j such that `pos`$[j]$ is i (predicate `perm`).

The separation pre-condition is necessary for the proof of the post-condition because any aliasing between fields `out` may crush the results of the sorting algorithm. The encoding of this pre-condition in FOL is done by a conjunction of dis-equalities which is quadratic on the number of pointers concerned. More precisely, for n inputs (and so n outputs and n positions), there are $O(n^2)$ such constraints. (In SL, this requirement is encoded in linear formulas.) The original code from which our example is inspired instantiate n with 24 and therefore generates a huge number of dis-equalities. Several techniques have been proposed to reduce the number of dis-equalities generated by the separation constraints. For example, a classic technique is assigning a distinct logic value (a color) to each pointer in the separated set. This technique does not apply in our example if the type `data_t` is a record with more than one field because the color shall concern only the numerical value to be sorted.

As an alternative, we propose to use precise points-to analyses to lift out such constraints and to simplify the memory model used for the proof of the function. Importantly, we perform a per-call proof of `sort4`, instead of a unitary proof. For each call of `sort4`, the static analysis tries to check that the separation pre-condition is satisfied and provides a model for the memory where the pointers are dispatched over disjoint zones. Unfortunately, the precision of the points-to analyses (and consequently the number of separation constraints discharged) may change radically with the kind of initialization done for the arguments of `sort4`. We will illustrate this behavior for two calls of `sort4` given in Figure 3: the call in listing (a) uses variables and the one in listing (b) uses arrays. Notice that each call satisfies the separation pre-condition of `sort4`.

Typed memory model: For completeness, we quickly present first how DV tools using FOL deal with our example using the Burstall-Bornat model. In this model, the memory is represented by a set of array variables, each array corresponding to a (pre-defined, basic) type of memory locations. For our example, the memory model includes six array variables: `M_int32`, `M_uint8`, `M_int32_ref`, `M_uint8_ref`, `M_int32_ref_ref`, `M_uint8_ref_ref` storing values of type respectively `int32_t`, `uint8_t`, `int32_t*`, `uint8_t*`, `int32_t**` and `uint8_t**`. Program variables are used as indices in these arrays, e.g., variable `inArr` is an index in array `M_int32_ref_ref` and `sortArr` is index of `M_int32`.

Listing 1.1: (a) using variables

```
1  data_t  df_1,df_2,...,df_8;
2  pos_t   pf_1,pf_2,pf_3,pf_4;
3  intf4_t SORT = {
4    .in1=&df1,  .in2=&df2,
5    .in3=&df3,  .in4=&df4,
6    .out1=&df5, .out2=&df6,
7    .out3=&df7, .out4=&df8,
8    .pos1=&pf1, .pos2=&pf2,
9    .pos3=&pf3, .pos4=&pf4 };
10
11 df_1 = nondet_data();
12 df_2 = nondet_data();
13 df_3 = nondet_data();
14 df_4 = nondet_data();
15
16 sort4(&SORT);
```

Listing 1.2: (b) using arrays

```
1  data_t  df[8];
2  pos_t   pf[4];
3  intf4_t SORT = {
4    .in1=df+1,  .in2=df+2,
5    .in3=df+3,  .in4=df+4,
6    .out1=df+5, .out2=df+6,
7    .out3=df+7, .out4=df,
8    .pos1=pf,   .pos2=pf+1,
9    .pos3=pf+2, .pos4=pf+3 };
10
11 df[1] = nondet_data();
12 df[2] = nondet_data();
13 df[3] = nondet_data();
14 df[4] = nondet_data();
15
16 sort4(&SORT);
```

Fig. 3: Two calls for the sorting function using different initialization

The separation pre-condition of `sort4` is encoded by dis-equalities, e.g., `M_int32_ref[args_in4] <> M_int32_ref[args_out1]` where `args_in4` is bound to the term *shift*(`M_int32_ref_ref[args], in4`) which encodes the access to the memory location `&(args->in4)` using the logic function *shift*; `args_out1` is defined similarly. However, these dis-equalities are not propagated through the assignments at lines 18–23 in Figure 2, which interpret the sequence of (input/output/position) fields as arrays. Therefore, additional annotations are required to prove the correct initialization of the output at lines 39–41. Some of these annotations may be avoided using our method that employs pointer analyses to infer precise memory models, as we show below.

Base-offset pointer analysis: Consider now a pointer analysis which is field and context sensitive, and which computes an over-approximation of the value of each pointer expression at each program statement. The over-approximation, that we name *abstract location*, is built upon the standard concrete memory model of C [25]. An abstract location is *a partial map* between the set of program's variables and the set of intervals in $\mathbb{N}$. An element of this abstraction, $(\mathtt{v}, i^{\#})$, denotes the symbolic (i.e., not related with the locations in the virtual memory space used during the concrete execution) memory block that starts at the location of the program variable `v` (called also *base*), and the abstraction by an interval $i^{\#}$ of the set of possible offsets (in bytes) inside the symbolic block of `v` to which the pointer expression may be evaluated. In this memory model, symbolic blocks of different program variables are implicitly separated: it is impossible to move from the block of one variable to another using pointer arithmetic. The memory model is modeled by a set of logic arrays, one for each symbolic block. The over-approximation computed by the analysis allows to dispatch a pointer expression used in a statement on these arrays.

In our example, for the call of `sort4` in Figure 3 (a), the memory model includes the symbolic blocks for program's variable `df`i, `pf`i and `SORT`. The above analysis computes for the pointer expressions `args->in1` and `*(args->in1)` at the start of `sort4`, the abstract location $\{(\mathtt{SORT}, [0, 0])\}$ and $(\mathtt{df1}, [0, 0])$ re-

spectively. The abstract locations for the pointer expressions involving other fields of `args` are computed similarly. The separation pre-condition of `sort4` is implied by these abstract locations. After the fields of `args` are interpreted as arrays (lines 18–23 of `sort4`), the pointer expression `outArr+i` at line 39, where `i` is restricted to the interval $[0, 3]$, is over-approximated to the abstract location $\{(\texttt{SORT}, [16, 31])\}$. Similarly, `inArr+i` is abstracted by $\{(\texttt{SORT}, [0, 15])\}$. Therefore, the left value given by the pointer expression `outArr[i]` (at line 39) is (precisely) computed to be $\{(\texttt{df5}, [0, 0]), ..., (\texttt{df8}, [0, 0])\}$. This allows proving the correctness of the output computed by `sort4`.

For the call in Figure 3 (b), the memory model includes symbolic blocks for program's variable `df`, `pf` and `SORT`. The analysis computes for pointer expressions `args->in1` and `*(args->in1)` (used at the start of `sort4`), the abstract location $\{(\texttt{SORT}, [0, 0])\}$ resp. $(\texttt{df}, [0, 3])$, which also allows to prove the separation pre-condition. The interpretation of fields as arrays (lines 18–23) leads to the abstract location $\{(\texttt{df}, [1, 4])\}$ for `inArr+i`, which is very precise. However, because the initialization of the field `SORT.out4` at line 18 in Figure 3 (b) breaks the uniformity of the interval, the pointer expression `outArr+i` (at line 39) is over-approximated to $\{(\texttt{df}, [0, 7])\}$. This prevents the proof of the post-condition.

In conclusion, such an analysis is able to infer a sound memory model that offers a finer grain of separation than the typed memory model. However, it is not precise enough to deal with the array of pointers and field duality in records.

Partitioning analysis: Based on the base-offset pointer analysis above, we define in Section 5.3 a new analysis that computes for each pointer expression an abstract location that collects a finite set of *slices of symbolic blocks*, i.e., the abstraction is a partial mapping from program's variables to sets of intervals representing offsets in the block. With this analysis, the abstract location computed for `outArr+i` (at line 39 of `sort4`, call in Figure 3 (b)) is more precise, i.e., $\{\texttt{df} \mapsto \{[5, 7], [0, 0]\}\}$, and it allows to prove the post-condition for `sort4`. Notice that the analysis computes *a finite set of slices* in symbolic blocks whose concretizations (sets of locations) are pairwise disjoint. For this reason, this analysis may be imprecise if its parameter fixing the maximum size of this set is exceeded. This analysis also deals precisely with the call of `sort4` in Figure 3 (a).

Dealing with different analyses: The above comments demonstrate the diversity of results obtained for the memory models for different points-to analysis algorithms. One of our contributions is to define a generic interface for the definition of the memory model for the DV based on the results obtained by static analyses doing points-to analysis (SPA). This interface eases the integration of a new SPA algorithm and the comparison of results obtained with different SPA algorithms. We formalize this interface in Section 4 and instantiate it for different SPA algorithms in Section 5. Our results are presented in Section 6.

3 Generating Verification Conditions

To fix ideas, we recall the basic principles of generating verification conditions (VC) using a memory model by means of a simple C-like language.

$\mathtt{n} \in \mathbb{N}, \mathtt{k} \in \mathbb{Z}$	integer constants	`num`	integer type in $\{\mathtt{i8}, \mathtt{u8}, \mathtt{i16}, \ldots, \mathtt{u64}\}$
$\mathtt{rt} \in \mathtt{Rtyp}$	record type names	$\mathtt{f} \in \mathtt{Fld}$	field names
$\mathtt{v} \in \mathtt{Cvar}$	program variables	$\mathtt{op} \in \mathtt{O}$	unary and binary arithmetic operators

scalar types	$\mathtt{Styp} \ni \mathtt{u} ::= \mathtt{num} \mid \mathtt{t\ ptr}$
program types	$\mathtt{Ctyp} \ni \mathtt{t} ::= \mathtt{u} \mid \mathtt{rt} \mid \mathtt{u}[\mathtt{n}]$
expressions	$\mathtt{Expr} \ni \mathtt{e} ::= \mathtt{ie} \mid \mathtt{a}$
integer expressions	$\mathtt{Iexpr} \ni \mathtt{ie} ::= \mathtt{k} \mid \mathtt{lv} \mid \mathtt{op\ ie} \mid \mathtt{ie\ op\ ie}'$
address expressions	$\mathtt{Aexpr} \ni \mathtt{a} ::= \mathtt{null} \mid \mathtt{lv} \mid \&\mathtt{lv} \mid \mathtt{a} + \mathtt{ie}$
left-values	$\mathtt{Lval} \ni \mathtt{lv} ::= \mathtt{v} \mid \mathtt{lv.f} \mid *\mathtt{a}$
statements	$\mathtt{Stmt} \ni \mathtt{s} ::= \mathtt{lv{=}e} \mid \mathtt{assert\ e}$

Fig. 4: Syntax of our Clight fragment

3.1 A Clight Fragment

We consider a fragment of Clight [4] that excludes casts, union types and multi-dimensional arrays. We also restrict the numerical expressions to integer expressions. The syntax of expressions, types and atomic statements is defined by the grammar in Figure 4. This fragment is able to encode all assignment statements in Figures 2–3 using classic syntax sugar (e.g., `**(arr + i)` for `*arr[i]`, `&((*args).in1)` for `&(args->in1)`). Complex control statements can be encoded using the standard way. User defined types are pointer types, static size array types, and record types. A record type declares a list of typed fields with names from a set `Fld`; for simplicity, we suppose that each field has a unique name. We split expressions into integer expressions and address expressions to ease their typing. Expressions are statically typed by a type `t` in `Ctyp`. When this information is needed, we write $\mathtt{e}^{\mathtt{t}}$.

We choose to present our work on this simple fragment for readability. However, our framework may be extended to other constructs. For example, our running example contains struct initialization. Struct assignment may be added by explicit assignment of fields. Type casting for arithmetic and compatible pointer types (i.e., aligned on the same type) may be dealt soundly in DV tools employing array-based memory models using the technique in [31]. Functions calls may be also introduced if we choose context-sensitive SA. In general, DV tools conduct unit proofs for functions. We restrict this work to whole-program proofs, because it avoids the requirement that SA is able to conduct analyses starting with function's pre-conditions. Our memory model could however be instantiated with an inter-procedural SA, thus enabling unit proof of functions.

3.2 Memory Model

We define the denotational semantics of our language using an environment called *abstract memory model* (AMM). (This name is reminiscent of the first abstract memory model defined in [24,25] for CompCert. We enriched it with some notations to increase readability of our presentation.) Figure 5 summarizes

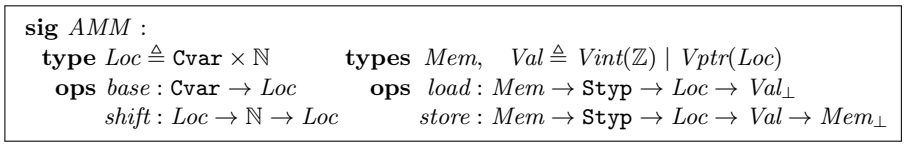

Fig. 5: Abstract signature for the concrete memory model

the elements of this abstract memory model. The link between abstract and concrete standard memory models is provided in the extended version.

The states of the memory are represented by an abstract data type *Mem* which associates locations of type *Loc* to values in the type *Val*. Locations are pairs (b, o) where b is the identifier of a symbolic block and o is an integer giving the offset of the location in the symbolic block of b. Because we are not considering dynamic allocation, symbolic blocks are all labeled by program's variables. Thus we simplify the concrete memory model by replacing block identifiers by program variables. Values of type *Loc* are built by two operations of *AMM*: *base*(v) gives the location of a program variable v and *shift*(ℓ, n) computes the location obtained by shifting the offset of location ℓ by n bytes. The shift operation abstracts pointer arithmetics. The typing function cty(.) is extended to elements of *Loc* based on the typing of expressions used to access them. Some operations are partial and we denote by $\perp$ the undefined value. A set A extended with the undefined value is denoted by $A_\perp$. The axiomatization of loading and storing operations is similar to the one in [24,25].

3.3 Semantics

Figure 6 defines the rules of the semantics using the abstract memory model, via the overloaded functions $[\![\cdot]\!]$. The semantic functions are partial: the undefined case $\perp$ cuts the evaluation. The operators $\widehat{\texttt{op}}$ are interpretations of operations op over integer types num. The functions offset$(\cdot)$ and sizeof$(\cdot)$ are defined by the Application Binary Interface (ABI) and depend on the architecture. Conversions between integer values are done using function cast$(\cdot,\cdot)$.

3.4 Generating Verification Conditions

Verification conditions (VC) are generated from Hoare's triple $\{P\}$ s $\{Q\}$ with P and Q formulas in some logic theory $\mathcal{T}$ used for program annotations and s a program statement. The classic method [23,18] is built on the computation of a formula $R_{\texttt{s}}(\boldsymbol{v}_b, \boldsymbol{v}_e)$ in $\mathcal{T}$ specifying the relation between the states of the program before and after the execution of s, which are represented by the set of logic variables $\boldsymbol{v}_b$ resp. $\boldsymbol{v}_e$. The VC built for the above Hoare's triple is $\forall \boldsymbol{v}_b, \boldsymbol{v}_e.\ \big(P(\boldsymbol{v}_b) \wedge R(\boldsymbol{v}_b, \boldsymbol{v}_e)\big) \implies Q(\boldsymbol{v}_e)$ and it is given to solvers for $\mathcal{T}$ to check its validity. In the following, we denote by $\mathcal{E}$ the set of logic terms built in the logic theory $\mathcal{T}$ using the constants, operations, and variables in a set $\mathcal{X}$. For a logic sort τ, we designate by $\mathcal{E}_\tau$ the terms of type τ.

Compilation environment: Formula $R_{\texttt{s}}(\cdot,\cdot)$ is defined based on the dynamic semantics of statements, like the one given in Figure 6 for our language. The

$[\![\cdot]\!] : \texttt{Stmt} \to \mathit{Mem} \to \mathit{Mem}_\bot$

$[\![\texttt{lv}^{\texttt{u}}\texttt{=e}]\!](m) \triangleq \mathit{store}(m, \texttt{u}, [\![\texttt{lv}]\!](m), [\![\texttt{e}]\!](m))$

$[\![\texttt{assert e}]\!](m) \triangleq \text{if } [\![\texttt{e}]\!](m) \not\sim 0 \text{ then } m \text{ else } \bot$

$[\![\cdot]\!] : \texttt{Expr} \to \mathit{Mem} \to \mathit{Val}_\bot$

$[\![\texttt{ie}]\!](m) \triangleq \mathit{Vint}([\![\texttt{ie}]\!](m))$

$[\![\texttt{a}]\!](m) \triangleq \mathit{Vptr}([\![\texttt{a}]\!](m))$

$[\![\cdot]\!] : \texttt{Iexpr} \to \mathit{Mem} \to \mathbb{Z}_\bot$

$[\![\texttt{i}]\!](m) \triangleq \texttt{i}$

$[\![\texttt{lv}^{\texttt{num}}]\!](m) \triangleq i,\ \mathit{Vint}(i) = \mathit{load}(m, \texttt{num}, [\![\texttt{lv}]\!](m))$

$[\![\texttt{op ie}]\!](m) \triangleq \widehat{\texttt{op}}([\![\texttt{ie}]\!](m))$

$[\![\cdot]\!] : \texttt{Lval} \to \mathit{Mem} \to \mathit{Loc}_\bot$

$[\![\texttt{v}]\!](m) \triangleq \mathit{base}(\texttt{v})$

$[\![\texttt{lv.f}]\!](m) \triangleq \mathit{shift}([\![\texttt{lv}]\!](m), \texttt{offset(f)})$

$[\![*\texttt{a}]\!](m) \triangleq [\![\texttt{a}]\!](m)$

$[\![\cdot]\!] : \texttt{Aexpr} \to \mathit{Mem} \to \mathit{Loc}_\bot$

$[\![\texttt{null}]\!](m) \triangleq \mathit{base}(\texttt{null})$

$[\![\texttt{lv}^{\texttt{u}[n]}]\!](m) \triangleq [\![\texttt{lv}]\!](m)$

$[\![\texttt{lv}^{\texttt{t ptr}}]\!](m) \triangleq \ell \text{ where } \mathit{load}(m, \texttt{t ptr}, [\![\texttt{lv}]\!](m)) = \mathit{Vptr}(\ell)$

$[\![\&\texttt{lv}]\!](m) \triangleq [\![\texttt{lv}]\!](m)$

$[\![\texttt{a}^{\texttt{t ptr}} + \texttt{ie}]\!](m) \triangleq \mathit{shift}([\![\texttt{a}]\!](m), \texttt{sizeof(t)} \times \texttt{cast}([\![\texttt{ie}]\!](m), \texttt{u32}))$

Fig. 6: Semantics of our Clight fragment

compilation of this semantics into formulas $\mathcal{T}$ uses a *memory model environment* (called simply environment) that implements the interface of the abstract memory model given in Figure 5. This environment changes at each context call and keeps the information required by the practical compilation into formulas, e.g., the set of variables used for modeling the state at the current control point of this specific context call. Figure 7 defines the signature of memory environments.

The types Mem and Loc encapsulate information about the program states and memory locations respectively. Notice that the logical representation of locations is hidden by this interface, which allows to capture very different memory models. The compilation information about the values stored is given by the type Val, which represent integers by integer terms in $\mathcal{T}$, i.e., in the set $\mathcal{E}_\mathbb{I}$. Operation shift implements arithmetics on locations by an integer term. Operation store encapsulates the updating of the environment by an assignment and produces a new environment and a term in $\mathcal{E}_\mathbb{B}$, i.e., a formula of $\mathcal{T}$.

Prerequisites on the logic theory: For DV tools based on first order logic, the theory $\mathcal{T}$ is a multi-sorted FOL that embeds the logic theory used to annotate programs (which usually includes boolean and integer arithmetics theories) and the McCarthy's array theory [26] employed by the Burstall-Bornat memory model [6] to represent atomic memory blocks. The memory model environment associates to each memory blocks a set of logic array variables using base operations. It encodes the operations $\mathsf{load}(\mathsf{m}, \mathtt{t}, \ell)$ resp. $\mathsf{store}(\mathsf{m}, \mathtt{t}, \ell, v)$ into logic array operations $\mathit{read}(a, o)$ resp. $\mathit{store}(a, o, v)$, where a is the array variable for the symbolic block b of location ℓ that stores values of type $\mathtt{t}$ and o is the offset of ℓ in b. $\mathcal{T}$ also embeds abstract data types (or at least polymorphic pairs with

sig MME :

type Loc

ops base : Cvar → Loc

shift : Mem → Loc → $\mathcal{E}_\mathbb{I}$ → $\mathsf{Loc}_\bot$

types Mem, Val ≜ Vint($\mathcal{E}_\mathbb{I}$) | Vptr(Loc)

ops load : Mem → Styp → Loc → $\mathsf{Val}_\bot$

store : Mem → Styp → Loc → Val → $(\mathsf{Mem} \times \mathcal{E}_\mathbb{B})_\bot$

Fig. 7: Signature of the memory model environments

component selection by *fst* and *snd*), and uninterpreted functions. Polymorphic conditional expression "$(e_{cond})?e_{true} : e_{false}$" are also needed.

In the following, we use the logic theory above $\mathcal{T}$ and suppose that an infinite number of fresh variables can be generated. To ease the reading of environment definitions, we distinguish the logic terms by using the mathematical style and by underlining the terms of $\mathcal{T}$, e.g., $\underline{x + x}$. For example, the logic term $\underline{read(\mathsf{m(b)}, 4 + x)}$ is built from a VC generator term $\mathsf{m(b)}$ that computes a logic term of array type and the logic sub-term $\underline{read(\cdot, 4 + x)}$.

Example: Consider the Hoare's triple $\{P\}$ $(*(\&\mathtt{r.f}))^{\mathtt{i8}} = \mathtt{5}$ $\{Q\}$. Let l_0 be $\mathsf{shift}(\mathsf{m}_0, \mathsf{base}(\mathtt{r}), \mathtt{offset(f)})$, where m_0 (resp. m_1) is the environment for the source state (resp. modified by the store for the destination state); that is $\mathsf{m}_1, \phi_1 \triangleq \mathsf{store}(\mathsf{m}_0, \mathtt{i8}, \mathsf{l}_0, \mathsf{Vint}(5))$. The formula $\underline{P}$ (resp. $\underline{Q}$) is generated from P (resp. Q) using compilation environment m_0 (resp. m_1). Then the VC generated by the above method is $\underline{P} \wedge \phi_1 \implies \underline{Q}$. Notice that the above calls of the environment's operations follow the order given by the semantics in Figure 6, except for the failure cases. Indeed, to simplify our presentation, we consider that statement's pre-condition includes the constraints that eliminate runs leading to undefined behaviors. Therefore, the VC generation focuses on encoding in $R_{\mathtt{s}}(\cdot, \cdot)$ the correct executions of statements.

4 Partition-based Memory Model

We define a functor that produces memory models environments implementing the interface on Figure 7 from the information inferred by a pointer analysis. The main idea is that the SA produces a finite partitioning of symbolic blocks into a set of pairwise disjoint sub-blocks and each sub-block is mapped to a specific set of array logic variables by the compilation environment. We first formalize the pre-requisites for the pointer analysis using a signature constrained by well-formed properties. Then, we define the functor by providing an implementation for each element of the interface on Figure 7.

4.1 Pointer Analysis Signature

A necessary condition on the pointer analysis is its soundness. To ease the reasoning about this property of analysis, we adopt the abstract interpretation [16] framework. In this setting, a SA computes an abstract representation $s^\#$ of the set of concrete states reached by the program's executions before the execution of each statement. The abstract states $s^\#$ belong to a complete lattice $(S^\#, \sqsubseteq^\#)$ which is related to the set of concrete program configurations *State* by a pair of functions $\alpha : 2^{State} \to S^\#$ (abstraction) and $\gamma : S^\# \to 2^{State}$ (concretization) forming a Galois connection. In the following, we overload the symbol γ to denote concretization functions for other abstract objects.

Aside being sound, the SA shall be context sensitive and provide, for each context call, an implementation of the signature on Figure 8. The values of $\mathcal{S}$ provides, for each statement of the current context, the abstract state in $S^\#$

sig PA :

type $\mathcal{L}$	**type** $\mathcal{S}$	**type** $\mathcal{B}$
ops base : $\texttt{Cvar} \to \mathcal{L}$	**ops** load : $\mathcal{S} \to \mathsf{Ptr} \to \mathcal{L} \to \mathcal{L}$	**ops** base : $\mathcal{B} \to \texttt{Cvar}$
domain : $\mathcal{L} \to 2^{\mathcal{B}}$	shift : $\mathcal{S} \to \mathcal{L} \to \mathcal{E}_{\mathbb{I}} \to \mathcal{L}$	slice : $\mathcal{B} \to \mathcal{E}_{\mathbb{I}} \to \mathcal{E}_{\mathbb{B}}$

$$\text{disjointness: } \forall b_1^\#, b_2^\# \in \mathcal{B} \cdot b_1^\# \neq b_2^\# \Rightarrow \gamma(b_1^\#) \cap \gamma(b_2^\#) = \emptyset \quad (1)$$

$$\text{completeness: } \forall \mathtt{v} \in \texttt{Cvar}\ \forall i \in [0, \texttt{sizeof}(\texttt{cty}(\mathtt{v})) - 1)\ \exists b^\# \in \mathcal{B} \cdot (\mathtt{v}, i) \in \gamma(b^\#) \quad (2)$$

$$\text{unique base: } \forall b^\# \in \mathcal{B}\ \exists! \mathtt{v} \in \texttt{Cvar} \cdot \gamma(b^\#) \subset \{(\mathtt{v}, i) \mid i \in \mathbb{N}\} \quad (3)$$

$$\text{sound } \mathcal{B} \text{ ops: } \forall b^\# \in \mathcal{B} \cdot \gamma(b^\#) = \{(\mathtt{v}, i) \in \mathit{Loc} \mid \mathtt{v} = \mathsf{base}(b^\#) \wedge \mathsf{slice}(b^\#, \underline{i}) = \mathit{true}\} \quad (4)$$

$$\text{sound } \mathcal{L} \text{ ops: } \forall \ell^\# \in \mathcal{L}\ \forall \ell \in \gamma(\ell^\#)\ \exists b^\# \in \mathsf{domain}(\ell^\#) \cdot \ell \in \gamma(b^\#) \quad (5)$$

$$\text{sound } \mathcal{S} \text{ ops: } \forall \mathtt{s}\ \forall s^\# \in \mathcal{S}(\mathtt{s})\ \forall \ell^\# \in s^\# \cdot \gamma(\mathsf{shift}(s^\#, \ell^\#, e)) \supseteq \{\mathit{shift}(\ell, i) \mid \ell \in \gamma(\ell^\#), m \in \gamma(s^\#), i \in [\![e]\!](m)\} \quad (6)$$

$$\forall \mathtt{s}\ \forall s^\# \in \mathcal{S}(\mathtt{s})\ \forall \ell^\# \in s^\# \cdot \gamma(\mathsf{load}(s^\#, \mathtt{t\ ptr}, \ell^\#)) \supseteq \{\mathit{load}(m, \mathtt{t\ ptr}, \ell) \mid \ell \in \gamma(\ell^\#), m \in \gamma(s^\#)\} \quad (7)$$

Fig. 8: A signature for pointer analysis and its properties

computed by the analysis. The type $\mathcal{L}$ represents the domain of abstract values computed for the pointer expressions in abstract states. The concretization function $\gamma : \mathcal{L} \to 2^{Loc}$ maps abstract locations to sets of concrete locations.

The type $\mathcal{B}$ stands for the set of pairwise disjoint abstract blocks partitioning the symbolic memory blocs, for the fixed specific context call. The concretization function for abstract blocks $\gamma : \mathcal{B} \to 2^{Loc}$ maps blocks to set of concrete locations. Equations (1) and (2) in Figure 8 specify that abstract blocks in $\mathcal{B}$ shall form a partition of the set of concrete locations available in symbolic blocks such that an abstract block belongs to a unique symbolic block.

The operation $\mathsf{base}(b^\#)$ returns the symbolic block to which $b^\#$ belongs, represented by the program variable labeling this symbolic block. The range of an abstract block $b^\#$ inside its symbolic block is specified by the operation $\mathsf{slice}(b^\#, e)$, which returns a formula (boolean term in $\mathcal{E}_{\mathbb{B}}$) that constrains e to be in this range. The soundness of the base and slice operations is specified by equation (4). The set of abstract blocks covered by an abstract location is provided by the operation domain, whose soundness is specified by equation (5). The operation base abstracts the offset 0 of a program variable. Abstract locations may be shifted by an integer term using operation shift. Operation $\mathsf{load}(s, \mathtt{t\ ptr}, \ell^\#)$ computes the abstract location stored at $\ell^\#$ in some context s, i.e., it dereferences $\ell^\#$ of type $\mathtt{t\ ptr\ ptr}$ for some $\mathtt{t}$. (We denote by Ptr the set of all pointer types in the program.) The last two operations shall be sound abstract transformers on abstract locations, as stated in equations (6) resp. (7).

4.2 A Functor for Memory Model Environments

We define now our functor that uses the signature PA to define the elements of the memory model environment MME defined in Figure 7. To disambiguate symbols, we prefix names of types and operations by the name of the signature or logic theory when necessary.

Environment's type: A compilation environment $\mathsf{m} \in \mathsf{Mem}$ stores the mapping to abstract states from PA and and a total function that associates to each abstract block in $\mathsf{PA}.\mathcal{B}$ a logic variable in $\mathcal{X}$:

$$\mathsf{MME.Mem} \triangleq \mathsf{PA}.\mathcal{S} \times [\mathsf{PA}.\mathcal{B} \to \mathcal{T}.\mathcal{X}] \tag{8}$$

where $[A \to B]$ denotes the set of total functions from A to B, i.e., B^A. We designate by m_s and m_ϵ the first and second component of some $\mathsf{m} \in \mathsf{Mem}$.

If an abstract block $b^\#$ stores only one type of values, the logic variable $\mathsf{m}_\epsilon(b^\#)$ has type $\mathit{array}(\mathbb{Z}, \tau)$ where τ is the logic type for the values stored. For blocks storing integer values (i.e., `num`), τ is naturally (logical) $\mathbb{Z}$ or $\mathbb{N}$. For blocks storing pointer values, τ is $\mathbb{Z} \times \mathbb{Z}$, (b, o) where the b denotes the abstract block of the location and o represents the location's offset. We denote by $\underline{b^\#}$ the integer constant that uniquely identifies $b^\# \in \mathcal{B}$. If an abstract block $b^\#$ stores values of both kinds of scalar types (notice that only scalar values are stored in array-based models), the logic variable $\mathsf{m}_\epsilon(b^\#)$ has the type pair of arrays, $(\mathit{array}(\mathbb{Z}, \mathbb{Z}), \mathit{array}(\mathbb{Z}, \mathbb{Z} \times \mathbb{Z}))$ where the first array is used for integer values and the second one for pointer values. For readability, we detail here only the case of homogeneously typed blocks. Notice that the mapping m_ϵ binds fresh array variable names to abstract blocks changed by store operation.

Locations' type: The type $\mathsf{MME.Loc}$ collects the logic encoding of locations as a pair of integer terms $(e_b, e_o) \in \mathcal{E}_\mathbb{I} \times \mathcal{E}_\mathbb{I}$ together with the abstract location ℓ provided by the static analysis, i.e., $\mathsf{MME.Loc} \triangleq \mathcal{E}_{\mathbb{I}\times\mathbb{I}} \times \mathsf{PA}.\mathcal{L}$. Intuitively, in the logic pair (e_b, e_o), e_b is interpreted as an abstract block identifier and e_o models the offset of the location in the *symbolic block of the abstract block* e_b, i.e., an integer in the slice of e_b.

Locations' operations: The values of $\mathsf{MME.Loc}$ are built by two operations $\mathsf{MME.base}$ and $\mathsf{MME.shift}$ defined as follows. For a program variable $\mathtt{v}$, $\mathsf{MME.base}(\mathtt{v})$ is based on the abstract location $\ell^\#$ returned by $\mathsf{PA.base}(\mathtt{v})$. The domain of $\ell^\#$ shall have only one abstract block $b^\#$ because program variables are located at the start of symbolic blocks. Moreover, the term denoting the offset shall be the constant 0. Formally:

$$\mathsf{MME.base}(\mathtt{v}) \triangleq \langle(\underline{b^\#}, 0), \ell^\#\rangle \text{ where } \mathsf{PA.base}(\mathtt{v}) = \ell^\#, \mathsf{domain}(\ell^\#) = \{b^\#\} \tag{9}$$

The shifting of a location in Loc by an expression e is computed based on the abstract shift operation as follows:

$$\mathsf{MME.shift}(\mathsf{m}, \langle(\underline{e_b}, \underline{e_o}), \ell^\#\rangle, e) \triangleq \langle(\underline{e'_b}, \underline{e_o + e}), \ell^\#_s\rangle \tag{10}$$

where $\ell^\#_s = \mathsf{PA.shift}(\mathsf{m}_s, \ell^\#, e)$ and the new logic base e'_b selects (using a conditional expression) the base $b^\#_i$ from the ones of $\ell^\#_s$. Let us denote by $\mathsf{fits}(e_b, \ell^\#, b^\#)$ the boolean term testing that the block identifier in e_b is one of the blocks identifiers in $\mathsf{PA.domain}(\ell^\#)$ which has the same symbolic block (i.e., base) as $b^\#_i$, i.e.:

$$\mathsf{fits}(e_b, \ell^\#, b^\#) \triangleq \bigvee_{b^\#_j \in \mathsf{PA.domain}(\ell^\#) \text{ s.t. } \mathsf{PA.base}(b^\#_j) = \mathsf{PA.base}(b^\#)} e_b = b^\#_j \tag{11}$$

Using fits, if $\mathsf{PA.domain}(\ell_s^\#)$ is $\{b_1^\#, \ldots, b_n^\#\}$, the formal definition of e'_b is:

$$e'_b \triangleq \left(\begin{array}{l} \mathsf{fits}(e_b, \ell^\#, b_1^\#) \wedge \mathsf{PA.slice}(b_1^\#, \underline{e_o + e})\ \underline{?\ b_1^\#\ :} \\ \quad \ddots\ \mathsf{fits}(e_b, \ell^\#, b_{n-1}^\#) \wedge \mathsf{PA.slice}(b_{n-1}^\#, \underline{e_o + e})\ \underline{?\ b_{n-1}^\#\ :\ b_n^\#} \end{array}\right) \quad (12)$$

Indeed, since the shift operation can not change the symbolic block, we have to test, using fits, that each resulting block identifier $b_i^\#$ has the same symbolic block as e_b.

The size of the expression encoding $\mathsf{MME.shift}$ depends on the product of sizes of domains computed by PA for $\ell^\#$ and $\ell_s^\#$. If the abstract locations have a singleton domain, i.e. $\mathsf{PA.domain}(\ell_s^\#) = \{b_1^\#\}$, then e'_b is simply $\underline{b_1^\#}$. When the precision of the SA does not enable such simplification, we could soundly avoid big expressions generated by $\mathsf{MME.shift}$ by using in $\mathsf{MME.load}$ and $\mathsf{MME.store}$ operations only the component abstract location of an environment's location.

Loading from memory: Reading an integer value in the environment m at a location $\mathsf{l} = \langle (e_b, e_o), \ell^\# \rangle$ is compiled into a read operation (denoted by $a[e]$ for concision) from an array variable obtained by statically dispatching the logical base e_b of l among the possible base identifiers in $\mathsf{PA.domain}(\ell^\#) = \{b_1^\#, \ldots, b_n^\#\}$ as follows:

$$\mathsf{MME.load}(\mathsf{m}, \mathtt{num}, \langle \underline{(e_b, e_o)}, \ell^\# \rangle) \triangleq \mathtt{Vint}\,(e) \quad (13)$$

where

$$e \triangleq \left(\begin{array}{l} \underline{e_b = b_1^\#\ ?\ \mathsf{m}_\epsilon(b_1^\#)[e_o]\ :} \\ \quad \ddots\ \underline{e_b = b_{n-1}^\#\ ?\ \mathsf{m}_\epsilon(b_{n-1}^\#)[e_o]\ :\ \mathsf{m}_\epsilon(b_n^\#)[e_o]} \end{array}\right) \quad (14)$$

The size of the expression above may be reduced by asking to SA an over-approximation $o^\#$ of the values of expression e_o in the current state. If SA is able to produce a precise result for $o^\#$, we could remove from the expression above the cases for abstract blocks $b_j^\#$ for which $\mathsf{PA.slice}(b_j^\#, o^\#) = \mathit{false}$ (i.e., the formula is invalid for the values in $o^\#$).

The expression in equation (14) is also used for reading pointer values. In this case, the expression obtained is a tuple. The abstract location corresponding to this logic expression is obtained using the abstract $\mathsf{PA.load}$ operation in the abstract state component m_s of the environment:

$$\mathsf{MME.load}(\mathsf{m}, \mathtt{t\ ptr}, \langle \underline{(b, o)}, \ell^\# \rangle) \triangleq \mathtt{Vptr}\left(e, \mathsf{PA.load}(\mathsf{m}_s, \mathtt{t\ ptr}, \ell^\#)\right) \quad (15)$$

Storing in memory: The compilation of *store* semantic operation is done by the $\mathsf{MME.store}$ operation that produces a new environment m' and a boolean term (formula) e' encoding the relation between the logic arrays associated to blocks before and after the assignment as follows:

$$\mathsf{MME.store}(\mathsf{m}, \mathsf{t}, \langle \underline{(e_b, e_o)}, \ell^\# \rangle, \mathsf{v}) \triangleq \mathsf{m}', \underline{e}' \ \text{ for } \ \mathsf{v} \in \{\mathsf{Vint}(\underline{e}), \mathsf{Vptr}(\langle \underline{e}, \ell_v^\# \rangle)\} \quad (16)$$

where $\mathsf{m}' = \langle s'^\#, \mathsf{m}'_\epsilon \rangle$ with $s'^\#$ the abstract state computed by the analysis for the control pointer after the assignment compiled. The new block mapping m'_ϵ

uses fresh logic variables for the abstract blocks in the domain $\mathsf{PA.domain}(\ell^\#) = \{b_1^\#, \ldots, b_n^\#\}$ of the abstract location $\ell^\#$ at which is done the update:

$$\mathsf{m}'_\epsilon \triangleq \mathsf{m}[b_1^\# \longleftarrow \alpha_1, \cdots, b_n^\# \longleftarrow \alpha_n] \tag{17}$$

The fresh variables are related with the old ones using the store operator on logic arrays, denoted by $a[i \longleftarrow e]$, in the generated formula e' defined as follows:

$$e' \triangleq \underline{\wedge_{i=1}^n \big((e_b = b_i^\#)\ ?\ \alpha_i = \mathsf{m}[b_i^\#][e_o \longleftarrow e]\ :\ \alpha_i = \mathsf{m}[b_i^\#]\big)} \tag{18}$$

The size of this expression may be reduced using the SA results in a similar way as for load. In general, the size of expressions generated by the compilation in equations (12), (14) and (18) depends on size of the domain for the abstract locations computed by the static analysis. Indeed, if the analysis always provides abstract locations with a singleton domain, the compilation produces expressions with only one component, while proving most separation annotations. However, if the analysis computes a small set $\mathcal{B}$ (however bigger or equal to the number of program variables), the VC generated does not win any concision (we are falling back to the separation given by the typed model).

Functor's properties: The requirements on the signature PA ensure that the operations domain, load and shift are sound. This enforces the soundness of definitions for the MME's operations. Based on this observation, we conjecture that these operations compute a sound post-condition relation, although this relation maybe not the strongest post-condition. A formal proof is left for future work.

5 Instances of Pointer Analysis Signature

The signature PA may be implemented by several existing pointer analyses. We consider three of them here and we show how they fulfill the requirements of PA. We also define an analysis which exploits the results of a precise pointer analysis to provide an appropriate partitioning of the memory in $\mathsf{PA}.\mathcal{B}$.

All pointer analyses we consider computes statically the possible values (i) of an address expression, i.e., an over-approximation of $[\![\mathtt{a}]\!]$ ($\mathtt{a} \in \mathtt{Aexpr}$ from Figure 4) and (ii) of an address dereference, i.e., an over-approximation of $[\![*\mathtt{a}]\!]$. For these reason, these analyses belong to the points-to analyses class [19].

5.1 Basic Analyses (B and $\mathbf{B}_\top$)

The first points-to analysis abstracts locations by a finite set of pairs $(\mathtt{v}, I^\#)$ built from a symbolic block identifier v and an abstraction for sets of integers $I^\#$ collecting the possible offsets of the location in the symbolic block. If we fix $\mathcal{I}^\#$ to be the abstract domain used to represents sets of integers, then the abstract domain for locations is defined by $Loc^\# \triangleq 2^{\mathtt{Cvar} \times \mathcal{I}^\#}$.

Many abstract domains have been proposed to deal with integer sets in abstract interpretation framework. For points-to analysis, most approaches use the classic domain of intervals [16]. To obtain more precise results, we consider here

$$\mathcal{L} \triangleq Loc^{\#} \quad \mathcal{S} \triangleq \texttt{Stmt} \to S^{\#} \tag{20}$$

$$\mathcal{B} \triangleq \texttt{Cvar} \quad \mathsf{base}(\mathtt{v}) \triangleq \{(\mathtt{v}, \{0\})\} \quad \mathsf{slice}(\mathtt{v}, e) \triangleq 0 \leq e < \mathtt{sizeof}(\mathtt{cty}(\mathtt{v})) \tag{21}$$

$$\mathsf{domain}(\ell^{\#}) \triangleq \{\mathtt{v} \mid (\mathtt{v}, I^{\#}) \in \ell^{\#}\} \tag{22}$$

$$\mathsf{shift}(\mathtt{s}, \ell^{\#}, e) \triangleq \sqcup^{\#}_{(\mathtt{v}_k, I^{\#}_k) \in \ell^{\#}} \{(\mathtt{v}_k, I^{\#}_k +^{\#} [\![e]\!]^{\#}(\mathtt{s}))\} \tag{23}$$

$$\mathsf{load}(\mathtt{s}, \mathtt{t\ ptr}, [\![\mathtt{a}]\!]^{\#}(\mathtt{s})) \triangleq [\![*\mathtt{a}]\!]^{\#}(\mathtt{s}) \tag{24}$$

Fig. 9: Implementation of PA by analyses $\mathbf{B}$ and $\mathbf{B}_\top$

the extension of the interval domain which also keeps modulo constraints and small sets of integers. This domain is implemented in the Eva plugin of Frama-C [22]. Then, the abstract sets in $\mathcal{I}^{\#}$ are defined by the following grammar:

$$\mathcal{I}^{\#} \ni I^{\#} ::= \top \mid [i_\infty..i'_\infty]r\%n \mid \{i_1, \ldots, i_n\} \tag{19}$$

where $r, n \in \mathbb{N}$ are natural constants, $i_1, \ldots, i_n \in \mathbb{Z}$ are integer constants and $i_\infty, i'_\infty \in \mathbb{Z} \cup \{+\infty, -\infty\}$ are integer constants extended with two symbols to capture unspecified bounds. We wrote $[i_\infty..i'_\infty]$ for $[i_\infty..i'_\infty]0\%1$. The concretization of a value $I^{\#}$ in $\mathcal{I}^{\#}$, $\gamma : \mathcal{I}^{\#} \to 2^{\mathbb{Z}}$ maps $[i_\infty..i'_\infty]r\%n$ to the set of integers $k \in [i, i']$ such that $k\%n = r$. Because the abstract intervals are used to capture offsets in symbolic blocks which have a known size (given by the ABI), the concrete offsets are always bounded, but they may be very large. We obtain independence of the ABI by introducing unspecified bounds for intervals and the $\top$ value. For efficiency, the size of explicit sets $\{i_1, \ldots, i_n\}$ is kept bounded by a parameter of the analysis, denoted in the following **ilvl**. The domain $\mathcal{I}^{\#}$ comes with lattice operators (e.g., join $\sqcup^{\#}$) and abstract transformers for operations on integers. Our work requires a sound abstract transformer for addition, $+^{\#}$.

Precise offsets ($\boldsymbol{B}$): Let us consider a precise instance of such an analysis, i.e. field-sensitive and employing the abstract domain of intervals $\mathcal{I}^{\#}$ defined above. Let $S^{\#}$ be the abstract domain for program's states implemented in this analysis. This domain captures the abstract values for all program's variables. We denote by $[\![\mathtt{a}]\!]^{\#}(\mathtt{s})$ the abstract location (in $Loc^{\#}$) computed by the analysis for the address expression $\mathtt{a}$ at statement $\mathtt{s}$. For address expressions typed as pointer to pointer types, the abstract value of the address expression $[\![*\mathtt{a}]\!]^{\#}(\mathtt{s})$ is also an element of $Loc^{\#}$ and computes the points-to information.

The types and operations of PA are shown in Figure 9. The symbolic blocks are not partitioned, since $\mathcal{B} \triangleq \texttt{Cvar}$. Then, the slice for a block is the set of valid offsets for the symbolic block and the generated constraint is very simple. Abstract locations are shifted precisely using the abstract transformer for addition in $\mathcal{I}^{\#}$. It is usually precise when e is a constant. The soundness properties required by PA are trivially satisfied due to the simple form of abstract blocks' type and the soundness of operations on the abstract domains used.

Imprecise offsets ($\boldsymbol{B}_\top$): We also consider an instance of the points-to analysis which is not field-sensitive. For example, the $\mathbf{B}_\top$ analysis computes for $[\![\,\texttt{\&SORT.out2}\,]\!]^{\#}(\mathtt{s}_3)$, where $\mathtt{s}_3$ is the assignment at line 3 of listing in Figure 3(a), the set of abstract location $\{(\mathtt{df}i, \top), \ldots, (\mathtt{pf}j, \top) \mid 1 \leq i \leq 8, 1 \leq j \leq 4\}$. The definition of the elements of the signature PA is exactly the one given in Figure 9.

5.2 Partitioning by Cells (C)

Analyzers that do not handle aggregate types (arrays and structs) decompose the symbolic blocks of variables having aggregate types into atomic blocks that all have a scalar type. We call these blocks *cells*. For examples, the symbolic block of variable `pf` in Figure 3(b) is split into four cells of type `pos_t`. For this analysis, the definitions for PA are those given in Figure 9 except for the type $\mathcal{B}$ and the operations using this type slice and domain. To define $\mathcal{B}$, we first define the set $\mathcal{C}(\mathtt{t})$ of *cells-paths* of type $\mathtt{t}$ by induction on the syntax of $\mathtt{t}$ as follows:

$$\mathcal{C}(\mathtt{t}) \triangleq \begin{cases} \{\epsilon\} & \text{if } \mathtt{t} \in \mathtt{Styp} \\ \bigcup_{1\leq i\leq n} \mathtt{f}_i \cdot \mathcal{C}(\mathtt{t}_i) & \text{if } \mathtt{t} \text{ is the record type } \{\mathtt{f}_1 : \mathtt{t}_1, \ldots, \mathtt{f}_n : \mathtt{t}_n\} \\ \bigcup_{0\leq i<n} [i] \cdot \mathcal{C}(\mathtt{t}_e) & \text{if } \mathtt{t} \text{ is the array type } \mathtt{t}_e[n] \end{cases}$$

where the operator "$\cdot$" prefixes each path of its second operand by its first operand. For a variable $\mathtt{v}$, we define $\mathcal{C}(\mathtt{v}) = \mathtt{v} \cdot \mathcal{C}(\mathtt{cty}(v))$. For example in Figure 3)(b), $\mathcal{C}(\mathtt{df}) = \{\mathtt{df}\cdot[0], \ldots, \mathtt{df}\cdot[7]\}$. Given a cell-path c, we denote by $r(c)$ the range of offsets (in bytes) that correspond to the path and which is computed using ABI. Then, we replace definitions in equation (21-22) from Figure 9 by:

$$\mathcal{B} \triangleq \{\mathcal{C}(\mathtt{v}) \mid \mathtt{v} \in \mathtt{Cvar}\} \qquad \mathsf{slice}(\mathtt{v}\cdot c, e) \;\triangleq\; e \in r(c)$$
$$\mathsf{domain}(\ell^{\#}) \triangleq \{\mathtt{v}\cdot c \in \mathcal{B} \mid \exists i \in \mathbb{N}, (\mathtt{v}, i) \in \gamma(\ell^{\#}) \wedge i \in r(c)\}$$

meaning that the slice of a cell-path is given by the range of bytes corresponding to the cell, and the domain of an abstract location is defined by enumerating all cells that intersect with abstract location's abstract offsets.

5.3 Partitioning by Dereference Analysis (P)

We have seen in Section 4.2 that the size of generated VC strongly depends on two factors: the size of $\mathcal{B}$ and the number of abstract blocks in the domain of abstract locations. This section defines an analysis which, based on the results of **B**, aims to minimize these two factors while still producing sound results. Roughly, the idea is to group cells that are accessed by a set of left values which is upwards-closed w.r.t. the relation "points-to" computed by **B**. Therefore, two different abstract blocks will never be pointed-to by the same left value, i.e., if the domains of abstract locations $[\![*a_1]\!]^{\#}(\mathtt{s}_1)$ and $[\![*a_2]\!]^{\#}(\mathtt{s}_2)$ share an abstract block $b^{\#}$, then $[\![a_1]\!]^{\#}(\mathtt{s}_1)$ and $[\![a_2]\!]^{\#}(\mathtt{s}_2)$ belong to the same block.

For this, we define a partition P of *pointer-typed left-values* used by statements of the current context call using the equivalence relation $\simeq$ defined as follows. We denote by $\ell^{\#} \downarrow_n$ the set of concrete locations $\gamma(\ell^{\#} +^{\#} 0) \cup \ldots \cup \gamma(\ell^{\#} +^{\#} n-1)$. Then, two left-values appearing in some statements are related by $\simeq$ if the concretization of the abstract locations computed by **B** for their addresses on the corresponding statements overlap. Formally, for any left-values $\mathtt{lv}_1$ and $\mathtt{lv}_2$ used in statements $\mathtt{s}_1$ resp. $\mathtt{s}_2$,

$$\left([\![(\&\mathtt{lv}_1)^{\mathtt{t}_1}]\!]^{\#}(\mathtt{s}_1) \downarrow_{n_1}\right) \bigcap \left([\![(\&\mathtt{lv}_2)^{\mathtt{t}_2}]\!]^{\#}(\mathtt{s}_2) \downarrow_{n_2}\right) \neq \emptyset \implies (\mathtt{lv}_1, \mathtt{s}_1) \simeq (\mathtt{lv}_2, \mathtt{s}_2)$$

where $n_i = \texttt{sizeof}(\texttt{t}_i)$. By definition, this relationship is reflexive and symmetric, and we close it transitively. It is computed by a simple iterative process on top of the results of **B** analysis. For a given element $p \in P$, we compute the set of concrete locations pointing to left-values in p:

$$B(p) \triangleq \bigcup_{(\texttt{lv}_i, \texttt{s}_i) \in p} \gamma([\![\&\texttt{lv}_i]\!]^{\#}(\texttt{s}_i))$$

Analysis **P** implements signature PA using the definitions in Figure 9 except for (21-22) that are replaced by:

$$\begin{aligned} \mathcal{B} &\triangleq \{\langle \texttt{v}, s\rangle \mid \exists p \in P \wedge s = \{i \mid (\texttt{v}, i) \in B(p)\}\} \\ \mathsf{slice}(\langle \texttt{v}, s\rangle, e) &\triangleq e \in s \\ \mathsf{domain}(\ell^{\#}) &\triangleq \{\langle \texttt{v}, s\rangle \in \mathcal{B} \mid \exists i \in \mathbb{N}, (\texttt{v}, i) \in \gamma(\ell^{\#}) \wedge i \in s\} \end{aligned}$$

In the example on Figure 3(b), if **B** is precise enough, **P** computes a $\mathcal{B}$ which splits the symbolic block labeled by the array variable `df` into (only) two abstract blocks: one for the bytes located at indexes $[1..4]$ (whose addresses are stored in input fields) and another for indexes $\{0\} \cup [5..7]$ (stored in output fields).

6 Experimental Results

6.1 Implementation

We implemented our framework in Frama-C [22], an extensible and modular platform for the analysis of software written in C. Frama-C includes various plug-ins, interacting with each other through interfaces defined by the platform.

The plug-in Eva is a context-sensitive static analyzer based on abstract interpretation; it employs several numerical abstract domains, including the one defined in eq. (19) for sets of integers. On top of the value analysis provided by Eva, which includes the **B** analysis from Section 5, we coded new partition analyses to obtain analyses $\mathbf{B}_{\top}$, **C** and **P**.

The WP plug-in of Frama-C is a DV tool which also includes a built-in simplifier for formulae, Qed [14], a driver to call SMT solvers and the signature MME for memory model environments [15]. We coded in WP the signature PA, the functor defined in Section 4.2, and each implementation of PA for the above static analyses. The full development represents 1680 LoC of Ocaml.

6.2 Experimental setup

Case study: We consider a case study which extends our running example from Figure 2 such that the type `data_t` is a record which encapsulates numerical values to be sorted and other information. We attempt to prove the functional correctness of the `sort` function for various number of inputs $\mathbf{N} \in \{4, 8, 16, 32\}$. The specification of `sort` consists of 40 ACSL properties, which WP transforms into 62 VC for each memory model. We also consider 3 different context calls

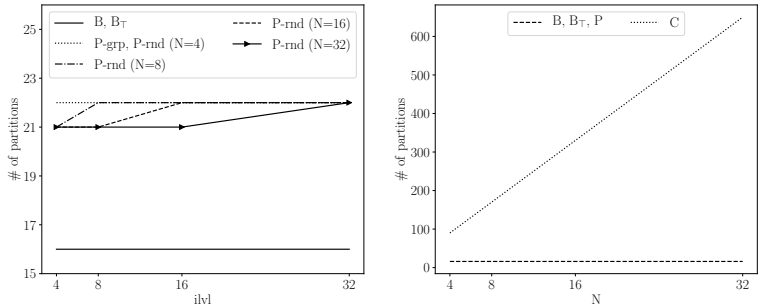

Fig. 10: Comparison between analyses on number of partitions

for `sort` as the entry point for the analysis. They initialize the fields of the `SORT` variable using pointers to: variables on the stack similar to Figure 3(a) (**vars**), fields of a single record (**strct**) and two arrays (for values and permutations) (**arrs**). In addition, we consider two variants for contexts **strct** and **arrs**. In the (**grp**) variant, all input and output fields are grouped together, i.e., inputs point to the first **N** fields/indexes in a regular way and outputs to the remainder. For the (**rdn**) variant, inputs and outputs are initialized in a randomized order, as in Figure 3(b) for **arrs**. The latter case is designed to defeat points-to analyses where offsets are abstracted solely by intervals plus congruences.

Variants of memory models: For comparison with the basic DV tools, we also conduct proof using the default memory model of WP (case **Typed**). To observe the influence of the precision of points-to analysis **B** on the generated memory models environments, we vary the parameter **ilvl** which gives the upper limit for the size of small sets kept by the abstract domain $\mathcal{I}^{\#}$ in Section 5.1. We apply **B** for **ilvl** in $\{4, 8, 16, 32\}$ to generate its memory model environment and the VC. For the same values, we launch the **C** (resp. **P**) analysis after **B** and generate the corresponding environments.

Proving VCs: WP generates VC using the library for many sorted first-order logic provided by Qed. After applying on-the-fly simplifications of VCs, Qed exports the VC to back-end solvers. We configure WP to discharge simplified VCs to the Alt-ergo prover and the remaining unproved VCs to be sent to CVC4. Those experiments ran on an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz with a timeout of 10 seconds per goal for each solver.

6.3 Results

Figure 10 shows the number of partitions (size of $\mathsf{PA}.\mathcal{B}$) inferred by the various analysis for a given call context. Recall that the partitioning generated by **B** is always constant, since fixed by the program variables. As expected, **C**'s result is linear in the number of inputs (right plot in 10). The partitioning by **P** creates fewer abstract blocks when **N** is less than **ilvl** (left plot in 10). Fewer blocks means a less precise analysis: in our example, the two equivalence classes that get merged are those corresponding to inputs and outputs.

Figure 11 (left) shows that **B** partitioning is sufficient to prove all goals for the **vars** context, since all values are implicitly separated onto different symbolic

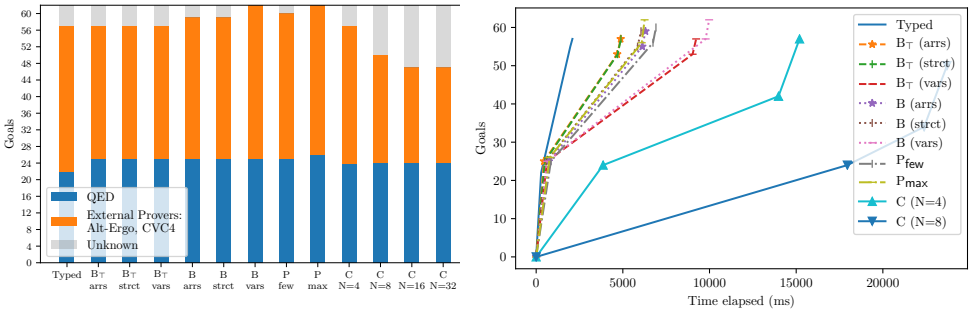

Fig. 11: Results for solving generated VC

bases. However, for contexts **strct** and **arrs**, inputs and outputs share the same symbolic base which is too imprecise to prove all goals. Analysis $\mathbf{B}_\top$ infers that the fields of `SORT` point to all possible inputs and outputs, which yields even worse results. The results for **C** partitioning worsen with the increase in the number of inputs due to the complexity of the VCs generated. For **P** partitioning, we are not interested in the **vars** context considering it is but a small refinement of **B** in that context. In our experiments, we were able to identify two classes of experiments giving similar results in term of provability and time: $\mathbf{P}_{\text{max}}$ are results for experiments where partitions are maximal and conversely for $\mathbf{P}_{\text{few}}$. For readability reasons, we display only the worse results of those two classes.

Figure 11 (right) shows for each model the total time spent on the VCs that get proven (i.e., do not timeout), and the total number of proven goals. For an equal number of proven goals, shorter times are better. We observe that more partitions lead to bigger VCs which take more time to be proven, especially for **C** partitioning. Refining **B** partitioning within **P** leads to a better provability at the cost of a negligible increase in time in provers. Indeed, we are interested in proving all VCs since some goals (shown as valid) implicitly assume that other goals are verified. These results demonstrate that **P** analysis offers the best trade-off between partition's granularity and provability in reasonable time, regardless of the context. Moreover, all verification conditions are proved for the regular context; for randomized contexts, better results are obtained by increasing the precision of points-to analysis **B**. The improvement of **P** is real because **B** exhibits the same performance only for the **vars** context.

7 Related Work and Conclusion

Memory Model for C: Program verification and certified compilation have proposed several memory models to capture the semantics of C pointers. All these models view the memory as a collection of disjoint regions. Two main classes may be distinguished: (i) the regions are typed by the value stored, therefore regions storing values of different types are disjoint and (ii) the regions are seen as raw arrays of bytes to capture low-level manipulations of memory in C. The first class provides a good abstraction for verification of type-safe languages, (e.g., Java-like [2,1], HOL [27]) or type-safe C programs (GRASSHoper [30], HIP/Sleek [13]). The second class is mainly used inside static analyzers for

C (Infer [10], MemCAD [11], Eva [8]) or deductive verifiers (Caduceus [17], HAVOC [12], SMACK [31], VCC [5], VeriFast [21]). Hybrid memory models either introduce typing in raw memory models for efficiency, or introduce raw models in typed ones for precision. WP supports both classes of models and provides instances of the environment MME for them [15].

The CompCert project [25,24] employs an abstract memory model to capture in an uniform way refinements of memory models for the certified compilation of C. This work also inspired [33], which surveys several concrete memory models for C and proposes a method to design static analyzers based on abstract memory models. Eva is not built following these principles for efficiency reasons.

Separation Logic versus FOL: Separation Logic [29] is used in many verification tools for C (e.g., GRASSHoper, HIP/Sleek, Infer, VCC, VeriFast) due to the efficiency of local reasoning. The specification logic used in Frama-C, ACSL [3], includes a separating conjunction operator (understood by WP and Eva plugins), but it is far weaker than the standard separating conjunction operator. The underlying solvers for SL of the above tools are either not available or deal with the type safe fragment of C. The recent SL-COMP initiative motivated the development of several independent solvers for type safe fragments of SL, one of them included in the CVC4 [32] solver. Our work focuses on DV tools using FOL and infers separation properties between memory regions. Our pointer analyses may be used in SL-based tools to obtain precise properties on arrays of pointers.

Pointer Analyses for DV: Static analysis based on region inference is used in [20] to partition a typed memory model. The analysis is less precise than the points-to analysis in Eva because the loss of precision for one location could force many precise locations to be collapsed in the same region. [31] employs pointer analysis to ensure a sound usage of the typed memory model in presence of casts. This work may be applied to extend the class of programs we deal with, but our focus is on improving efficiency of DV, not its realm. Recent work [36] proposes a precise points-to analysis to infer separation information in order to decrease the size of VCs. Although Eva is doing a less precise analysis, it is still able to infer such separation properties. In addition, we define a formalized channel to transfer such information to DV tools. The authors of [5] explore different memory models to generate with VCC a benchmark of problems for SMT solvers. By implementing various memory models for WP, we increase such benchmark.

Conclusion: We have formalized the collaboration of a pointer analysis tool and a deductive verification tool by a functor which exploits the results of the pointer analysis to define sound and precise memory model environments used in the generation of verification conditions in first order logic theories. We applied this functor to several pointer analyses, including classic analyses (points-to analysis) and a new analysis that allows to obtain precise partitioning information of the program's memory. We reported on the implementation of the functor in Frama-C and on the results obtained by different analyses on a benchmark of C programs that exhibit complex features of pointers in C (arrays of pointers, duality of fields) and complex separation annotations. The results obtained show the interest of our functor for the automatization of deductive verification.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO*, number 4111 in LNCS, pages 364–387. Springer, 2005.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS*, LNCS, pages 49–69. Springer, 2004.
3. P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.
4. S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, Oct. 2009.
5. S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. In *Proceedigns of CADE-23*, volume 6803 of *LNCS*, pages 177–191. Springer, 2011.
6. R. Bornat. Proving pointer programs in Hoare logic. In *Proceedings of MPC*, volume 1837 of *LNCS*, pages 102–126. Springer-Verlag, 2000.
7. J. Brotherston and M. Kanovich. On the Complexity of Pointer Arithmetic in Separation Logic (an extended version). *arXiv:1803.03164 [cs]*, Mar. 2018. arXiv: 1803.03164.
8. D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions*. PhD thesis, University of Rennes, 2017.
9. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
10. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.
11. B.-Y. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *Proceedings of SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.
12. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A low-level memory model and an accompanying reachability predicate. *STTT*, 11(2):105–116, February 2009.
13. W. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
14. L. Correnson. Qed. computing what remains to be proved. In *Proceedings of NFM*, volume 8430 of *LNCS*, pages 215–229. Springer, 2014.
15. L. Correnson and F. Bobot. Exploring memory models with Frama-C/WP, 2017. Personal comunication.
16. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
17. J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
18. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, Jan. 2001.
19. M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of PASTE*, pages 54–61, Snowbird, Utah, United States, 2001. ACM Press.

20. T. Hubert and C. Marché. Separation analysis for deductive verification. In *Proceedings of HAV*, pages 81–93, Braga, Portugal, Mar. 2007.
21. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
22. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
23. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
24. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.
25. X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008.
26. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
27. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Proceedings of CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
28. A. Miné. Static analysis by abstract interpretation of concurrent programs. Technical report, École normale supérieure, May 2013. `http://www-apr.lip6.fr/~mine/hdr/hdr-compact-col.pdf`.
29. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
30. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Proceedings of CAV*, pages 711–728. Springer, 2014.
31. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *Proceedings of VMCAI*, volume 5403 of *LNCS*, pages 290–304. Springer, 2009.
32. A. Reynolds, R. Iosif, C. Serban, and T. King. A Decision Procedure for Separation Logic in SMT. In *Proceedings of ATVA*, volume 9938 of *LNCS*, pages 244–261, Cham, 2016. Springer International Publishing.
33. P. Sotin, B. Jeannet, and X. Rival. Concrete memory models for shape analysis. *Electr. Notes Theor. Comput. Sci.*, 267(1):139–150, 2010.
34. Stackoverflow. Is it legal to access struct members via offset pointers from other struct members? *https://stackoverflow.com/questions/51737910/*. Last consulted: October 5th, 2018.
35. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108. ACM, 2007.
36. W. Wang, C. Barrett, and T. Wies. Partitioned memory models for program analysis. In *Proceedings of VMCAI*, volume 10145 of *LNCS*, pages 539–558. Springer, 2017.