Soundness of a Dataflow Analysis for Memory Monitoring
Dara Ly, Nikolai Kosmatov, Julien Signoles, Frédéric Loulergue

To cite this version:

HAL Id: cea-02283406
https://cea.hal.science/cea-02283406
Submitted on 10 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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

Nikolai Kosmatov  
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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HILT 2017, November 5-6, 2018, Boston, MA, USA  
© 2018 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6679-3/18/11. . . $15.00  
https://doi.org/10.1145/nnnnnnn.nnnnnn

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
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].

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.

Execution Memory. 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 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 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).

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.

Assumptions 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].

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 $lo$, $hi$, $\delta$, and a value $v$. Statement 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 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
is true if and only if the contents of offset (in bytes) of the block containing address Informal semantics

besides these (is not valid. Finally, returns it. Again, this operation can fail, e.g., if the supplied address denoted either contains a memory state as a return type. A value of type

\( \text{int f(void)} \) {
    \text{int x, y, z, *p;}
    \text{p = &x;}
    \text{x = 0;}
    \text{y = 1;}
    \text{z = 2;}
    /\ast\ast \text{assert}
    \text{\valid(p); */}
    \text{*p = 3;}
    \text{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 \( \text{Some}(M) \), or contains nothing (denoted \( \text{None} \)). Statement \( \text{Load}(M, b, \delta) \) reads the value at address \( (b, \delta) \) in \( M \), if possible, and returns it. Again, this operation can fail, e.g., if the supplied address is not valid. Finally, \( \text{Store}(M, b, \delta, v) \) writes the value \( v \) at address \( (b, \delta) \) 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 \( M \): \( \text{bounds}(M, b) = (\text{lo}, \text{hi}) \)

\( \text{lo} \) and \( \text{hi} \) 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 \( M \models (b, \delta) \) the proposition "\( (b, \delta) \) is a valid address in the memory state \( M \)" meaning that:

1. \( b \) was allocated and not (yet) deallocated in \( M \); and
2. \( \text{lo} \leq \delta < \text{hi} \) where \( \text{lo} \) and \( \text{hi} \) 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 \( \text{lo}, \text{hi}, \delta \). Statement \( \text{store}_{\text{obs}}(\overline{M}, b, \text{lo}, \text{hi}) \) records block \( b \) as allocated, with bounds \( \text{lo} \) and \( \text{hi} \), and returns the updated

\begin{figure}
\begin{tabular}{|c|c|c|}
\hline
Properties & Informal semantics \\
\hline\hline
\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 \\
\hline
\end{tabular}
\end{figure}

\begin{figure}
\begin{center}
\begin{tabular}{|c|c|}
\hline
\text{left values} & \text{variable} \\
\text{\_} & \ast\text{a} \\
\text{mem. values} & \text{pointer access} \\
\text{\_} & \text{left value} \\
\text{\_} & \text{pointer offset} \\
\text{\_} & \text{address} \\
\text{expressions} & \text{memory value} \\
\text{\_} & \text{arithmetic expr.} \\
\hline
\text{statements} & \text{skip} \\
\text{\_} & \text{assignment} \\
\text{\_} & \text{allocation} \\
\text{\_} & \text{deallocation} \\
\text{\_} & \text{assertion} \\
\text{\_} & \text{conditional} \\
\text{\_} & \text{loop} \\
\text{\_} & \text{sequence} \\
\text{predicates} & \text{comparators} \\
\text{\_} & \text{logic connectors} \\
\text{\_} & \text{a valid pointer} \\
\text{\_} & \ast\text{a initialized} \\
\hline
\text{terms} & \text{pure expressions} \\
\text{\_} & \text{base addr. of a's mem. block} \\
\text{\_} & \text{size of a's mem. block} \\
\text{\_} & \text{offset of a in its mem. block} \\
\text{\_} & \text{pure expr. combined with mem.-related constructs} \\
\hline
\text{types} & \text{integral type} \\
\text{\_} & \text{pointer type} \\
\hline
\end{tabular}
\end{center}
\caption{Formal syntax of the considered language.}
\label{fig:grammar}
\end{figure}
observation memory. Statement $\text{delete\_block}(M, b)$ marks $b$ as deallocating in $M$ and returns the updated observation memory. Statement $\text{initialize}(M, b, \delta)$ marks the content of address $(b, \delta)$ as initialized and returns the updated observation memory. Finally, $\text{read\_init}(M, b, \delta)$ reads the initialization status of the content at address $(b, \delta)$, 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:

- $M \models_{1v} l \equiv (b, \delta)$ evaluates a left-value $l$ to an address $(b, \delta)$ in a memory $M$;
- $M \models e \Rightarrow v$ evaluates an expression $e$ to a value $v$ in a memory $M$;
- $M \models t \Rightarrow v$ evaluates a term $t$ to a value $v$ in a memory $M$;
- $M \models p \Rightarrow P$ evaluates a predicate $p$ to a truth value $P$ in a memory $M$;
- $(s, M_1, M_1) \rightarrow (M_2, M_2)$ evaluates a statement $s$ in an execution memory $M_1$ and an observation memory $M_1$, then ends the execution in an execution memory $M_2$ and an observation memory $M_2$;
- $(s_1, M_1, M_1) \rightarrow (s_2, M_2, M_2)$ also evaluates a statement $s_1$ in an execution memory $M_1$ and an observation memory $M_1$, then continues the execution with the statement $s_2$ in an execution memory $M_2$ and an observation memory $M_2$.

We consider only execution of well typed programs (even if the simple type system is omitted here), thus equipped with a typing environment $\Gamma$. 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 $\text{Undefined}$.

#### Predicates

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.
Soundness of a Dataflow Analysis for Memory Monitoring

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 \( \rightarrow_M \) between blocks of a memory \( M \) as "given the memory state \( M, b \rightarrow_M b' \) if there is one value in \( b \) that is a pointer to \( b' \) with a valid offset". This definition is

\[
\begin{align*}
M & \models e \Rightarrow \text{PE-valid} \\
M & \models e \Rightarrow \text{PE-invalid} \\
M & \models e \Rightarrow \text{PE-init} \\
M & \models e \Rightarrow \text{PE-uninit} \\
M & \models e \Rightarrow \text{PE-eq} \\
M & \models e \Rightarrow \text{PE-neq}
\end{align*}
\]

Figure 5: Evaluation of terms and predicates.

\[
\begin{align*}
M & \models e \Rightarrow v \\
M & \models (l = e, M, M_1) \rightarrow (M_2, M_2) \\
M & \models (l = e, M, M_1) \rightarrow (M_2, M_2)
\end{align*}
\]

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

\[
\begin{align*}
M & \models e \Rightarrow v \\
M & \models (l = e, M, M_1) \rightarrow (M_2, M_2) \\
M & \models (l = e, M, M_1) \rightarrow (M_2, M_2)
\end{align*}
\]

Figure 7: Evaluation of control flow related statements.
formally expressed as follows:

\[ b \mapsto_M b' \text{ if } \exists \delta, \delta' \in \mathbb{Z} \text{ such that } \begin{align*}
M &\vdash (b, \delta) \\
M &\vdash (b', \delta') \\
1 \text{load}(M, b, \delta) &\Rightarrow \text{Some}(\text{Ptr}(b', \delta'))
\end{align*} \]

We define \( \mapsto^*_M \) 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 \( M \vdash a \Rightarrow \text{Ptr}(b_a, \delta_a) \), the set of reachable blocks from \( a \) in \( M \) is defined by:

\[ R_M(a) = \{ b \in \text{Blocks} \mid b_a \mapsto^*_M b \} \]

### 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 \( I(s) \) (resp. \( F(s) \)) 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 \( I(s) = 1 \) and \( F(s) = \{ 2 \} \).

![Figure 8: CFG of a simple program.](image)

We define the analysis as a pair \( (\text{live}_{\text{in}}, \text{live}_{\text{out}}) \) 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 \( l \), \( \text{live}_{\text{in}}(l) \) and \( \text{live}_{\text{out}}(l) \) 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 live 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 \( \text{gen} \) 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 \( \rho \) and \( \theta \). 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 \( \text{base} \), inductively defined as following:

\[
\begin{align*}
\text{base}(x) &= x \\
\text{base}(\ast x) &= \text{base}(a) \\
\text{base}(\& x) &= \text{base}(a) \\
\text{base}(a + + e) &= \text{base}(a).
\end{align*}
\]

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 \( \exists x \in \text{live}_{\text{out}}(l) \Rightarrow e \mapsto_x^* lv \). Indeed \( lv \) is monitored if and only if it is reachable from one of variables in \( \text{live}_{\text{out}}(l) \). 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 \( \mapsto^*_\mathcal{A} \), where \( e \mapsto^*_\mathcal{A} lv \) means that there is a path from \( e \) 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 observation memories, and state that they are the same in a certain set of blocks. Let \( M \) and \( M' \) be observation memories, and \( B \) be a set of blocks. We say that \( M \) and \( M' \) are equivalent on \( B \), and we note \( M \sim_B M' \) if they “have the same values on the blocks in \( B \), that is:

\[
\begin{align*}
\forall b \in B, \{ \\
\quad \forall \delta \in \mathbb{Z}, \quad M \vdash (b, \delta) &\Leftrightarrow M' \vdash (b, \delta) \\
\quad \forall \delta \in \mathbb{Z}, \quad \text{read}_{\text{init}}(M, b, \delta) &\Leftrightarrow \text{read}_{\text{init}}(M', b, \delta)
\}.
\end{align*}
\]

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 \( (s, M_1, \overline{M_1}) \rightarrow (M_2, \overline{M_2}) \) be an execution step, and let \( \overline{M_1} \) be an observation memory such that \( \overline{M_1} \sim_B \overline{M_1}' \), with \( B_1 = \bigcup_{x \in \text{live}_{\text{out}}(I(s))} R_M(x) \).

Then there exists \( \overline{M_2}' \) such that \( (s, M_1, \overline{M_1}') \rightarrow (M_2, \overline{M_2}') \) and \( \overline{M_2} \sim_B \overline{M_2}' \), with \( B_2 = \bigcup_{x \in \text{live}_{\text{out}}(I(s))} R_M(x) \).

**Continued Evaluation.**

**Theorem 4.2.** Let \( (s_1, M_1, \overline{M_1}) \rightarrow (s_2, M_2, \overline{M_2}) \) be an execution step, and let \( \overline{M_1} \) be an observation memory such that \( \overline{M_1} \sim_B \overline{M_1}' \), with \( B_1 = \bigcup_{x \in \text{live}_{\text{out}}(I(s_1))} R_M(x) \).

Then there exists \( \overline{M_2}' \) such that \( (s_1, M_1, \overline{M_1}') \rightarrow (s_2, M_2, \overline{M_2}') \) and \( \overline{M_2} \sim_B \overline{M_2}' \), with \( B_2 = \bigcup_{x \in \text{live}_{\text{out}}(I(s_2))} R_M(x) \).

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

\[
\begin{align*}
(s, M_1, \overline{M_1}) &\rightarrow (M_2, \overline{M_2}) \\
\sim_{B_1} &\sim_{B_1} \\
(s, M_1, \overline{M_1}') &\rightarrow (M_2, \overline{M_2}')
\end{align*}
\]
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 \( \text{live}_{\text{in}} \)\ and \( \text{live}_{\text{out}} \).

We use two kinds of auxiliary lemmas. Some describe properties of the equivalence relation \( \sim \) 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 \( M \sim_X M' \) and \( Y \subseteq X \) then \( M \sim_Y M' \).

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

**Lemma 5.3 (Equivalence Preservation (Initialization)).** For a given domain \( B \), 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:

\[
\forall a \in \text{Mem}_{\text{val}}, \forall M, \ R_M(a) \subseteq R_M(\&\text{base}(a)).
\]

**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 \( \text{store}(M_1, b, \delta, v) = \text{Some}(M_2) \) with \( b \notin R_{M_1}(\&x) \), then \( R_{M_1}(\&x) = R_{M_1}(\&x) \).

**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 \( \text{store}(M_1, b, \delta, \text{Int}(n)) = \text{Some}(M_2) \). Then for any variable \( x \), the set of reachable blocks from \( x \) is the same in \( M_1 \) and \( M_2 \):

\[
\forall x \in E, \ R_{M_1}(\&x) = R_{M_2}(\&x).
\]

**Lemma 5.7 (Maximum extension of a reachable set).** If \( M_1 \) and \( M_2 \) are two memory states such that \( \text{store}(M_1, b, \delta, \text{Ptr}(b_1, \delta_1)) = \text{Some}(M_2) \), then the following inclusion is verified: \( R_{M_1}(a) \subseteq R_{M_2}(a) \cup \{ b' | b_1 \mapsto M_1 b' \} \).

5.2 Case of Assignments.

We detail the main case of the induction: that of assignments. Given an execution step \( \langle \{ lv = e \}^{1 \cdot}, M_1, M'_1 \rangle \rightarrow (M_2, M'_2) \) and an observation memory \( M'_1 \) such that \( M'_1 \sim_{b_1} M'_2 \), we want to show the existence of \( M'_2 \) such that:

- \( \langle \{ lv = e \}^{1 \cdot}, M_1, M'_1 \rangle \rightarrow (M_2, M'_2) \) is a valid execution step
- \( M_2 \sim_{b_2} M'_2 \).

\( B_1 \) and \( B_2 \) are defined as:

\[
B_1 = \bigcup_{v \in \text{live}_{\text{in}}(l)} R_{M_1}(\&x) \quad B_2 = \bigcup_{v \in \text{live}_{\text{in}}(l)} R_{M_2}(\&x).
\]

The execution step has the following form:

\[
M_1 \models e \Rightarrow v \quad \text{store}(M_1, b_1, \delta_1, v) = \text{Some}(M_2) \quad M_1 \models lv \Leftrightarrow (b_1, \delta_1) \quad \text{initialize}(M_1, b_1, \delta_1) = M_2
\]

\[
\langle \{ lv = e \}^{1 \cdot}, M_1, M'_1 \rangle \rightarrow (M_2, M'_2)
\]

We define \( M'_2 = \text{initialize}(M'_1, b_1, \delta_1) \) and prove that \( M'_2 \sim_{b_2} M'_2 \). We distinguish between the case where the block written to is monitored \( b_1 \in B_2 \) and the case where it is not.

In both cases we make use of the following relation:

\[
M'_2 \sim_{b_1} M'_1 \sim_{b_1} M'_2
\]

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 \( b_1 \notin B_2 \). Let us prove that \( B_2 \subseteq B_1 \). Since \( \forall l, \text{live}_{\text{in}}(l) \supseteq \text{live}_{\text{out}}(l) \), it suffices to show that \( \exists x \in \text{live}_{\text{out}}(l), R_{M_2}(\&x) = R_{M_1}(\&x) \). Since for any of these sets we have \( b_1 \notin R_{M_2}(\&x) \) we can apply Lemma 5.5 to conclude.
We can now rewrite the assumption \( b_1 \notin B_2 \) as \( B_2 \subseteq (b_1)^c \) so that all domains involved in equation 1 are supersets of \( B_2 \). The conclusion follows from Lemma 5.1.

5.2.2 case \( b_1 \in B_2 \). Using Lemma 5.2, we consider \( B_2 \) as the (disjoint) union of \( B_2 \setminus \{b_1\} \) and \( \{b_1\} \), 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 \( B_2 \subseteq B_1 \), using the same method as previously. Applying Lemma 5.1 to 3 then yields \( M_2 \sim B_2 \setminus \{b_1\} \sim M_2' \).

Besides, since \( b_1 \in B_2 \subseteq B_1 \) and \( M \sim B_1 \), we have more specifically \( M_1 \sim \{b_1\} \sim M_1' \). By Lemma 5.3 we can conclude that \( M_2 \sim \{b_1\} \sim M_2' \).

subcase \( \tau \star \). If the expression is a pointer \( v = \text{Ptr}(b_\text{v}, \delta_\text{c}) \) for some \( (b_\text{v}, \delta_\text{c}) \), we can use Lemma 5.4 in conjunction with the definition of reachable blocks sets: \( R_M(a) \subseteq R_M(a) \cup R_M(e) \). We consider the union of these inclusions:

\[
B_2 \subseteq \bigcup_{x \in \text{live}_{out}(l)} R_M.(x) \cup R_M.(e).\tag{2}
\]

Let us prove that terms of the right side are both subsets of \( B_1 \). For the first one, using the definition of \( B_1 \) and the inclusion \( \text{live}_{out}(l) \subseteq \text{live}_{in}(l) \) we can write

\[
\bigcup_{x \in \text{live}_{out}(l)} R_M.(x) \subseteq \bigcup_{x \in \text{live}_{in}(l)} R_M.(x) = B_1.
\]

For the second one, we use Lemma 5.4 in conjunction with the fact that \( \text{base}(e) \in \text{live}_{out}(l) \). This corresponds to the first case in the definition of the generation function \( \text{gen}([lv = e]) \), which is defined by the condition: \( \exists x \in \text{live}_{out}(l), \&x \mapsto_A lv \). This condition is necessarily verified here: since \( b_1 \in B_2 \), by definition of \( B_2 \) there is some \( x \in \text{live}_{out}(l) \) such that \( b_1 \in R_M.(x) \). Let \( b_x \) be the block allocated for \( x \). By definition of \( R_M.(l) \), \( b_x \mapsto_A B_1 \). Finally, applying the correctness property of the points-to analysis to this relation yields the expected result: \( x \mapsto_A lv \) with \( x \in \text{live}_{out}(l) \).

Now we can deduce from equation 2 that \( B_2 \subseteq B_1 \), and conclude with the same arguments as for the subcase \( \text{int} \).

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 Framac-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 Check-Pointer 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 read 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


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 \( (s, M_1, M_1') \) be an execution step, and let \( M_1' \) be an observation memory such that \( M_1 \sim_{\mathcal{G}} M_1' \) with \( B_1 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s))} R_{M_1}(\delta x) \).

Then there exists \( M_2' \) such that \( (s, M_1, M_1') \rightarrow (M_2, M_2') \) and \( M_2 \sim_{\mathcal{G}} M_2' \) with \( B_2 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s))} R_{M_2}(\delta x) \).

Continued Evaluation. Let \( (s_1, M_1, M_1') \rightarrow (s_2, M_2, M_2') \) be an execution step, and let \( M_1' \) be an observation memory such that \( M_1 \sim_{\mathcal{G}} M_1' \) with \( B_1 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s_1))} R_{M_1}(\delta x) \).

Then there exists \( M_2' \) such that \( (s_1, M_1, M_1') \rightarrow (s_2, M_2, M_2') \) and \( M_2 \sim_{\mathcal{G}} M_2' \) with \( B_2 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s_1))} R_{M_2}(\delta x) \).

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 \( \sim_{\mathcal{G}} \)

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

Lemma C.2 (Domain Restriction). If \( M_1 \sim_X M_2 \) and \( Y \subseteq X \) then \( M_1 \sim_Y M_2 \).

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

The above properties are a direct consequence of the definition of \( \sim_{\mathcal{G}} \), 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 initialization 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.

D MEMORY OPERATIONS

Let \( (s, M_1, M_1') \rightarrow (M_2, M_2') \) be an execution step, and let \( M_1' \) be an observation memory such that \( M_1 \sim_{\mathcal{G}} M_1' \) with \( B_1 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s))} R_{M_1}(\delta x) \).

We want to show that there exists \( M_2' \) such that \( (s, M_1, M_1') \rightarrow (M_2, M_2') \) and \( M_2 \sim_{\mathcal{G}} M_2' \) with \( B_2 = \bigcup_{x \in \text{live}_{\text{fix}}(I(s))} R_{M_2}(\delta x) \).

D.1 Assignment

The execution step has the form:
Now we can deduce from 4 that $B_2 \subseteq B_1$, and conclude with the same arguments as for the subcase $\text{int}$.

### D.2 Allocation

**Eval-malloc**

$\begin{align*}
\text{store\_block}(M_1, b, \text{lo}, \text{hi}) &= M_2 \\
\text{initialize}(M_3, b_1, \delta_1) &= M_2
\end{align*}$

1. $\{lv = \text{malloc}(e); [\ell, M_1, \delta_1] \rightarrow (M_2, M_2)\}$

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:

$\begin{align*}
M'_2 &= \text{store\_block}(M_1, b, \text{lo}, \text{hi}) \\
M'_2 &= \text{initialize}(M_3, b_1, \delta_1) \\
B_3 &= \bigcup_{x \in \text{live\_in}(l)} R_{M_3}(\{x\})
\end{align*}$

We proceed in two steps:

1. prove $M'_3 \sim_{B_3} M'_3$
2. use the previous equivalence to prove $M'_3 \sim_{B_2} M_2$.

#### 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,

$\forall x \in \text{live\_in}(l), R_{M_3}(\{x\}) = R_{M_1}(\{x\})$

and thus $B_3 = B_1$.

Since $M_3 \sim_{B_1} M'_3$ and the same allocation is performed on both memories, by Lemma C.5 the equivalence is preserved: $M_3 \sim_{B_1} M'_3$. $B_1$ and $B_3$ 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 $M_2 \sim_{B_2} M'_2$, assuming the equivalence on the intermediate state: $M_3 \sim_{B_3} M'_3$, $b$ being a freshly allocated block in both $M_3$ and $M'_3$, we have $M_3 \sim_{\{b\}} M'_3$, which means that we can extend the equivalence domain by using Lemma C.3.

$\begin{align*}
M_3 &\sim_{B_3 \cup \{b\}} M'_3
\end{align*}$

**case $b \notin B_3$.** Then by Lemma C.8, $\forall x \in \text{live\_in}(l), R_{M_3}(\{x\})$ is unaffected by the write at $b_1$, so $B_3 = B_2$. We can thus use Lemma C.4 to transfer the equivalence to the next program state: $M_2 \sim_{B_3 \cup \{b\}} M'_2$. To conclude, we use the domain equality $B_3 = B_2$, and restrict the domain back to $B_2$ by Lemma C.2.

**case $b \in B_3$.** By Lemma C.10, $\forall x \in \text{live\_in}(l), R_{M_3}(\{x\}) \subseteq R_{M_3}(\{x\}) \cup \{b\}$. Considering the union on all $x \in \text{live\_in}(l)$ we get: $B_2 \subseteq B_3 \cup \{b\}$. Conclusion follows by Lemma C.4.
D.3 Dealllocation

**Eval-free**

\[ M_1 \models_e a \Rightarrow (b, 0) \]

\[ \text{free}(M_1, b) = \text{Some}(M_2) \quad \text{delete}\_\text{block}(M_1, b) = M_2 \]

\[ \langle \text{free}(a) ; t \rangle, M_1, M_2 \rangle \to (M_2, M_2) \]

\( M_2 \) 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 \( x, R_{M_1}(x) \subseteq R_{M_2}(x) \). Since \( \text{live}_1(l) \supseteq \text{live}_{out}(l) \) by definition of the transfer function, this implies \( B_1 \supseteq B_2 \). Therefore, by domain restriction (Lemma C.2), \( M_1 \sim_{B_1} M_1' \). Conclusion follows from Lemma C.6.

## E OTHER CASES OF IMMEDIATE TERMINATION

### E.1 Skip

**Eval-skip**

\[ \langle \text{skip}, M_1, M_1 \rangle \to (M_1, M_1) \]

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

### E.2 End of a loop

**Eval-whileEnd**

\[ M \models_e e \Rightarrow \text{Int}(0) \]

\[ \langle \text{while } l \{[e^1]\} s, M, M \rangle \to (M, M) \]

By definition of the functions \( \text{live}_{in}, \text{live}_{out} \), \( \text{live}_{in}(l) \supseteq \text{live}_{out}(l) \). Since the memory is unchanged, this implies \( B_1 \supseteq B_2 \). Conclude with Lemma C.2.

### E.3 Assertion

**Eval-assert**

\[ M \models_p p \Rightarrow \top \]

\[ \langle \text{assert } [p] \Rightarrow \text{Int}; \text{undef}, M, M \rangle \to (M, M) \]

\( \text{live}_{in}(l) \supseteq \text{live}_{out}(l) = \text{live}_{in}(I(s)) \) by definition of the analysis. Since the memory is unchanged, this implies \( B_1 \supseteq B_2 \) and we conclude with Lemma C.2.

## F CASES OF CONTINUED EXECUTION

Let \( \langle s_1, M_1, M_1 \rangle \to \langle s_2, M_2, M_2 \rangle \) be an execution step, and let \( M_1' \) be an observation memory such that \( M_1 \sim_{B_1} M_1' \), with \( B_1 = \bigcup_{x \in \text{live}_{in}(I(s))} R_{M_1}(x) \).

We want to show that there exists \( M_2' \) such that \( \langle s_1, M_1, M_1' \rangle \to \langle s_2, M_2, M_2' \rangle \) and \( M_2 \sim_{B_2} M_2' \) with \( B_2 = \bigcup_{x \in \text{live}_{in}(I(s))} R_{M_2}(x) \).

Case addressed in this section is 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 \( B_1 \supseteq B_2 \), it suffices to prove the corresponding live variable sets inclusion \( \text{live}_{in}(I(s_1)) \supseteq \text{live}_{in}(I(s_2)) \). In all the following cases, this inclusion is a direct consequence of the relation between the dataflow analysis result \( (\text{live}_{in}, \text{live}_{out}) \) and the CFG structure, or in other words the fact that \( (\text{live}_{in}, \text{live}_{out}) \) 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 (assignment, allocation...), or it is a composed statement (loop, sequence, conditional...).

### F.1.1 Subcase eval-seqEnd.

\[ \text{eval-seqEnd} \]

\[ \langle s_1, M_1, M_1 \rangle \to \langle s_2, M_2, M_2 \rangle \]

\( \text{live}_{out}(I(s_1)) \supseteq \text{live}_{in}(I(s_2)) \) by definition of the analysis, so \( B \supseteq B_2 \) and we can restrict the equivalence to \( B_2 \) by Lemma C.2: \( M_2 \sim_{B_2} M_2' \). We conclude by replacing \( M_2 \) with \( M_2' \) in the evaluation derivation:

\[ \langle s_1, M_1, M_1 \rangle \to \langle s_2, M_2, M_2' \rangle \]

### F.1.2 Subcase eval-seqCont.

\[ \text{eval-seqCont} \]

\[ \langle s_1, M_1, M_1 \rangle \to \langle s_2, M_2, M_2 \rangle \]

The proof is the same as in the previous case, substituting \( \text{live}_{in}(I(s_1)) \) for \( \text{live}_{out}(I(s_1)) \).

## F.2 Conditional Branching

### F.2.1 Eval-ifTrue

\[ M \models_e e \Rightarrow \text{Int}(n) \quad n \neq 0 \]

\[ \langle \text{if } [e^1] \text{ then } s_f \text{ else } s_f, M, M \rangle \to (s_f, M, M) \]

Here \( M_1, M_1', M_2, M_2' \) are all equal (to \( M \)). Therefore it suffices to show that \( B_1 \supseteq B_2 \), that is:

\[ \bigcup_{x \in \text{live}_{in}(l)} R_{M_1}(x) = \bigcup_{x \in \text{live}_{in}(I(s_f))} R_{M_2}(x) \]

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

\( \text{live}_{in}(l) \supseteq \text{live}_{in}(I(s_f)) \) \( \cup \text{live}_{in}(I(s_f)) \)

(definition of the transfer function).

### F.2.2 Eval-ifFalse

\[ M \models_e e \Rightarrow \text{Int}(0) \]

\[ \langle \text{if } [e^1] \text{ then } s_f \text{ else } s_f, M, M \rangle \to (s_f, M, M) \]

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

\( \text{live}_{in}(l) \supseteq \text{live}_{in}(I(s)) \)

which results from the definition of \( (\text{live}_{in}, \text{live}_{out}) \).