Soundness of a Dataflow Analysis for Memory Monitoring

Dara Ly, Nikolai Kosmatov, Julien Signoles, Frédéric Loulergue

Soundness of a Dataflow Analysis for Memory Monitoring

Dara Ly
CEA List
Software Reliability and Security Lab
Gif-sur-Yvette, France
Université d'Orléans
LIFO EA 4022
Orléans, France
firstname.name@cea.fr

Nikolai Kosmatov
Julien Signoles
CEA List
Software Reliability and Security Lab
Gif-sur-Yvette, France
firstname.name@cea.fr

Frédéric Loulergue
Northern Arizona University
School of Informatics Computing and Cyber Systems
Flagstaff, Arizona, USA
firstname.name@nau.edu

ABSTRACT

An important concern addressed by runtime verification tools for C code is related to detecting memory errors. It requires to monitor some properties of memory locations (e.g., their validity and initialization) along the whole program execution. Static analysis based optimizations have been shown to significantly improve the performances of such tools by reducing the monitoring of irrelevant locations. However, soundness of the verdict of the whole tool strongly depends on the soundness of the underlying static analysis technique. This paper tackles this issue for the dataflow analysis used to optimize the E-ACSL runtime assertion checking tool. We formally define the core dataflow analysis used by E-ACSL and prove its soundness.

KEYWORDS

dataflow analysis, memory monitoring, runtime assertion checking, proof of soundness, formal semantics, E-ACSL tool

ACM Reference Format:
Dara Ly, Nikolai Kosmatov, Julien Signoles, and Frédéric Loulergue. 2018. Soundness of a Dataflow Analysis for Memory Monitoring. In *Proceedings of Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT 2017).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context. Memory related errors can provoke serious defects in software [5]. A study based on IBM MVS software [26] reports that about 50% of detected software errors were related to pointers and array accesses. Memory errors account for about 50% of reported security vulnerabilities [27]. This is particularly an issue for a programming language like C that is both the most commonly used for development of critical system software and one of the most poorly equipped with adequate protection mechanisms. The developer remains responsible for correct allocation and deallocation of memory, initialization of variables, pointer dereferencing and manipulation (using casts, offsets, etc.), as well as for the validity of indices in array accesses.

Among the most useful techniques for detecting and locating software errors, *runtime assertion checking* has become a widely-used programming practice [6]. Runtime checking of memory-related properties can be realized using systematic monitoring of memory operations. However, to do so efficiently is difficult, because of the large number of memory accesses in modern programs.

This paper more specifically considers the problem of memory monitoring of C programs for runtime assertion checking in the context of Frama-C [14], a platform for analysis of C code. Frama-C offers an expressive executable specification language E-ACSL, and comes with a runtime assertion checking plugin, also called E-ACSL [8]. The E-ACSL plugin takes as an input a C program P annotated with an E-ACSL specification, and outputs an instrumented program P' having the following properties:

- if the execution of P satisfies all the properties expressed in the specification, the functional behavior of P' is the same as that of the original program P;
- whenever the execution of P violates a property of the specification, the program P' is aborted and an error is signaled.

This is done by translating the given specification into C code: the generated code implements a monitor verifying at runtime the conformance of the program with regard to the specification. In order to support memory-related E-ACSL annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.), the instrumented program P' needs to keep track of relevant memory operations previously executed by the program.

Motivation. Previous work demonstrated that the performances of a runtime verification tool can be significantly improved using a preliminary static analysis step [11]. A dedicated dataflow analysis can be used to compute an (over-approximated) set of relevant memory locations that should be monitored. All operations changing the status (i.e. validity, initialization) of these locations should be tracked. All other locations are irrelevant: the monitoring tool does not need to monitor them. For the E-ACSL tool, this technique leads to important performance savings that vary between 60% and 73% [11].

However, this optimization can alter the correctness of the whole tool. Indeed, if the dataflow analysis is not sound, the monitoring of some relevant locations can be missed and their status can be wrongly identified. Since an E-ACSL annotation can check both for a pointer validity or its negation, this obviously can lead both

HILT 2017, November 5-6, 2018, Boston, MA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to false positives (an annotation failure is wrongly reported) and false negatives (a real annotation failure is not reported). For a tool like E-ACSL— whose goal is to check a formal specification and that is in particular used in combination with sound static analysis techniques — this lack of soundness guarantees is an important issue. Thus, to ensure that runtime assertion checking always provides the correct verdict, the soundness of the underlying dataflow analysis should be formally established.

Contributions. The main contributions of this paper include a formal definition of a core dataflow analysis used by the E-ACSL tool and a proof of soundness of this analysis. Another contribution is an operational semantics of the monitor that naturally extends a standard operational semantics. We think that it is particularly suitable when reasoning about memory monitoring tools such as E-ACSL and could be reused in future work.

Outline. The paper is organized as follows. Section 2 motivates our work. Section 3 introduces the considered programming and specification languages, in particular their formal semantics. The dataflow analysis is defined in Section 4. Section 5 states and proves the soundness property of the analysis. Section 6 presents related work, and Section 7 gives concluding remarks and future work.

2 MOTIVATING EXAMPLE

Consider the toy C function at the left of Fig. 1. It contains a formal assertion guaranteeing that dereferencing pointer `p` is safe. This kind of assertions may be automatically generated by Frama-C [14].

The E-ACSL tool [23] is able to take this annotated function as input in order to generate the instrumented version at the right of Fig. 1 (where we assume a 32-bit architecture). If executed, this instrumentation nicely stops the execution whenever the assertion is violated, preventing an incorrect memory access. To be able to check at runtime this kind of memory-related annotations, the code generated by E-ACSL records memory operations in the E-ACSL dynamic memory model linked to the instrumented program [28]. For instance, every allocation (resp. deallocation) is recorded through a call to function `store_block` (resp. `delete_block`), while every assignment is recorded through a call to function `full_init`.

However, even if the E-ACSL dynamic memory model is heavily optimized, these additional instructions significantly slow down the execution [28]. Fortunately, a complete instrumentation is almost never necessary. Here, for instance, only `p` needs to be monitored. Monitoring memory locations not relevant to `p` is not useful. In this example, when the annotation is evaluated, `p` is equal to the address of `x` because of the alias created by the first assignment.

E-ACSL implements an over-approximating dataflow analysis to compute what needs to be monitored. It takes care of aliasing and has already been proved capable to significantly improve the efficiency of the generated code [11]. Here, for instance, all the lines of code in gray may soundly be discarded, but the fact that `x` is properly allocated must still be monitored.

The goal of our work is to formalize the core of this analysis in order to prove its soundness. Indeed, if unsound, it would lead to incorrect verdict by E-ACSL. Here, if the line `store_block(&x, 4)` is omitted, E-ACSL would incorrectly conclude that dereferencing pointer `p` is unsafe. Even worse, in other contexts, it could conclude that dereferencing a pointer would be safe while it is not, leading to security weaknesses.

3 LANGUAGE DEFINITION

E-ACSL takes as an input a C program annotated with E-ACSL specifications [8]. The whole language is far too complex to be handled formally in this paper. Rather, we present our analysis, its formalization and its proof of correctness on a relevant subset of C annotated with E-ACSL specifications. This subset is large enough to present the main issues tackled in our analysis.

3.0.1 Syntax. Figure 2 presents the syntax of our language. Expressions contain arithmetic expressions and pointer accesses (addresses and offset shifts), and are pure (i.e. side-effect free). Since our analysis focuses on memory management, we only detail memory values here.

Basic statements are assignments of an expression to a left-value (which can be either a variable or a pointer access), memory allocations and deallocations, and assertions. Compound statements are sequences, conditionals and loops.

Assertions evaluate terms of a propositional calculus with integer arithmetic and predicates over memory locations. Terms and predicates notably contain five built-in logic functions and predicates (detailed in Fig. 3) usable to express a rich set of memory-related properties [11].

3.0.2 Memory model. Since our language features manual memory management via allocation and deallocation, its formal semantics relies on a memory model that specifies read and write accesses to the memory, here called *execution memory* (or, in short, *memory* whenever the context is clear enough).

Execution Memory. Since our language is a simple C-like language, we use a simplified CompCert memory model [16]. Indeed, the CompCert memory model was developed for the purpose of defining a semantics for C. In this model, the memory is viewed in an abstract manner as a collection of *blocks*. Physical placement of blocks and their relative position is not modeled. A block contains a range of valid *offsets* within it and is entirely defined by its bounds at allocation time. Consequently, a valid *address* is defined by an ordered pair (b, δ) of a block and an offset. In the following, `mem` denotes the type of memory states, while `block` denotes that of blocks. A memory state associates addresses to *values*. These values have the type `val` defined as follows:

$$v \quad ::= \quad \texttt{Int}(n) \quad | \quad \texttt{Ptr}(b, \delta) \quad | \quad \texttt{Undef}$$

Here, `Undef` corresponds to the initial value at any address that was not yet written to.

Four operations may be performed on a given memory state: *allocation*, *deallocation*, *load* and *store*, respectively defined through the operations `alloc`, `dealloc`, `load` and `store` below.

Assume a memory state M, a block b, three integers lo, hi, δ and a value v. Statement $\texttt{alloc}(M, lo, hi)$ allocates a new block in M, with lower bound lo (inclusive) and higher bound hi (exclusive), and returns the updated memory state with the new block. Statement $\texttt{dealloc}(M, b)$ deallocates the block b in M, returning the updated memory state where b has bounds $(0, 0)$, which makes it impossible to write to or read from. We model the fact that some operations

```
int f(void) {
  int x, y, z, *p;
  p = &x;
  x = 0;
  y = 1;
  z = 2;
  /*@ assert
      \valid(p); */
  *p = 3;
  return x;
}
```

```
int f(void) {
  int x, y, z, *p;
  store_block(&p, 4);
  store_block(&z, 4);
  store_block(&y, 4);
  store_block(&x, 4);
  full_init(&p);
  p = &x;
  full_init(&x);
  x = 0;
  full_init(&y);
  y = 1;
```

```
  full_init(&z);
  z = 2;
  /*@ assert \valid(p); */
  e_acsl_assert(valid(p,
            sizeof(int)));
  *p = 3;
  delete_block(&p);
  delete_block(&z);
  delete_block(&y);
  delete_block(&x);
  return x;
}
```

Figure 1: A C function (left) and its instrumented version (right).

fail (e.g. `dealloc` called on a block already deallocated) by using an *option type* as a return type. A value of type `memory option` either contains a memory state M (in which case the option value is denoted $\texttt{Some}(M)$), or contains nothing (denoted `None`). Statement $\texttt{load}(M, b, \delta)$ reads the value at address (b, δ) in M, if possible, and returns it. Again, this operation can fail, e.g., if the supplied address is not valid. Finally, $\texttt{store}(M, b, \delta, v)$ writes the value v at address (b, δ) in M if possible, and fails otherwise. If successful, the function returns the memory state updated with the new value. Besides these four operations, the model also provides a function `bounds` giving the bounds of a block b in a memory state M: $\texttt{bounds}(M, b) = (lo, hi)$

left values	$l ::=$	x	variable
	$\mid$	$\star a$	pointer access
mem. values	$a ::=$	l	left value
	$\mid$	a ++ e	pointer offset
	$\mid$	$\&l$	address
expressions	$e ::=$	a	memory value
	$\mid$	$\cdots$	arithmetic expr.
statements	$s ::=$	`skip;`	skip
	$\mid$	l = e;	assignment
	$\mid$	l = `malloc`(n);	allocation
	$\mid$	`free`(l);	deallocation
	$\mid$	/*@ `assert` p; */	assertion
	$\mid$	`if` (e) `then` s `else` s	conditional
	$\mid$	`while`(e) s	loop
	$\mid$	s s	sequence
predicates	$p ::=$	$t \equiv t \mid t \leq t$	comparators
	$\mid$	$p \wedge p \mid p \vee p \mid \neg p$	logic connectors
	$\mid$	`\valid`(a)	a valid pointer
	$\mid$	`\init`(a)	$\star a$ initialized
terms	$t ::=$	e	pure expressions
	$\mid$	`\base_addr`(a)	base addr. of a's mem. block
	$\mid$	`\block_length`(a)	size of a's mem. block
	$\mid$	`\offset`(a)	offset of a in its mem. block
	$\mid$	$\cdots$	pure expr. combined with mem.-related constructs
types	$\tau ::=$	`int`	integral type
	$\mid$	$\tau\star$	pointer type

Figure 2: Formal syntax of the considered language.

Properties	Informal semantics
`\base_addr`(a)	base address of the block containing address `a`
`\block_length` (a)	size (in bytes) of the block containing address `a`
`\offset` (a)	offset (in bytes) of `a` in its block
`\valid`(a)	is true if and only if reading and writing the contents of `a` is safe
`\init`(a)	is true if and only if the contents of `a` has been initialized

Figure 3: Memory Built-ins.

where lo and hi are respectively the lower and higher bounds of b in M.

For the sake of simplicity, we suppose that all types have a size of one byte in memory, thus ruling out all considerations of alignment or overlapping memory accesses.

The axioms of this model are similar to those of the CompCert memory model [16]. They express algebraic properties allowing to reason about memory states, such as "load-after-store". As these properties are reasonably intuitive, we omit them here.

In the following, we note $M \vDash (b, \delta)$ the proposition "(b, δ) is a valid address in the memory state M", meaning that:

(1) b was allocated and not (yet) deallocated in M; and
(2) $lo \leq \delta < hi$ where lo and hi are the bounds of b in M.

Observation Memory. In addition to the memory space in which the program executes, our language uses a second memory store, called *observation memory*, in order to record the pieces of information (often called *metadata*) about the execution memory blocks that are required to evaluate at runtime the memory built-ins of Fig. 3. This model is similar to that of the execution memory, but instead of storing data values, it records which memory blocks have been allocated, their size and their per-byte initialization status (i.e. which bytes are initialized or uninitialized).

Consequently, along with execution memory, we define another type `obs` to model such a representation of the execution memory used by a monitor to evaluate the memory-related predicates of our language. The set of operations associated to observation memory differs slightly from that of execution memory.

Assume an observation memory $\overline{M}$, a block b and three integers lo, hi, δ. Statement $\texttt{store\_block}(\overline{M}, b, lo, hi)$ records block b as allocated, with bounds lo and hi, and returns the updated

observation memory. Statement `delete_block`$(\overline{M}, b)$ marks b as deallocated in $\overline{M}$ and returns the updated observation memory. Statement `initialize`$(\overline{M}, b, \delta)$ marks the content of address (b, δ) as initialized and returns the updated observation memory. Finally, `read_init`$(\overline{M}, b, \delta)$ reads the initialization status of the content at address (b, δ), returning `true` if an `initialize` operation has been performed at this address before, and `false` otherwise. The semantics of the observation memory statements is permissive: they do not fail and silently ignore incorrect operations such as an attempt to delete a non-existing block or an attempt to record as initialized a non-existing memory location.

3.1 Operational semantics

This section introduces an operational semantics for our language. The evaluation of expressions draws inspiration from Clight [3], the source language of CompCert, while statements have a small-step semantics akin to that of a While language, as in [19] for instance.

A noticeable trait of our semantics is that it is *blocking*: any execution ends immediately (cannot continue further) if an assertion is violated. This property is indeed the fundamental property of Frama-C and E-ACSL [7].

The semantics is expressed by means of six forms of evaluation judgments:

- $M \vDash_{\text{lv}} l \Leftarrow (b, \delta)$ evaluates a left-value l to an address (b, δ) in a memory M;
- $M \vDash_{\text{e}} e \Rightarrow v$ evaluates an expression e to a value v in a memory M;
- $M \vDash_{\text{t}} t \Rightarrow v$ evaluates a term t to a value v in a memory M;
- $M \vDash_{\text{p}} p \Rightarrow P$ evaluates a predicate p to a truth value P in a memory M;
- $\langle s, M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})$ evaluates a statement s in an execution memory M_1 and an observation memory $\overline{M_1}$, then ends the execution in an execution memory M_2 and an observation memory $\overline{M_2}$.
- $\langle s_1, M_1, \overline{M_1}\rangle \rightarrow \langle s_2, M_2, \overline{M_2}\rangle$ also evaluates a statement $s1$ in an execution memory M_1 and an observation memory $\overline{M_1}$, then continues the execution with the statement s_2 in an execution memory M_2 and an observation memory $\overline{M_2}$.

We consider only execution of well typed programs (even if the simple type system is omitted here), thus equipped with a typing environment Γ. We also associate to a given program an environment E and initial memory states M_0 and $\overline{M_0}$. The environment E maps each variable of the program to a block in M_0, which is otherwise empty.

The evaluation of expressions and left-values is described in Figure 4. Left-values are terms at the left of an assignment, therefore their evaluation yields a memory location to be written to. As mentioned before, evaluating expressions does not alter memory: the expressions are pure. When evaluating a left-value as an expression, the left-value is resolved into a memory location, which is then read in the execution memory with a `load`. As specified in the C standard [10], reading an uninitialized value is an undefined behavior. Consequently, the evaluation is valid only if the retrieved value is not `Undef`.

$$\textsc{le-var}\ \frac{E(x) = b}{M \vDash_{\text{lv}} x \Leftarrow (b, 0)} \qquad \textsc{le-deref}\ \frac{M \vDash_{\text{e}} a \Rightarrow \texttt{Ptr}(b, \delta)}{M \vDash_{\text{lv}} \star a \Leftarrow (b, \delta)}$$

$$\textsc{ee-lval}\ \frac{v \neq \texttt{Undef} \qquad M \vDash_{\text{lv}} l \Leftarrow (b, \delta) \qquad \texttt{load}(M, b, \delta) = \texttt{Some}(v)}{M \vDash_{\text{e}} l \Rightarrow v} \qquad \textsc{ee-int}\ \frac{}{M \vDash_{\text{e}} n \Rightarrow \texttt{Int}(n)}$$

$$\textsc{ee-shift}\ \frac{M \vDash_{\text{e}} a \Rightarrow \texttt{Ptr}(b, \delta) \qquad M \vDash_{\text{e}} e \Rightarrow \texttt{Int}(n)}{M \vDash_{\text{e}} a \texttt{++} e \Rightarrow \texttt{Ptr}(b, \delta + n)} \qquad \textsc{ee-addr}\ \frac{M \vDash_{\text{lv}} l \Leftarrow (b, \delta)}{M \vDash_{\text{e}} \&l \Rightarrow \texttt{Ptr}(b, \delta)}$$

$$\textsc{ee-op}\ \frac{M \vDash_{\text{e}} e_1 \Rightarrow \texttt{Int}(n_1) \qquad M \vDash_{\text{e}} e_2 \Rightarrow \texttt{Int}(n_2)}{M \vDash_{\text{e}} e_1\ \texttt{op}\ e_2 \Rightarrow \texttt{Int}(n_1\ \texttt{op}\ n_2)}$$

where $\texttt{op} \in \{+, -, *, /, ==, <=\}$

$$\textsc{ee-negNz}\ \frac{M \vDash_{\text{e}} e \Rightarrow \texttt{Int}(n) \qquad n \neq 0}{M \vDash_{\text{e}} !e \Rightarrow \texttt{Int}(0)} \qquad \textsc{ee-negZ}\ \frac{M \vDash_{\text{e}} e \Rightarrow \texttt{Int}(0)}{M \vDash_{\text{e}} !e \Rightarrow \texttt{Int}(1)}$$

Figure 4: Evaluation of expressions and left-values.

Predicates describe properties of the program at the current execution point. Their semantics is given in Figure 5, along with that of logical terms.

Statements can be broadly classified in two categories: those operating on the memory, and those defining the program's control flow. The former, described in Figure 6, modify the execution memory and record the corresponding operation in the observation memory. For instance an assignment (rule ASSIGN) stores the computed value at the memory location defined by the left-value, but also records in the observation memory that the assigned memory location is now initialized. Similarly, in MALLOC (resp. FREE) the addition (resp. deletion) of a block is recorded in the observation memory.

The semantics of control flow related statements, presented in Figure 7, is standard. Notice that an assertion (rule ASSERT) whose predicate is evaluated to `false` stops the execution.

4 DATAFLOW ANALYSIS

Our dataflow analysis aims at computing an over-approximation of the set of blocks needed to evaluate at runtime the predicates of a given program. As such it is reminiscent of liveness analysis, in the sense that it computes parts of memory that may be used later for some computations. However our problem differs significantly from the "canonical" liveness analysis such as in [19], in that our language has dynamic allocations and pointer aliasing. The main difficulty is to deal with this additional complexity while keeping the analysis simple enough to be proven sound.

We manage it by: providing the result of a preliminary points-to analysis to the present analysis, and making large over-approximations.

$$\textsc{te-ofs}\ \frac{M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)}{M \vDash_{\mathrm{t}} \backslash\mathtt{offset}(a) \Rightarrow \mathtt{Int}(\delta)}$$

$$\textsc{te-baddr}\ \frac{M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)}{M \vDash_{\mathrm{t}} \backslash\mathtt{base\_addr}(a) \Rightarrow \mathtt{Ptr}(b,0)}$$

$$\textsc{te-bsize}\ \frac{\begin{array}{c}M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)\\ \mathsf{bounds}(M,b) = (lo,hi)\end{array}}{M \vDash_{\mathrm{t}} \backslash\mathtt{block\_length}(a) \Rightarrow \mathtt{Int}(hi-lo)}$$

$$\textsc{te-expr}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow v}{M \vDash_{\mathrm{t}} e \Rightarrow v} \qquad \textsc{pe-valid}\ \frac{\begin{array}{c}M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)\\ M \vDash (b,\delta)\end{array}}{M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(a) \Rightarrow \mathtt{true}}$$

$$\textsc{pe-invalid}\ \frac{\begin{array}{c}M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)\\ M \nvDash (b,\delta)\end{array}}{M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(a) \Rightarrow \mathtt{false}}$$

$$\textsc{pe-init}\ \frac{\begin{array}{c}M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)\\ \mathsf{load}(M,b,\delta) = \mathtt{Some}(v)\\ v \neq \mathtt{Undef}\end{array}}{M \vDash_{\mathrm{p}} \backslash\mathtt{init}(a) \Rightarrow \mathtt{true}}$$

$$\textsc{pe-uninit}\ \frac{\begin{array}{c}M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b,\delta)\\ \mathsf{load}(M,b,\delta) = \mathtt{Some}(\mathtt{Undef})\end{array}}{M \vDash_{\mathrm{p}} \backslash\mathtt{init}(a) \Rightarrow \mathtt{false}}$$

$$\textsc{pe-eq}\ \frac{\begin{array}{c}M \vDash_{\mathrm{t}} t_1 \Rightarrow v_1\\ M \vDash_{\mathrm{t}} t_2 \Rightarrow v_2 \qquad v_1 = v_2\end{array}}{M \vDash_{\mathrm{p}} t_1 \equiv t_2 \Rightarrow \mathtt{true}}$$

$$\textsc{pe-neq}\ \frac{\begin{array}{c}M \vDash_{\mathrm{t}} t_1 \Rightarrow v_1\\ M \vDash_{\mathrm{t}} t_2 \Rightarrow v_2 \qquad v_1 \neq v_2\end{array}}{M \vDash_{\mathrm{p}} t_1 \equiv t_2 \Rightarrow \mathtt{false}}$$

Figure 5: Evaluation of terms and predicates.

$$\textsc{assign}\ \frac{\begin{array}{ll}M_1 \vDash_{\mathrm{e}} e \Rightarrow v & \mathsf{store}(M_1,b,\delta,v) = \mathtt{Some}(M_2)\\ M_1 \vDash_{\mathrm{lv}} l \Leftarrow (b,\delta) & \mathsf{initialize}(\overline{M_1},b,\delta) = \overline{M_2}\end{array}}{\langle l = e;\ , M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}$$

$$\textsc{malloc}\ \frac{\begin{array}{ll} & \mathsf{store\_block}(\overline{M_1},b',lo,hi) = \overline{M_3}\\ hi - lo = n & \mathsf{initialize}(\overline{M_3},b,\delta) = \overline{M_2}\\ M_1 \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n) & \mathsf{alloc}(M_1,lo,hi) = (M_3,b')\\ M_1 \vDash_{\mathrm{lv}} l \Leftarrow (b,\delta) & \mathsf{store}(M_3,b,\delta,\mathtt{Ptr}(b',0)) = \mathtt{Some}(M_2)\end{array}}{\langle l = \mathtt{malloc}(e);\ , M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}$$

$$\textsc{free}\ \frac{\begin{array}{c}M_1 \vDash_{\mathrm{e}} a \Rightarrow (b,0)\\ \mathsf{free}(M_1,b) = \mathtt{Some}(M_2) \qquad \mathsf{delete\_block}(\overline{M_1},b) = \overline{M_2}\end{array}}{\langle \mathtt{free}(a);\ , M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}$$

$$\textsc{assert}\ \frac{M \vDash_{\mathrm{p}} p \Rightarrow \mathtt{true}}{\langle /{*}@\ \mathtt{assert}\ p;\ {*}/\ , M, \overline{M}\rangle \rightarrow (M, \overline{M})}$$

Figure 6: Evaluation of assignments, allocations, deallocations, and assertions.

$$\textsc{seqCont}\ \frac{\langle s_1, M_1, \overline{M_1}\rangle \rightarrow \langle \hat{s}_1, M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \rightarrow \langle \hat{s}_1\ s_2, M_2, \overline{M_2}\rangle}$$

$$\textsc{seqEnd}\ \frac{\langle s_1, M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \rightarrow \langle s_2, M_2, \overline{M_2}\rangle}$$

$$\textsc{ifTrue}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n) \qquad n \neq 0}{\langle \mathtt{if}\ (e)\ \mathtt{then}\ s_t\ \mathtt{else}\ s_f, M, \overline{M}\rangle \rightarrow \langle s_t, M, \overline{M}\rangle}$$

$$\textsc{ifFalse}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(0)}{\langle \mathtt{if}\ (e)\ \mathtt{then}\ s_t\ \mathtt{else}\ s_f, M, \overline{M}\rangle \rightarrow \langle s_f, M, \overline{M}\rangle}$$

$$\textsc{whileCont}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n) \qquad n \neq 0}{\langle \mathtt{while}\ (e)\ s, M, \overline{M}\rangle \rightarrow \langle s\ \mathtt{while}\ (e)\ s, M, \overline{M}\rangle}$$

$$\textsc{whileEnd}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(0)}{\langle \mathtt{while}\ (e)\ s, M, \overline{M}\rangle \rightarrow (M, \overline{M})}$$

Figure 7: Evaluation of control flow related statements.

The latter leads to a quite imprecise analysis meeting one of our primary concerns to keep the E-ACSL compilation time reasonably fast. Therefore, in the balance between precision and speed of the analysis, we favor speed most of the time. However, even if imprecise, this analysis allows E-ACSL to improve significantly the efficiency of the generated monitor by reducing the instrumentation [11].

The analysis computes for each program point a certain set of variables which defines the zones of memory that will be instrumented for runtime monitoring purposes. At some program point l, any block of memory *reachable* (through dereferencing and offset shifts) from one of the variables computed at l is instrumented. Therefore, although the analysis is formally expressed in terms of variables, it may be useful to consider these variables as representing the set of all memory blocks reachable from them. Instrumenting all these blocks—when maybe only some of them would be strictly necessary—is one of the cases where we trade precision for speed in the analysis.

Reachability is formally defined by viewing memory as a graph where blocks are vertices, and pointers define edges. Following this intuition, we define the relation $\mapsto_M$ between blocks of a memory M as "given the memory state M, $b \mapsto_M b'$ if there is one value in b that is a pointer to b' with a valid offset". This definition is

formally expressed as follows:

$$b \mapsto_M b' \text{ if } \exists \delta, \delta' \in \mathbb{Z} \text{ such that } \begin{cases} M \vDash (b, \delta) \\ M \vDash (b', \delta') \\ \texttt{load}(M, b, \delta) = \texttt{Some}(\texttt{Ptr}(b', \delta')) \end{cases}$$

We define $\mapsto^\star_M$ to be the reflexive, transitive closure of $\mapsto_M$ and use it to define the set of reachable blocks from a memory value. If a is a value such that $M \vDash_e a \Rightarrow \texttt{Ptr}(b_a, \delta_a)$, the set of reachable blocks from a in M is defined by:

$$\mathcal{R}_M(a) = \left\{ b \in \textbf{Blocks} \mid b_a \mapsto^\star_M b \right\}$$

4.1 Definition

The analysis is classically defined on a labeled Control Flow Graph (CFG) where basic blocks are single statements (skip, assignment, allocation, deallocation, and assertion) and tests (guards of conditionals and loops).

For a given program s we note $\mathcal{I}(s)$ (resp. $\mathcal{F}(s)$) the label of the CFG's initial block (resp. the labels of the set of final blocks). In the example program s in Figure 8, we have $\mathcal{I}(s) = 1$ and $\mathcal{F}(s) = \{2\}$.

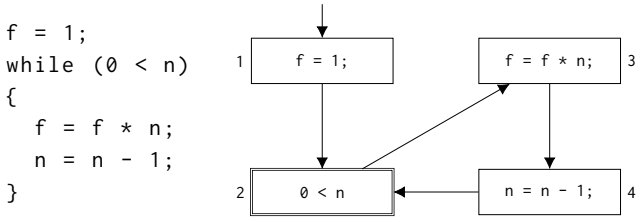

Figure 8: CFG of a simple program.

We define the analysis as a pair $(live_{in}, live_{out})$ of functions mapping each label of the CFG to a set of program variables. They are assumed to be the least solution of the constraint system presented in Figure 9. As we are solely interested in the *soundness* of the analysis, we do not study how this solution is computed.

For a given label l, $live_{in}(l)$ and $live_{out}(l)$ are the set of live variables respectively at the entry and the exit of the block labeled l. (To avoid any risk of confusion between labels and left values, in the remainder of the paper, we denote a left value by lv.) Like a liveness analysis, the analysis is *backward*: the information associated to a block is computed from that of its successors in the CFG. Like a liveness analysis, it is *over-approximating*: it combines together information originating from multiple branches using the union operator.

The function *gen* defines how statements generate live variables; while most statements simply transfer the existing liveness information, new liveness information is generated in two cases.

First, predicates generate the base variables of their arguments, as described by the recursive functions ρ and θ. The *base variable* of a memory value a is the first variable appearing in a's syntactical structure. It is, very concretely, the first variable written, in left-to-right order, when writing the source code for a. The base variable is computed by the function base, inductively defined as following:

$$\begin{array}{rclcrcl} \texttt{base}(x) & = & x & \qquad & \texttt{base}(\star a) & = & \texttt{base}(a) \\ \texttt{base}(\&x) & = & x & & \texttt{base}(\&(\star a)) & = & \texttt{base}(a) \\ & & & & \texttt{base}(a \texttt{ ++ } e) & = & \texttt{base}(a). \end{array}$$

Second, in case of an assignment $lv = e$; where e is a pointer, if the left-value lv is under monitoring, the assigned pointer e is generated. The condition "lv is under monitoring" is formally expressed by the proposition $\exists x \in live_{out}(l),\ \&x \mapsto^\star_{\mathcal{A}} lv$. Indeed lv is monitored if and only if it is reachable from one of variables in $live_{out}(l)$. Since reachability cannot be computed statically, our analysis uses as a parameter a points-to analysis $\mathcal{A}$ computing which values of the program may point to which others. We use the result of $\mathcal{A}$ in the form of the relation $\mapsto^\star_{\mathcal{A}}$: $\&x \mapsto^\star_{\mathcal{A}} lv$ means that there is a path from $\&x$ to lv in the points-to graph.

4.2 Correctness

Since our analysis aims at determining which blocks must be monitored by E-ACSL, being correct means that instrumenting only blocks resulting from the analysis must not alter the evaluation of assertions in the output program. Therefore, an informal way to state the correctness could be: "at any program point, altering the observation memory outside of the domain defined by the analysis does not alter the program's behavior".

In order to further formalize this idea, we need to define notions for comparing two execution traces of a program. In particular, we want to be able to compare two observation memories, and state that they are the same on a certain set of blocks. Let $\overline{M}$ and $\overline{M'}$ be observation memories, and $\mathcal{B}$ be a set of blocks. We say that $\overline{M}$ and $\overline{M'}$ are equivalent on $\mathcal{B}$, and we note $\overline{M} \sim_{\mathcal{B}} \overline{M'}$ if they "have the same values on the blocks in $\mathcal{B}$", that is:

$$\forall b \in \mathcal{B}, \begin{cases} \forall \delta \in \mathbb{Z}, & \overline{M} \vDash (b, \delta) & \Leftrightarrow \overline{M'} \vDash (b, \delta) \\ & \texttt{bounds}(\overline{M}, b) & = \texttt{bounds}(\overline{M'}, b) \\ \forall \delta \in \mathbb{Z}, & \texttt{read\_init}(\overline{M}, b, \delta) & = \texttt{read\_init}(\overline{M'}, b, \delta) \end{cases}$$

Following Nielson et al. [19] the theorem is composed of two statements: one describes the case of an evaluation step that terminates the program, and the other one the case where it continues thereafter.

Immediate Termination.

Theorem 4.1. *Let $\langle s, M_1, \overline{M_1} \rangle \to (M_2, \overline{M_2})$ be an execution step, and let $\overline{M'_1}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M'_1}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s))} \mathcal{R}_{M_1}(\&x)$.*

Then there exists $\overline{M'_2}$ such that $\langle s, M_1, \overline{M'_1} \rangle \to (M_2, \overline{M'_2})$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M'_2}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{out}(\mathcal{I}(s))} \mathcal{R}_{M_2}(\&x)$.

Continued Evaluation.

Theorem 4.2. *Let $\langle s_1, M_1, \overline{M_1} \rangle \to \langle s_2, M_2, \overline{M_2} \rangle$ be an execution step, and let $\overline{M'_1}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M'_1}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s_1))} \mathcal{R}_{M_1}(\&x)$.*

Then there exists $\overline{M'_2}$ such that $\langle s_1, M_1, \overline{M'_1} \rangle \to \langle s_2, M_2, \overline{M'_2} \rangle$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M'_2}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{in}(\mathcal{I}(s_2))} \mathcal{R}_{M_2}(\&x)$.

The following diagram gives a visual intuition of the correctness property, here in the case of immediate termination:

$$\begin{array}{ccc} \langle s, M_1, \overline{M_1} \rangle & \to & (M_2, \overline{M_2}) \\ \sim_{\mathcal{B}_1} & & \sim_{\mathcal{B}_2} \\ \langle s, M_1, \overline{M'_1} \rangle & \to & (M_2, \overline{M'_2}) \end{array}$$

$$
\begin{array}{rcl}
live_{out}(l) & \supseteq & \begin{cases} \emptyset & \text{if } l \in \mathcal{F}(s) \\ \bigcup \{live_{in}(l') \mid (l,l') \in flow(s)\} & \text{otherwise} \end{cases} \\
live_{in}(l) & \supseteq & live_{out}(l) \cup gen(l)
\end{array}
$$

$$
gen\left([lv = e;]^l\right) = \begin{cases} \{\texttt{base}(e)\} & \text{if } lv \text{ is a pointer, and } \exists x \in live_{out}(l),\ \&x \mapsto^{\star}_{\mathcal{A}} lv \\ \emptyset & \text{otherwise} \end{cases}
$$

$$
gen\left([\texttt{skip};]^l\right) = \emptyset \qquad gen\left([e]^l\right) = \emptyset \qquad gen\left([p]^l\right) = \rho(p)
$$

$$
gen\left([lv = \texttt{alloc}(e);]^l\right) = \emptyset \qquad gen\left([\texttt{free}(l);]^l\right) = \emptyset
$$

$$
\begin{array}{rclcrcll}
\rho(\backslash\texttt{valid}(a)) & = & \{\texttt{base}(a)\} & & \rho(p_1 \odot p_2) & = & \rho(p_1) \cup \rho(p_2) & \odot \in \{\wedge, \vee\} \\
\rho(\backslash\texttt{init}(a)) & = & \{\texttt{base}(a)\} & & \rho(t_1 \diamond t_2) & = & \theta(t_1) \cup \theta(t_2) & \diamond \in \{\equiv, \leq\} \\
\rho(\neg p) & = & \rho(p) & & & & &
\end{array}
$$

$$
\begin{array}{rclcrcl}
\theta(e) & = & \emptyset & & \theta(\backslash\texttt{base\_addr}(a)) & = & \{\texttt{base}(a)\} \\
\theta(\backslash\texttt{offset}(a)) & = & \{\texttt{base}(a)\} & & \theta(\backslash\texttt{block\_length}(a)) & = & \{\texttt{base}(a)\}
\end{array}
$$

Figure 9: Dataflow analysis definition.

5 PROOF OUTLINE

The proof is by induction on the structure of the evaluation derivation. The most difficult cases are statements that perform memory operations, namely assignment, allocation and deallocation. The other cases are straightforward, the proof relying mainly on the properties of least fixed point of ($live_{in}$, $live_{out}$).

We use two kinds of auxiliary lemmas. Some describe properties of the equivalence relation $\sim_{\mathcal{B}}$ such as the fact that it is an equivalence relation, how to combine equivalences between the same two memories on multiple domains, or conversely how two memories equivalent on a given domain remain equivalent when the same memory operation is performed on both. Others describe the evolution (or invariance) of reachable blocks sets when memory operations are performed.

5.1 Lemmas.

Lemma 5.1 (Domain Restriction). *If $\overline{M} \sim_X \overline{M'}$ and $Y \subseteq X$ then $\overline{M} \sim_Y \overline{M'}$.*

Lemma 5.2 (Domain Union). *If $\overline{M} \sim_X \overline{M'}$ and $\overline{M} \sim_Y \overline{M'}$ then $\overline{M} \sim_{X \cup Y} \overline{M'}$.*

Lemma 5.3 (Equivalence Preservation (initialization)). *For a given domain $\mathcal{B}$, two equivalent memories remain equivalent when the same* `initialize` *operation is performed on both.*

Lemma 5.4 (Reachability from base address). *Any block reachable from a given memory location a is also reachable from its base address:*

$$\forall a \in \textbf{Memval},\ \forall M,\ \mathcal{R}_M(a) \subseteq \mathcal{R}_M(\&\texttt{base}(a)).$$

Lemma 5.5 (Writing outside of a reachable set does not modify it). *Let x be a variable. For a given memory state M_1, if $\texttt{store}(M_1, b, \delta, v) = \texttt{Some}(M_2)$ with $b \notin \mathcal{R}_{M_1}(\&x)$, then $\mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x)$.*

Lemma 5.6 (Writing an integer does not modify reachable set). *Suppose that M_1 and M_2 are two memory states in the trace of a well-typed program, such that $\texttt{store}(M_1, b, \delta, \texttt{Int}(n)) = \texttt{Some}(M_2)$. Then for any variable x, the set of reachable blocs from x is the same in M_1 and M_2:*

$$\forall x \in E,\ \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x).$$

Lemma 5.7 (Maximum extension of a reachable set). *If M_1 and M_2 are two memory states such that $\texttt{store}(M_1, b, \delta, \texttt{Ptr}(b_v, \delta_v)) = \texttt{Some}(M_2)$, then the following inclusion is verified: $\mathcal{R}_{M_2}(a) \subseteq \mathcal{R}_{M_1}(a) \cup \{b' \mid b_v \mapsto^{\star}_{M_1} b'\}$.*

5.2 Case of Assignments.

We detail the main case of the induction: that of assignments. Given an execution step $\langle [lv = e;]^l, M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})$ and an observation memory $\overline{M_1'}$ such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$, we want to show the existence of $\overline{M_2'}$ such that:

- $\langle [lv = e;]^l, M_1, \overline{M_1'}\rangle \rightarrow (M_2, \overline{M_2'})$ is a valid execution step
- $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$.

$\mathcal{B}_1$ and $\mathcal{B}_2$ are defined as:

$$
\begin{array}{rcl}
\mathcal{B}_1 & = & \bigcup_{x \in live_{in}(l)} \mathcal{R}_{M_1}(\&x) \\
\mathcal{B}_2 & = & \bigcup_{x \in live_{out}(l)} \mathcal{R}_{M_2}(\&x).
\end{array}
$$

The execution step has the following form:

$$
\frac{\begin{array}{ll} M_1 \vDash_{\text{e}} e \Rightarrow v & \texttt{store}(M_1, b_l, \delta_l, v) = \texttt{Some}(M_2) \\ M_1 \vDash_{\text{lv}} lv \Leftarrow (b_l, \delta_l) & \texttt{initialize}(\overline{M_1}, b_l, \delta_l) = \overline{M_2} \end{array}}{\langle [lv = e;]^l, M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}
$$

We define $\overline{M_2'} = \texttt{initialize}(\overline{M_1'}, b_l, \delta_l)$, and prove that $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$. We distinguish between the case where the block written to is monitored ($b_l \in \mathcal{B}_2$) and the case where it is not.

In both cases we make use of the following relation:

$$\overline{M_2} \sim_{\{b_l\}^{\complement}} \overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'} \sim_{\{b_l\}^{\complement}} \overline{M_2'} \tag{1}$$

in which the first and last equivalences are derived from Lemma 5.3 and the middle one is an hypothesis of our theorem.

5.2.1 case $b_l \notin \mathcal{B}_2$. Let us prove that $\mathcal{B}_2 \subseteq \mathcal{B}_1$. Since $\forall\, l,\ live_{in}(l) \supseteq live_{out}(l)$, it suffices to show that $\forall x \in live_{out}(l), \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x)$. Since for any of these sets we have $b_l \notin \mathcal{R}_{M_2}(\&x)$ we can apply Lemma 5.5 to conclude.

We can now rewrite the assumption $b_l \notin \mathcal{B}_2$ as $\mathcal{B}_2 \subseteq \{b_l\}^{\complement}$ so that all domains involved in equation 1 are supersets of $\mathcal{B}_2$. The conclusion follows from Lemma 5.1.

5.2.2 case $b_l \in \mathcal{B}_2$. Using Lemma 5.2, we consider $\mathcal{B}_2$ as the (disjoint) union of $\mathcal{B}_2 \backslash \{b_l\}$ and $\{b_l\}$, and we show the equivalence on each of these subdomains. Here we have to consider the type of the assigned expression.

subcase `int`. If the expression has the type `int`, we can use Lemma 5.6 to prove that $\mathcal{B}_2 \subseteq \mathcal{B}_1$, using the same method as previously. Applying Lemma 5.1 to 3 then yields $\overline{M_2} \sim_{\mathcal{B}_2 \backslash \{b_l\}} \overline{M_2'}$. Besides, since $b_l \in \mathcal{B}_2 \subseteq \mathcal{B}_1$ and $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$ we have more specifically $\overline{M_1} \sim_{\{b_l\}} \overline{M_1'}$. By Lemma 5.3 we can conclude that $\overline{M_2} \sim_{\{b_l\}} \overline{M_2'}$.

subcase $\tau\star$. If the expression is a pointer $v = \mathtt{Ptr}(b_v, \delta_v)$ for some (b_v, δ_v), we can use Lemma 5.7 to approximate the evolution of reachable blocks sets: $\mathcal{R}_{M_2}(a) \subseteq \mathcal{R}_{M_1}(a) \cup \mathcal{R}_{M_1}(e)$. We consider the union of these inclusions:

$$\mathcal{B}_2 \subseteq \bigcup_{x \in live_{out}(l)} \mathcal{R}_{M_1}(\&x) \cup \mathcal{R}_{M_1}(e). \tag{2}$$

Let us prove that terms of the right side are both subsets of $\mathcal{B}_1$. For the first one, using the definition of $\mathcal{B}_1$ and the inclusion $live_{out}(l) \subseteq live_{in}(l)$ we can write

$$\bigcup_{x \in live_{out}(l)} \mathcal{R}_{M_1}(\&x) \quad \subseteq \quad \bigcup_{x \in live_{in}(l)} \mathcal{R}_{M_1}(\&x) = \mathcal{B}_1.$$

For the second one, we use Lemma 5.4 in conjunction with the fact that $\mathsf{base}(e) \in live_{in}(l)$. This corresponds to the first case in the definition of the generation function $gen([lv = e;])$, which is defined by the condition: $\exists x \in live_{out}(l)$, $\&x \mapsto^{\star}_{\mathcal{A}} lv$. This condition is necessarily verified here: since $b_l \in \mathcal{B}_2$, by definition of $\mathcal{B}_2$ there is some $x \in live_{out}(l)$ such that $b_l \in \mathcal{R}_{M_2}(\&x)$. Let b_x be the block allocated for x. By definition of $\mathcal{R}_{M_2}()$, $b_x \mapsto^{\star}_{M_2} b_l$. Finally, applying the correctness property of the points-to analysis to this relation yields the expected result: $x \mapsto^{\star}_{\mathcal{A}} lv$ with $x \in live_{out}(l)$.

Now we can deduce from equation 2 that $\mathcal{B}_2 \subseteq \mathcal{B}_1$, and conclude with the same arguments as for the subcase `int`.

6 RELATED WORK

The E-ACSL plugin aims at performing runtime assertion checking for C code, specified in a rich specification language E-ACSL. Being a part of the Frama-C framework, it can be combined with other analyses for quite unique usages. For example, runtime assertion checking can be used to assist the user in case of proof failures during deductive verification [21].

E-ACSL can also be used as a dynamic bug finder, see for example [25, 29] for a recent comparison of such sanitizers. These tools implement various techniques, in particular concerning the storage of metadata necessary for the analysis. There are roughly three main techniques.

A shadow memory is in linear correspondence with the program memory. AddressSanitizer [22] stores 1 shadow byte per aligned 8-byte sequences in the application. Shadow memory does not incur a large runtime overhead but its adaptation can be quite tricky when richer metadata is necessary, as it is the case for E-ACSL [28].

Another technique used to attach additional metadata to each pointer is the so-called fat pointers technique [1, 20]. There is significant overhead in this case, and instrumented code cannot directly be linked with non instrumented code because of differences in memory layout.

Finally, metadata can be stored in a separate data structure [17]. In a previous version of E-ACSL, metadata was stored in a Patricia trie [15]. This technique allows the instrumented code to be linked with non instrumented code and to store richer metadata but execution time overhead may be significant.

Mehlich [17] mentions that "the current implementation of CheckPointer could be improved by using static analysis". Jakobsson et al. [11] improved E-ACSL by a static analysis (that we formalize in this paper) in order to avoid monitoring memory locations that do not need to be monitored. They combine shadow memory (used for memory locations that only require validity or initialization checks) with a Patricia trie (for locations requiring more complex checks). Other sanitizers rely on static analysis to improve the performances of runtime checking [18, 24, 30]. The work that is the closest to ours in intent and using a proved analysis is CCured [18], but the techniques are different. CCured is based on fat pointers, their static analysis is a type system, and only memory safety is checked. The correctness of the analysis is ensured by a theorem stating that the execution of an instrumented program can either terminate correctly or stop because the instrumentation has detected an error, ruling out an incorrect termination because of an invalid memory access. The proof assumes that there is no dynamic memory allocation in the program.

Our ultimate goal is to be able to extract a correct E-ACSL-like plugin from a Coq development. This goal necessitates an implementation and verification in Coq of the presented static analysis extended to CompCert C. Our formalization follows the CompCert memory model [16]. There are several works on the verification of static analyses with a proof assistant. We only briefly present some papers related to the Coq proof assistant. The CompCert compiler itself contains static analysis for the sake of optimization [2], including a generic implementation of the Kildall algorithm [13]. This is however related to the RTL language of CompCert. Like Verasco [12] the presented memory analysis will rather be conducted at the level of the Cminor language. Unlike Verasco, we plan to directly implement the analysis as a dataflow analysis rather than as an abstract interpretation. [9] is a generic formalization of a solver that can be used for static analysis. However the algorithms are formalized as relations and cannot be extracted to executable code. [4] is an extractable dataflow analysis in Coq in the context of Java.

7 CONCLUSION AND PERSPECTIVES

We have formalized the dataflow analysis underlying a major optimization of the E-ACSL tool for runtime assertion checking and memory debugging. This optimization was shown to bring 60% to 73% performance savings [11]. The analysis is proved correct with regard to an operational semantics which reflects the use of a second memory store to monitor the program memory state.

We believe that making the memory monitoring explicit in the language semantics (via the so-called observation memory) significantly eases formal reasoning about optimizations performed at this level, and can be beneficial in other contexts. We plan to further explore this notion in our ongoing effort towards a mechanized formalization of E-ACSL. This future work can be carried out in three main directions.

First, the core language presented here can be extended to be more representative of the real E-ACSL language, making the formalization *wider*. Features such as structures and function calls strongly influence the memory layout of program memory space, so this would most probably need some adaptation.

Second, this work is still mostly pen-and-paper. We could make the formalization *deeper* by porting it to the Coq proof assistant.

Finally, in this work we have *assumed* that E-ACSL gives the described semantics to an annotated program by translating annotations into C code, but this translation is not formalized yet. Since this translation is the core E-ACSL, verifying it has the highest priority in the global formalization effort of E-ACSL.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments and feedbacks. This work was partly supported by project VESSEDIA, which has received funding from the European Union's Horizon 2020 Research and Innovation Program under grant agreement No 731453. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defence.

REFERENCES

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Programming Languages Design and Implementation (PLDI)*. ACM, 1994.
[2] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs (TYPES)*. Springer, 2006.
[3] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43, 2009.
[4] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342, 2005.
[5] Steve Christey. 2011 CWE/SANS top 25 most dangerous software errors. Technical Report 1.0.3, The MITRE Corporation, http://www.mitre.org, 2011.
[6] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *Software Engineering Notes*, 31, 2006.
[7] Loïc Correnson and Julien Signoles. Combining analyses for C program verification. In *Formal Methods for Industrial Case Studies (FMICS)*. Springer, 2012.
[8] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *Applied Computing (SAC)*. ACM, 2013.
[9] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In *Static Analysis (SAS)*. Springer, 2010.
[10] *Programming languages – C*. ISO/IEC 9899:1999, 1999.
[11] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Fast as a shadow, expressive as a tree: optimized memory monitoring for C. *Science of Computer Programming*, 132, 2016.
[12] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Principles of Programming Languages (POPL)*. ACM, 2015.
[13] Gary A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*. ACM, 1973.
[14] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27, 2015.
[15] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *Runtime Verification (RV)*. Springer, 2013.
[16] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In Andrew W. Appel, editor, *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
[17] Michael Mehlich. CheckPointer - a C memory access validator. In *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011.
[18] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *Programming Languages and Systems (TOPLAS)*, 27, 2005.
[19] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, 1999.
[20] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Programming Language Design and Implementation (PLDI)*. ACM, 2009.
[21] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. Your proof fails? Testing helps to find the reason. In *Tests and Proofs (TAP)*. Springer, 2016.
[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX Annual Technical Conference (USENIX)*. USENIX Association, 2012.
[23] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a runtime verification tool for safety and security of C programs. tool paper. In *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. EasyChair, 2017.
[24] Matthew S. Simpson and Rajeev Barua. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience*, 43, 2013.
[25] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. In *Security and Privacy (S&P)*. IEEE, 2019. to appear.
[26] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: a study of field failures in operating systems. In *Fault Tolerant Computing (FTCS)*. IEEE, 1991.
[27] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2012.
[28] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow state encoding for efficient monitoring of block-level properties. In *Memory Management (ISMM)*. ACM, 2017.
[29] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. Detection of security vulnerabilities in C code using runtime verification: an experience report. In *Tests and Proofs (TAP)*. Springer, 2018.
[30] Jun Yuan and Rob Johnson. CAWDOR: compiler assisted worm defense. In *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2012.

A APPENDIX

This appendix contains a detailed soundness proof of our dataflow analysis. Section B recalls the formal statement of the theorem and presents the global structure of the proof. Section C lists the lemmas used along the proof; they are stated informally, and only the main idea of their proof is given. Please note that they numbering differs from the one used in the article body. The remaining sections detail all the cases of the induction:

- memory operations, which represent the main difficulty of this proof, are tackled in Section D
- Section E includes the others cases of execution in which the program immediately terminates
- Section F handles control-flow related statements.

B THEOREM STATEMENT AND PROOF STRUCTURE

Immediate Termination. Let $\langle s, M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})$ be an execution step, and let $\overline{M_1'}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s))} \mathcal{R}_{M_1}(\&x)$.

Then there exists $\overline{M_2'}$ such that $\langle s, M_1, \overline{M_1'}\rangle \to (M_2, \overline{M_2'})$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{out}(\mathcal{I}(s))} \mathcal{R}_{M_2}(\&x)$.

Continued Evaluation. Let $\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle$ be an execution step, and let $\overline{M_1'}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s_1))} \mathcal{R}_{M_1}(\&x)$.

Then there exists $\overline{M_2'}$ such that $\langle s_1, M_1, \overline{M_1'}\rangle \to \langle s_2, M_2, \overline{M_2'}\rangle$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{in}(\mathcal{I}(s_2))} \mathcal{R}_{M_2}(\&x)$.

B.1 Structure of the Proof

The proof is by induction on the structure of the evaluation derivation. The most difficult cases are statements that perform memory operations, namely assignment, allocation and deallocation. The other cases are straightforward.

C INTERMEDIATE LEMMAS

C.1 Properties of $\sim_{\mathcal{B}}$

Lemma C.1 (Equivalence Relation). *$\forall X$, $\sim_X$ is an equivalence relation on observation memories.*

Lemma C.2 (Domain Restriction). *If $\overline{M_1} \sim_X \overline{M_2}$ and $Y \subseteq X$ then $\overline{M_1} \sim_Y \overline{M_2}$*

Lemma C.3 (Domain Union). *If $\overline{M_1} \sim_X \overline{M_2}$ and $\overline{M_1} \sim_Y \overline{M_2}$ then $\overline{M_1} \sim_{X\cup Y} \overline{M_2}$.*

The above properties are a direct consequence of the definition of $\sim_{\mathcal{B}}$, whereas the following lemmas use the axiomatic properties of observation memory.

Lemma C.4 (Equivalence Preservation (initialization)). *For a given domain $\mathcal{B}$, two equivalent memories remain equivalent when the same* `initialize` *operation is performed on both.*

Lemma C.5 (Equivalence Preservation (block storage)). *For a given domain $\mathcal{B}$, two equivalent memories remain equivalent when the same* `store_block` *operation is performed on both.*

Lemma C.6 (Equivalence Preservation (block deletion)). *For a given domain $\mathcal{B}$, two equivalent memories remain equivalent when the same* `delete_block` *operation is performed on both.*

C.2 Properties of `base()`

Variable Address. The base address of any memory location is the address of a given variable: $\forall a, \exists x, \mathtt{base}(a) = \&x$.

Consequently, if we consider a trace of a given program, a base address always evaluates to the same block ; that is, the block pointed to by the base address does not depend on the point of the trace where the evaluation takes place.

Lemma C.7 (Reachability from base address). *Any block reachable from a given memory location a is also reachable from its base address:*

$$\forall a \in \mathbf{Memval},\ \forall M,\ \mathcal{R}_M(a) \subseteq \mathcal{R}_M(\&\mathtt{base}(a)).$$

Proof: by induction on the structure of a.

C.3 Evolution of $\mathcal{R}_M(\&x)$ when writing in M

The following lemmas describe the behavior of reachable blocks sets under memory updates. They are best understood by viewing the memory as a directed graph with blocks as vertices and pointers defining edges. From this point of view, the fact that successors of a block are always in the same reachable set as this block itself makes the proof of these lemmas straightforward.

Lemma C.8 (Writing outside of a reachable set does not modify it). *Let x be a variable. For a given memory state M_1, if $\mathtt{store}(M_1, b, \delta, v) = \mathtt{Some}(M_2)$ with $b \notin \mathcal{R}_{M_1}(\&x)$, then*

$$\mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x).$$

Lemma C.9 (Writing an integer does not modify reachable set). *Suppose that M_1 and M_2 are two memory states in the trace of a well-typed program, such that $\mathtt{store}(M_1, b, \delta, \mathtt{Int}(n)) = \mathtt{Some}(M_2)$. Then for any variable x, the set of reachable blocs from x is the same in M_1 and M_2:*

$$\forall x \in E,\ \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x).$$

Lemma C.10 (Maximum extension of a reachable set). *If M_1 and M_2 are two memory states such that*

$$\mathtt{store}(M_1, b, \delta, \mathtt{Ptr}(b_v, \delta_v)) = \mathtt{Some}(M_2),$$

then the following inclusion is verified:

$$\mathcal{R}_{M_2}(a) \subseteq \mathcal{R}_{M_1}(a) \cup \{b' \mid b_v \mapsto^{\star}_{M_1} b'\}$$

D MEMORY OPERATIONS

Let $\langle s, M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})$ be an execution step, and let $\overline{M_1'}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s))} \mathcal{R}_{M_1}(\&x)$.

We want to show that there exists $\overline{M_2'}$ such that $\langle s, M_1, \overline{M_1'}\rangle \to (M_2, \overline{M_2'})$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{out}(\mathcal{I}(l))} \mathcal{R}_{M_2}(\&x)$.

D.1 Assignment

The execution step has the form:

Eval-assign

$$\frac{\begin{array}{ll} & \Gamma \vdash_e e : \tau \\ M_1 \vDash_{\mathrm{e}} e \Rightarrow v & \texttt{store}(M_1, b_l, \delta_l, v) = \texttt{Some}(M_2) \\ M_1 \vDash_{\mathrm{lv}} lv \Leftarrow (b_l, \delta_l) & \texttt{initialize}(\overline{M_1}, b_l, \delta_l) = \overline{M_2} \end{array}}{\langle [lv = e;]^l, M_1, \overline{M_1} \rangle \rightarrow (M_2, \overline{M_2})}$$

We define $\overline{M_2'} = \texttt{initialize}(\overline{M_1'}, b_l, \delta_l)$, and prove that $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$. We distinguish between the case where the block written to is monitored ($b_l \in \mathcal{B}_2$) and the case where it is not.

In each of these cases we make use of the following assumption

$$\overline{M_2} \sim_{\{b_l\}^{\complement}} \overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'} \sim_{\{b_l\}^{\complement}} \overline{M_2'} \tag{3}$$

in which the first and last equivalences are derived from Lemma C.4 and the middle one is an hypothesis of our theorem.

D.1.1 case $b_l \notin \mathcal{B}_2$. Let us prove that $\mathcal{B}_2 \subseteq \mathcal{B}_1$. Since $\forall\, l$, $live_{in}(l) \supseteq live_{out}(l)$, it suffices to show that $\forall x \in live_{out}(l), \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x)$. Since for any of these sets we have $b_l \notin \mathcal{R}_{M_2}(\&x)$ we can apply Lemma C.8 to conclude.

We can now rewrite the assumption $b_l \notin \mathcal{B}_2$ as $\mathcal{B}_2 \subseteq \{b_l\}^{\complement}$ so that the all domains involved in 3 are supersets of $\mathcal{B}_2$. The conclusion follows from Lemma C.2.

D.1.2 case $b_l \in \mathcal{B}_2$. Using Lemma C.3, we consider $\mathcal{B}_2$ as the (disjoint) union of $\mathcal{B}_2 \backslash \{b_l\}$ and $\{b_l\}$, and we show the equivalence on each of these subdomains. Here we have to consider the type of the assigned expression.

subcase `int`. If the expression has the type `int`, we can use Lemma C.9 to prove that $\mathcal{B}_2 \subseteq \mathcal{B}_1$, using the same method as previously. Applying Lemma C.2 to 3 then yields $\overline{M_2} \sim_{\mathcal{B}_2 \backslash \{b_l\}} \overline{M_2'}$. Besides, since $b_l \in \mathcal{B}_2 \subseteq \mathcal{B}_1$ and $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$ we have more specifically $\overline{M_1} \sim_{\{b_l\}} \overline{M_1'}$. By Lemma C.4 we can conclude that $\overline{M_2} \sim_{\{b_l\}} \overline{M_2'}$.

subcase $\tau\star$. If the expression is a pointer $v = \texttt{Ptr}(b_v, \delta_v)$ for some (b_v, δ_v), we can use Lemma C.10 to approximate the evolution of reachable blocks sets: $\mathcal{R}_{M_2}(a) \subseteq \mathcal{R}_{M_1}(a) \cup \mathcal{R}_{M_1}(e)$. We consider the union of these inclusions:

$$\mathcal{B}_2 \subseteq \bigcup_{x \in live_{out}(l)} \mathcal{R}_{M_1}(\&x) \cup \mathcal{R}_{M_1}(e). \tag{4}$$

Let us prove that terms of the right side are both subsets of $\mathcal{B}_1$.

For the first one, using the definition of $\mathcal{B}_1$ and the inclusion $live_{out}(l) \subseteq live_{in}(l)$ we can write

$$\bigcup_{x \in live_{out}(l)} \mathcal{R}_{M_1}(\&x) \quad \subseteq \quad \bigcup_{x \in live_{in}(l)} \mathcal{R}_{M_1}(\&x) = \mathcal{B}_1.$$

For the second one, we use Lemma C.7 in conjunction with the fact that $\mathsf{base}(e) \in live_{in}(l)$. This corresponds to the first case in the definition of the generation function $gen([lv = e;])$, which is defined by the condition: $\exists x \in live_{out}(l)$, $\&x \mapsto^{\star}_{\mathcal{A}} lv$. This condition is necessarily verified here: since $b_l \in \mathcal{B}_2$, by definition of $\mathcal{B}_2$ there is some $x \in live_{out}(l)$ such that $b_l \in \mathcal{R}_{M_2}(\&x)$. Let b_x be the block allocated for x. By definition of $\mathcal{R}_{M_2}()$, $b_x \mapsto^{\star}_{M_2} b_l$. Finally, applying the correctness property of the points-to analysis to this relation yields the expected result: $x \mapsto^{\star}_{\mathcal{A}} lv$ with $x \in live_{out}(l)$.

Now we can deduce from 4 that $\mathcal{B}_2 \subseteq \mathcal{B}_1$, and conclude with the same arguments as for the subcase `int`.

D.2 Allocation

Eval-malloc

$$\frac{\begin{array}{ll} & \texttt{store\_block}(\overline{M_1}, b, lo, hi) = \overline{M_3} \\ hi - lo = n & \texttt{initialize}(\overline{M_3}, b_l, \delta_l) = \overline{M_2} \\ M \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(n) & \texttt{alloc}(M_1, lo, hi) = (b, M_3) \\ M \vDash_{\mathrm{lv}} lv \Leftarrow (b_l, \delta_l) & \texttt{store}(M_3, b_l, \delta, \texttt{Ptr}(b, 0)) = \texttt{Some}(M_2) \end{array}}{\langle [lv = \texttt{malloc}(e);]^l, M_1, \overline{M_1} \rangle \rightarrow (M_2, \overline{M_2})}$$

Although the allocation is syntactically an atomic statement, it is actually the composition of two memory operations: the allocation in the true sense of the word, followed by the assignment of a pointer to the newly allocated block. It is therefore natural to introduce an intermediate state in which the new block is allocated, but the left value is not pointing to it yet.

Accordingly, we pose the following definitions:

$$\begin{array}{rcl} \overline{M_3'} & = & \texttt{store\_block}(\overline{M_1'}, b, lo, hi) \\ \overline{M_2'} & = & \texttt{initialize}(\overline{M_3}, b_l, \delta_l) \\ \mathcal{B}_3 & = & \bigcup_{x \in live_{in}(l)} \mathcal{R}_{M_3}(\&x) \end{array}$$

We proceed in two steps:

(1) prove $\overline{M_3} \sim_{\mathcal{B}_3} \overline{M_3'}$
(2) use the previous equivalence to prove $\overline{M_2'} \sim_{\mathcal{B}_2} \overline{M_2}$.

D.2.1 Equivalence on intermediate state. Since M_3 is simply M_1 with a new block and no pointer to this block, reachable sets are the same in both memories. Consequently,

$$\forall x \in live_{in}(l),\ \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_3}(\&x)$$

and thus $\mathcal{B}_3 = \mathcal{B}_1$.

Since $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$ and the same allocation is performed on both memories, by Lemma C.5 the equivalence is preserved: $\overline{M_3} \sim_{\mathcal{B}_1} \overline{M_3'}$. $\mathcal{B}_1$ and $\mathcal{B}_3$ being equal, the conclusion follows.

D.2.2 Equivalence on final state. This setting is very similar to the case of a pointer assignment (which is only natural since a pointer assignment is indeed performed after the allocation of a new block).

We want to prove $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$, assuming the equivalence on the intermediate state: $\overline{M_3} \sim_{\mathcal{B}_3} \overline{M_3'}$. b being a freshly allocated block in both $\overline{M_3}$ and $\overline{M_3'}$, we have $\overline{M_3} \sim_{\{b\}} \overline{M_3'}$, which means that we can extend the equivalence domain by using Lemma C.3:

$$\overline{M_3} \sim_{\mathcal{B}_3 \cup \{b\}} \overline{M_3'}.$$

case $b_l \notin \mathcal{B}_3$. Then by Lemma C.8, $\forall x \in live_{in}(l)$, $\mathcal{R}_{M_3}(\&x)$ is unaffected by the write at b_l, so $\mathcal{B}_3 = \mathcal{B}_2$. We can thus use Lemma C.4 to transfer the equivalence to the next program state: $\overline{M_2} \sim_{\mathcal{B}_3 \cup \{b\}} \overline{M_2'}$. To conclude, we use the domain equality $\mathcal{B}_3 = \mathcal{B}_2$ and restrict the domain back to $\mathcal{B}_2$ by Lemma C.2.

case $b_l \in \mathcal{B}_3$. By Lemma C.10, $\forall x \in live_{in}(l)$, $\mathcal{R}_{M_2}(\&x) \subseteq \mathcal{R}_{M_3}(\&x) \cup \{b\}$. Considering the union on all $x \in live_{in}(l)$ we get: $\mathcal{B}_2 \subseteq \mathcal{B}_3 \cup \{b\}$. Conclusion follows by Lemma C.4.

D.3 Deallocation

$$\textsc{Eval-free}\ \frac{M_1 \vDash_{\mathrm{e}} a \Rightarrow (b, 0) \qquad \texttt{free}(M_1, b) = \texttt{Some}(M_2) \qquad \texttt{delete\_block}(\overline{M_1}, b) = \overline{M_2}}{\langle [\texttt{free}(a);]^l, M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})}$$

M_2 is obtained by deleting one block from M_1 so the connectivity between blocks in M_2 is necessarily lower than in M_1. More formally, for any variable x, $\mathcal{R}_{M_2}(\&x) \subseteq \mathcal{R}_{M_1}(\&x)$. Since $live_{in}(l) \supseteq live_{out}(l)$ by definition of the transfer function, this implies $\mathcal{B}_1 \supseteq \mathcal{B}_2$. Therefore, by domain restriction (Lemma C.2), $\overline{M_1} \sim_{\mathcal{B}_2} \overline{M_1'}$. Conclusion follows from Lemma C.6.

E OTHER CASES OF IMMEDIATE TERMINATION

E.1 Skip

$$\textsc{eval-skip}\ \langle \texttt{skip}, M_1, \overline{M_1}\rangle \to (M_1, \overline{M_1})$$

The memory state and the equivalence domain are the same before and after the statement execution, so the equivalence is trivially preserved.

E.2 End of a loop

$$\textsc{Eval-whileEnd}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(0)}{\langle \texttt{while}\ ([e]^l)\ s, M, \overline{M}\rangle \to (M, \overline{M})}$$

By definition of the functions $(live_{in}, live_{out})$, $live_{in}(l) \supseteq live_{out}(l)$. Since the memory is unchanged, this implies $\mathcal{B}_1 \supseteq \mathcal{B}_2$. Conclude by Lemma C.2.

E.3 Assertion

$$\textsc{Eval-assert}\ \frac{M \vDash_{\mathrm{p}} p \Rightarrow \top}{\langle /*@\ \texttt{assert}\ [p]^l;\ */\ , M, \overline{M}\rangle \to (M, \overline{M})}$$

$live_{in}(l) \supseteq live_{out}(l) = live_{in}(\mathcal{I}(s))$ by definition of the analysis. Since the memory is unchanged, this implies $\mathcal{B}_1 \supseteq \mathcal{B}_2$ and we conclude with Lemma C.2.

F CASES OF CONTINUED EXECUTION

Let $\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle$ be an execution step, and let $\overline{M_1'}$ be an observation memory such that $\overline{M_1} \sim_{\mathcal{B}_1} \overline{M_1'}$, with $\mathcal{B}_1 = \bigcup_{x \in live_{in}(\mathcal{I}(s_1))} \mathcal{R}_{M_1}(\&x)$.

We want to show that there exists $\overline{M_2'}$ such that $\langle s_1, M_1, \overline{M_1'}\rangle \to \langle s_2, M_2, \overline{M_2'}\rangle$ and $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$, with $\mathcal{B}_2 = \bigcup_{x \in live_{in}(\mathcal{I}(s_2))} \mathcal{R}_{M_2}(\&x)$.

The case addressed in this section are that of execution steps which do not terminate immediately; these cases correspond to the statements defining the program's control flow. Since these statements do not modify any of the memory stores, the reachable blocks sets are the same before and after the statement execution.

Therefore, in order to prove some domain inclusion $\mathcal{B}_1 \supseteq \mathcal{B}_2$, it suffices to prove the corresponding live variable sets inclusion $live_{in}(\mathcal{I}(s_1)) \supseteq live_{in}(\mathcal{I}(s_2))$. In all the following cases, this inclusion is a direct consequence of the relation between the dataflow analysis result $(live_{in}, live_{out})$ and the CFG structure, or in other words the fact that $(live_{in}, live_{out})$ is the least fixed point of the system defining it.

F.1 Sequence

There are two subcases: either the first statement is a single, atomic statement (assignement, allocation…), or it is a composed statement (loop, sequence, conditional…).

F.1.1 Subcase eval-seqEnd.

$$\textsc{eval-seqEnd}\ \frac{\langle s_1, M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle}$$

$live_{out}(\mathcal{I}(s_1)) \supseteq live_{in}(\mathcal{I}(s_2)$ by definition of the analysis, so $\mathcal{B} \supseteq \mathcal{B}_2$ and we can restrict the equivalence to $\mathcal{B}_2$ by Lemma C.2: $\overline{M_2} \sim_{\mathcal{B}_2} \overline{M_2'}$. We conclude by replacing $\overline{M_2}$ with $\overline{M_2'}$ in the evaluation derivation:

$$\frac{\langle s_1, M_1, \overline{M_1'}\rangle \to (M_2, \overline{M_2'})}{\langle s_1\ s_2, M_1, \overline{M_1'}\rangle \to \langle s_2, M_2, \overline{M_2'}\rangle}$$

F.1.2 Subcase eval-seqCont.

$$\textsc{eval-seqCont}\ \frac{\langle s_1, M_1, \overline{M_1}\rangle \to \langle \hat{s}_1, M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle \hat{s}_1\ s_2, M_2, \overline{M_2}\rangle}$$

The proof is the same as in the previous case, substituting $live_{in}(\mathcal{I}(\hat{s}_1))$ for $live_{out}(\mathcal{I}(s_1))$.

F.2 Conditional Branching

$$\textsc{Eval-ifTrue}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(n) \qquad n \neq 0}{\langle \texttt{if}\ ([e]^l)\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_t, M, \overline{M}\rangle}$$

$$\textsc{Eval-ifFalse}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(0)}{\langle \texttt{if}\ ([e]^l)\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_f, M, \overline{M}\rangle}$$

Here $\overline{M_1}$, $\overline{M_1'}$, $\overline{M_2}$, and $\overline{M_2'}$ are all equal (to $\overline{M}$). Therefore it suffices to show that $\mathcal{B}_1 \supseteq \mathcal{B}_2$, that is:

$$\bigcup_{x \in live_{in}(l)} \mathcal{R}_M(\&x) = \bigcup_{x \in live_{in}(\mathcal{I}(s))} \mathcal{R}_M(\&x)$$

where s is s_t or s_f, depending on the case. This follows directly from:

$$live_{in}(l) \supseteq live_{in}(\mathcal{I}(s_t)) \cup live_{in}(\mathcal{I}(s_f))$$

(definition of the transfer function).

F.3 Loop

$$\textsc{Eval-whileCont}\ \frac{M \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(n) \qquad n \neq 0}{\langle \texttt{while}\ ([e]^l)\ s, M, \overline{M}\rangle \to \langle s\ \texttt{while}\ ([e]^l)\ s, M, \overline{M}\rangle}$$

Since there no memory operation is performed, is suffices to show the inclusion: $live_{in}(l) \supseteq live_{in}(\mathcal{I}(s\ \texttt{while}\ (e)\ s))$. Applying the definition of $\mathcal{I}$ to the right hand side, we have $\mathcal{I}(s\ \texttt{while}\ (e)\ s)) = \mathcal{I}(s)$, so that we are left to prove:

$$live_{in}(l) \supseteq live_{in}(\mathcal{I}(s))$$

which results from the definition of $(live_{in}, live_{out})$.