Verified Runtime Assertion Checking for Memory Properties

Dara Ly[1,4], Nikolai Kosmatov[1,2], Frédéric Loulergue[3,4], and Julien Signoles[1]

[1] CEA, LIST, Software Security and Reliability Laboratory, Palaiseau, France
firstname.lastname@cea.fr
[2] Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com
[3] Northern Arizona University, School of Informatics Computing and Cyber Systems, Flagstaff, USA
frederic.loulergue@nau.edu
[4] Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France

Abstract. Runtime Assertion Checking (RAC) for expressive specification languages is a non-trivial verification task, that becomes even more complex for memory-related properties of imperative languages with dynamic memory allocation. It is important to ensure the soundness of RAC verdicts, in particular when RAC reports the absence of failures for execution traces. This paper presents a formalization of a program transformation technique for RAC of memory properties for a representative language with memory operations. It includes an observation memory model that is essential to record and monitor memory-related properties. We prove the soundness of RAC verdicts with regard to the semantics of this language.

1 Introduction

Runtime assertion checking (RAC) [7] is a well-established verification technique whose goal is to evaluate specified program properties (assertions, or more generally, annotations) during a particular program run and to report any detected failures. It is particularly challenging for languages like C, where memory-related properties (such as pointer validity or variable initialization) cannot be directly expressed in terms of the language, while their evaluation is crucial to ensure the soundness of the program and to avoid the numerous cases of *undefined behavior* [12]. Indeed, memory-related errors, such as invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, are very common. A study from IBM [29] reports that about 50% of detected software errors were related to pointers and array accesses.

Recent tools addressing memory safety of C programs, such as Valgrind and MemCheck [26,23], DrMemory [5] or AddressSanitizer [25], have become very popular and successful in detecting bugs. However, their soundness is usually not formally established, and often does not hold, since most of them rely on very efficient but possibly unsound heuristics [31]. While for a reported bug, it can be possible—at least, in theory—to carefully analyze the execution and check whether an error is correctly reported, the soundness of the "no-bug" verdict cannot be checked.

For runtime assertion checking, soundness becomes a major concern: this technique is used to verify the absence of failures, often in complement to sound deductive verification on parts of annotated code which were not (yet) proved. It is the case of the E-ACSL tool [28], part of the Frama-C verification platform [16] for static and dynamic analyses of C programs. A *formal proof of soundness* is highly desirable with regard to the complexity of verification of memory-related properties, that requires numerous instrumentation steps to record memory related operations—often in a complex, highly optimized *observation memory model* [17,13,32]— and to evaluate them thanks to this record. In this context, the proof of soundness is highly non-trivial: it requires to formalize not only the semantics of the considered programming and specification languages, but also the program transformation and the observation memory.

The purpose of the present work is to formalize and prove the soundness of a runtime assertion checker for memory-related properties. We consider a simple but representative imperative programming language with dynamic memory allocation and a specification language with a complete set of memory-related predicates, including pointer validity, variable initialization, as well as pointer offset, base address and size of memory blocks. We define their semantics and formalize a runtime assertion checker for these languages, including the underlying program transformation and observation memory model. Finally, we state and prove the soundness result ensuring that the resulting verdicts are correct with respect to the semantics.

The contributions of the paper include:

- a formalization of all major steps of a runtime assertion checker for a simple but representative language;
- a definition of a dedicated memory model for RAC with an observation memory, suitable for a modular definition and verification of program transformations injecting non-interfering code, and an associated proof technique;
- a proof of soundness of a runtime verifier for memory properties.

Outline. Section 2 gives an overview of the work and a motivating example. Section 3 defines the considered languages. The runtime assertion checker is formalized in Section 4, while Section 5 states and proves the soundness result. Finally, Sections 6 and 7 give some related work and conclusion.

2 Overview and motivating example

At a first glance, runtime assertion checking might be considered as an easy task: just directly translate each logic term and predicate from the source specification language to the corresponding expression of the target programming language and that's it. In that spirit, Barnett et al. [2] explain how they enforce Spec# contracts, but only a short paragraph is dedicated to their runtime checker (all the others being dedicated to static verifications). Here it is *in extenso*:

> The run-time checker is straightforward: each contract indicates some particular program points at which it must hold. A run-time assertion is generated for each, and any failure causes an exception to be thrown.

```
1  int search(int *t, int len, int x) { // search x in array t of size len
2    int lo = 0, hi = len - 1;          // initial search interval bounds
3    while (lo <= hi) {                 // while search interval non empty
4      int mid = lo + (hi - lo) / 2;    // take the middle value
5      /*@ assert(\valid(t + mid)); */
6      if (t[mid] == x) return mid;     // element found
7      else if (t[mid] < x) lo = mid + 1;
8      else hi = mid - 1;               // reduce the search interval
9    }
10   return -1;                         // element not found
11 }
12
13 int main(void) {
14   int t[5] = { -3, 2, 4, 7, 10 };
15   return search(t, 10, 7);
16 }
```

Fig. 1. Binary search annotated with a memory-related property.

However, this statement is not true for complex properties such as *memory properties*. Consider for instance the C function implementing binary search in Fig. 1. It contains an assertion at line 5, written in the E-ACSL specification language [9,27], stating that `t+mid` of type **`int`**`*` is a "valid memory location", ensuring that it is safe to dereference it at lines 6 and 7.

Checking such a property at runtime is not trivial: in particular, it requires to know at the annotation's program point (line 5) whether the **`sizeof(int)`** bytes starting from the address `t+mid` have been properly allocated by the program earlier in the execution, in the same memory block, without being freed in the meantime. For that purpose, runtime memory checkers (also called memory debuggers) need to store at runtime pieces of information about program memory in a disjoint memory space, named *observation memory* in this paper. For instance, the instrumented version of Fig. 1 created by the E-ACSL runtime assertion checker [28] is 111-line long (when deactivating its static optimization described in [21]) for tracking the program's memory manipulation. In particular, for the block `t` created and initialized at line 14, E-ACSL adds the following lines of code (assuming that **`sizeof(int)`** = 4, so `t` is 20-byte long):

```
__e_acsl_store_block((void *)(t),(size_t)20); //record new block
__e_acsl_full_init((void *)(& t)); //mark it as initialized
```

Optimized implementations of such functions are also pretty complex, as explained by Vorobyov et al. [32]. In this work, assuming their correct implementation, we *formalize* the whole instrumentation performed by a RAC tool, and *prove its soundness*. For that purpose, we provide a model for such functions.

Moreover, RAC often has to manipulate additional variables, e.g. to evaluate annotations. We also prove that the instrumentation has no effect on the functional behavior of the input program as long as no annotation is violated. For that purpose, we add a new memory space, named *observation memory*, that helps to prove non-interference in a modular way.

3 The Considered Languages

We model the instrumentation operated by RAC as a program transformation from a source language with logical assertions to a destination one with program

$$
\begin{array}{rlr}
e ::= & n & \text{integer constant} \\
| & x & \text{variable} \\
| & *e & \text{dereference} \\
| & \&e & \text{address} \\
| & \dagger\ e & \text{unary operator} \\
| & e\ \ddagger\ e & \text{binary operator} \\
\\
t ::= & e & \text{expression} \\
| & \bar{*}t & \text{dereference} \\
| & \bar{\&}t & \text{address} \\
| & \bar{\dagger}\ t & \text{unary operator} \\
| & t\ \bar{\ddagger}\ t & \text{binary operator} \\
| & \backslash\texttt{base\_address}(t) & \text{base address} \\
| & \backslash\texttt{offset}(t) & \text{pointer offset} \\
| & \backslash\texttt{block\_length}(t) & \text{block length} \\
\\
p ::= & \backslash\texttt{true} \mid \backslash\texttt{false} & \text{true, false} \\
| & t \bowtie t & \text{comparison} \\
| & p \wedge p & \text{conjunction} \\
| & p \vee p & \text{disjunction} \\
| & p \Rightarrow p & \text{implication} \\
| & \neg\, p & \text{negation} \\
| & \backslash\texttt{valid}(t) & \text{pointer validity} \\
| & \backslash\texttt{initialized}(t) & \text{initialization} \\
\\
s ::= & \texttt{skip;} & \text{noop} \\
| & e = e; & \text{assignment} \\
| & e = \texttt{malloc}(e); & \text{allocation} \\
| & \texttt{free}(e); & \text{deallocation} \\
| & \texttt{logical\_assert}(p); & \text{log. assertion} \\
| & s\ s & \text{sequence} \\
| & \texttt{if}(e)\ \texttt{then}\ s\ \texttt{else}\ s & \text{branching} \\
| & \texttt{while}(e)\ s & \text{loop} \\
| & \{\boldsymbol{d}\ s\} & \text{code block} \\
\\
d ::= & \tau\ x; & \text{var. declaration} \\
\\
\tau ::= & sgn\ sz & \text{integer type} \\
| & \tau* & \text{pointer type} \\
\\
sgn ::= & \texttt{unsigned} \mid \texttt{signed} & \\
sz ::= & \texttt{int} \mid \texttt{long} &
\end{array}
$$

Fig. 2. Syntax of the source language, with expressions e, logical terms t, predicates p, statements s, declarations d, types τ, signedness sgn and size sz.

assertions and observation memory primitives. We describe both languages in this section, before defining the program transformation in the next section.

3.1 Source Language

Our source language is a small C-like imperative language extended with formal annotations. It focuses on memory-related constructs and properties.

Syntax. Figure 2 presents the syntax of this source language. Expressions are (integer) constants, variables and operators (e.g. arithmetic operators), as well as the distinguished reference (&) and dereference ($*$) operators. Variables are implicitly type-annotated, and all programs are supposed well-typed with respect to a type system that we do not detail here.

Statements include assignment of a value to a memory location (variable or dereferenced pointer) and basic control flow (sequence, conditional branching, loop). Beside these, notable constructs are primitives for dynamic memory allocation and deallocation, the `logical_assert`(p); statement (which does nothing if predicate p evaluates to True and halts the execution otherwise), and code blocks with (possibly multiple) local variable declarations (denoted $\boldsymbol{d}$).

Predicates form a propositional calculus (with the usual conjunction, disjunction, negation, and implication connectives), whose atoms are pointer validity, pointed value initialization, and logical term comparison. Terms are a superset of C expressions, extended with block-level memory attributes such as the length of the block containing the pointer, the base address of the pointer (i.e. the address of the first byte of its block), or the offset of the pointer with regards to the base address. To express this extension, terms have to include syntactical constructs mapping those of expressions, denoted with an overline: for instance $\bar{*}$ denotes pointer dereferencing *for terms*.

Semantics Overview. We give our language a big-step operational semantics adapted from that of CompCert's Clight [4]. The choice of this style (rather than, say, small-step operational semantics) is motivated by its ease of use when reasoning about program transformations. Moreover, the semantics is *blocking* [8]: in case of an error, the evaluation cannot evolve.

The evaluation context is composed of a variable environment $\widehat{E}$ (mapping variables' names to memory blocks) and a memory state $\widehat{M}$ (mapping block offsets to values, as explained below). Five inductive relations define our semantics:

- $\widehat{E}, \widehat{M} \vDash_{\mathrm{e}} e \Rightarrow v$, evaluation of an expression e in the context of a variable environment $\widehat{E}$ and a memory state $\widehat{M}$, yielding a value v;
- $\widehat{E}, \widehat{M} \vDash_{\mathrm{lv}} e \Rightarrow (b, \delta)$, evaluation of an expression e as a left-value, yielding a memory location (b, δ) in a memory block b with an offset δ;
- $\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Rightarrow v$, evaluation of a logical term t, similarly yielding a value v;
- $\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p \Rightarrow \mathbf{b}$, evaluation of predicate p to a boolean truth value $\mathbf{b}$;
- $\widehat{E}, \widehat{M_1} \vDash_{\mathrm{s}} s \Rightarrow \widehat{M_2}$, evaluation of statement s in the context of a variable environment $\widehat{E}$ and an initial memory state $\widehat{M_1}$; the evaluation results in a final memory state $\widehat{M_2}$, while the environment $\widehat{E}$ remains the same.

To distinguish source and destination language judgments, environments and memory states of the source language are written with a hat: $(\widehat{E}, \widehat{M})$, while that of the destination language without it: (E, M).

Memory Model. In accordance with our choice of using CompCert as an inspiration for our semantics, we reuse the (first) memory model of CompCert [20], based on the notion of memory blocks. A memory location l in this model is a couple (b, δ) where b is the memory block l belongs to and $\delta \in \mathbb{Z}$ the offset of l within b. Blocks have *bounds* defined at allocation time, which determine the possible offset interval where a value may effectively be stored.

Following [20], the type of such a memory state is left abstract. It is only defined by four axiomatized operations over it that are informally described here. For that purpose, this paper uses the following notation: M (or $\widehat{M}$) denotes a memory state, b a block, δ an offset, v a value, and τ a type. As memory operations may fail, their return value has an option type, meaning that such a value is either ε (no return value) of $\lfloor v \rfloor$ (some value v).

Thus, $\mathrm{alloc}(M, lo, hi) = \lfloor (b, M') \rfloor$ means that the allocation of a new block in memory state M succeeds and returns the identifier b of the new block, along with the new memory state M'; in M', b is allocated with lower bound lo (inclusive) and higher bound hi (exclusive). $\mathrm{bounds}(M, b)$ returns the bounds recorded for b. If the operation fails, ε is returned. Conversely, $\mathrm{free}(M, b)$ deallocates a block b from M. If b was allocated in M and not previously deallocated, a new memory state $\lfloor M' \rfloor$ is returned. Otherwise, the deallocation fails and returns ε. $\mathrm{store}(\tau, M, b, \delta, v)$ stores value v with type τ at location (b, δ) in M, returning a new memory state $\lfloor M' \rfloor$ if it succeeds. The store can fail and return ε, for instance if it is out of b's bounds. Finally, $\mathrm{load}(\tau, M, b, \delta)$ reads from M at (b, δ) a value of type τ, returning a value $\lfloor v \rfloor$ upon success and ε upon failure.

$$\text{E\_VAR:}\ \frac{\widehat{E}(x) = b}{\widehat{E}, \widehat{M} \vDash_{\text{lv}} x \Rightarrow (b, 0)} \qquad \text{E\_DEREF:}\ \frac{\widehat{E}, \widehat{M} \vDash_{\text{e}} a \Rightarrow \texttt{Ptr}(b, \delta)}{\widehat{E}, \widehat{M} \vDash_{\text{lv}} *a \Rightarrow (b, \delta)} \qquad \text{E\_ADDR:}\ \frac{\widehat{E}, \widehat{M} \vDash_{\text{lv}} l \Rightarrow (b, \delta)}{\widehat{E}, \widehat{M} \vDash_{\text{e}} \&l \Rightarrow \texttt{Ptr}(b, \delta)}$$

$$\text{E\_LVAL:}\ \frac{\widehat{E}, \widehat{M} \vDash_{\text{lv}} l \Rightarrow (b, \delta) \quad \text{typeof}(l) = \tau \quad \text{load}(\tau, \widehat{M}, b, \delta) = \lfloor v \rfloor \quad v \neq \text{Undef}}{\widehat{E}, \widehat{M} \vDash_{\text{e}} l \Rightarrow v} \qquad \text{E\_INT:}\ \frac{}{\widehat{E}, \widehat{M} \vDash_{\text{e}} n \Rightarrow \texttt{Int}(n)}$$

Fig. 3. Semantics of expressions.

Semantics Inference Rules. The relations expressing the semantics of our source language are defined by a set of inference rules. Expressions (see Figure 3) evaluate either to a value, or, as left-values, to a memory location. A value is either an integer, a pointer to a memory location (that is, in our memory model, a block and an offset), or an undefined value: $v ::= \text{Int}(n) \mid \text{Ptr}(b, \delta) \mid \text{Undef}$.

Figure 4 defines the semantics of statements. Rule E_ASSIGN is an example of use of the memory model: the right-hand side of the assignment is evaluated to a value v, while the left-hand side is evaluated to a memory location (b, δ). A store() operation is then performed to write v into $\widehat{M_1}$ at location (b, δ), and must lead to a final memory state $\widehat{M_2}$ (recall our semantics is blocking). Selected rules defining the semantics of predicates and terms are given in Figure 5.

3.2 Destination Language

The destination language is quite close to the source language: it has the same expressions, and mostly the same statements (see Figure 6). The first difference is the absence of assertions over logical predicates, therefore removing the need for terms and predicates. These are substituted with a weaker, program assertion over expressions, similar to the `C assert` macro. The other difference is the addition of a set of primitives to interact with an additional *observation memory*. In order to give these primitives a semantics, we extend the evaluation relation with the state of the observation memory (denoted $\overline{M}$). Consequently, evaluation relations for the destination language take the following shapes:

- $E, M \vDash_{\text{e}} e \Rightarrow v$, evaluation of an expression (unchanged);
- $E, M \vDash_{\text{lv}} e \Rightarrow b, \delta$, evaluation of an expression as a left-value (unchanged);
- $E, M_1, \overline{M_1} \vDash_{\text{s}} s \Rightarrow M_2, \overline{M_2}$, evaluation of a statement; in addition to the final execution memory M_2, it also returns a final observation memory $\overline{M_2}$.

In the same way as the execution memory model is a prerequisite to the definition of the source language semantics, the observation memory must be defined prior to the semantics of the above primitives. The observation memory is basically a data structure for the runtime monitor to store metadata about the (execution) memory of the program under monitoring. As for the execution memory model, we define it with an abstract type, a set of functions over this type, and an axiomatization of these functions. Four of them are the observation counterparts of the execution memory operations. store_block$(\overline{M}, b, lo, hi)$ records block b as being allocated with bounds lo and hi, returning an updated

$$\text{E\_ASSIGN:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow v \qquad \text{store}(\tau, \widehat{M_1}, b, \delta, v) = \lfloor \widehat{M_2} \rfloor \qquad \widehat{E}, \widehat{M_1} \vDash_{\text{lv}} l \Rightarrow (b, \delta) \qquad \text{typeof}(e) = \tau}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} l = e; \Rightarrow \widehat{M_2}}$$

$$\text{E\_MALLOC:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow \text{Int}(n) \qquad \text{alloc}(\widehat{M_1}, 0, n) = (b', \widehat{M_2}) \qquad \widehat{E}, \widehat{M_1} \vDash_{\text{lv}} l \Rightarrow (b, \delta) \qquad \text{typeof}(l) = \tau* \qquad \text{store}(\tau*, \widehat{M_2}, b, \delta, \text{Ptr}(b', 0)) = \lfloor \widehat{M_3} \rfloor}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} l = \texttt{malloc}(e); \Rightarrow \widehat{M_3}}$$

$$\text{E\_FREE:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow \text{Ptr}(b, 0) \qquad \text{free}(\widehat{M_1}, b) = \lfloor \widehat{M_2} \rfloor}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} \texttt{free}(e); \Rightarrow \widehat{M_2}}$$

$$\text{E\_LOGICAL\_ASSERT:}\quad \frac{\widehat{E}, \widehat{M} \vDash_{\text{p}} p \Rightarrow \text{true}}{\widehat{E}, \widehat{M} \vDash_{\text{s}} \texttt{logical\_assert}(p); \Rightarrow \widehat{M}}$$

$$\text{E\_SEQ:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_1 \Rightarrow \widehat{M_2} \qquad \widehat{E}, \widehat{M_2} \vDash_{\text{s}} s_2 \Rightarrow \widehat{M_3}}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_1\ s_2 \Rightarrow \widehat{M_3}}$$

$$\text{E\_IF\_FALSE:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow \text{Int}(0) \qquad \widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_2 \Rightarrow \widehat{M_2}}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \widehat{M_2}}$$

$$\text{E\_IF\_TRUE:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow \text{Int}(n) \qquad n \neq 0 \qquad \widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_1 \Rightarrow \widehat{M_2}}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \widehat{M_2}}$$

$$\text{E\_WHILE\_FALSE}\quad \frac{\widehat{E}, \widehat{M} \vDash_{\text{e}} e \Rightarrow \text{Int}(0)}{\widehat{E}, \widehat{M} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Rightarrow \widehat{M}}$$

$$\text{E\_WHILE\_TRUE:}\quad \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{e}} e \Rightarrow \text{Int}(n) \qquad n \neq 0 \qquad \widehat{E}, \widehat{M_1} \vDash_{\text{s}} s \Rightarrow \widehat{M_2} \qquad \widehat{E}, \widehat{M_2} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Rightarrow \widehat{M_3}}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Rightarrow \widehat{M_3}}$$

$$\text{E\_BLOCK:}\quad \frac{\widehat{E_2}, \widehat{M_2} = \text{alloc\_vars}(\boldsymbol{d}, \widehat{E_1}, \widehat{M_1}) \qquad \widehat{E_2}, \widehat{M_2} \vDash_{\text{s}} s \Rightarrow \widehat{M_3} \qquad \widehat{M_4} = \text{dealloc\_vars}(\boldsymbol{d}, \widehat{E_2}, \widehat{M_3})}{\widehat{E_1}, \widehat{M_1} \vDash_{\text{s}} \{\boldsymbol{d}\ s\} \Rightarrow \widehat{M_4}}$$

Fig. 4. Semantics of the source language statements, where alloc_vars() allocates memory for the list of local variable declarations $\boldsymbol{d}$ using the alloc() operation, and adds the corresponding bindings into the environment. dealloc_vars() is the converse function.

observation memory state. delete_block$(\overline{M}, b)$ marks b as deallocated and returns an updated observation memory. initialize$(\tau, \overline{M}, b, \delta)$ marks the data with type τ at location (b, δ) as initialized and returns an updated observation memory. Conversely, is_initialized$(\tau, \overline{M}, b, \delta)$ returns 1 if location (b, δ) with type τ is marked as initialized in $\overline{M}$, and 0 otherwise. Two other functions provide information about *metadata* stored in the memory state: is_valid$(\tau, \overline{M}, b, \delta)$ returns 1 if accessing data with type τ at location (b, δ) is legal, and 0 otherwise, while bounds$(\overline{M}, b)$ returns the bounds that were recorded for b with store_block(). Vorobyov et al. explain how all these operations can be implemented [32].

Figure 7 presents the semantics of the destination language's additional statements, and their relation with the observation memory operations. Evaluation rules for the statements already present in the source language are omitted, as they are similar and only adapted to include observation memory states.

4 Program Transformation

We now turn to the implementation of a runtime monitor by program transformation. This transformation has two purposes: first, translating logical predi-

$$\text{E\_OR1:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} p_1 \Rightarrow \text{true}}{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} p_1 \vee p_2 \Rightarrow \text{true}} \qquad \text{E\_OR2:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} p_1 \Rightarrow \text{false} \qquad \widehat{E},\widehat{M} \vDash_{\mathrm{p}} p_2 \Rightarrow \mathbf{b}}{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} p_1 \vee p_2 \Rightarrow \mathbf{b}}$$

$$\text{E\_INIT\_TRUE:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} a \Rightarrow \texttt{Ptr}(b,\delta) \qquad \mathrm{load}(\tau,M,b,\delta) = \lfloor v \rfloor \qquad v \neq \text{Undef}}{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} \texttt{initialized}(a) \Rightarrow \text{true}}$$

$$\text{E\_INIT\_FALSE:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} a \Rightarrow \texttt{Ptr}(b,\delta) \qquad \mathrm{load}(\tau,M,b,\delta) = \lfloor \text{Undef} \rfloor}{\widehat{E},\widehat{M} \vDash_{\mathrm{p}} \texttt{initialized}(a) \Rightarrow \text{false}}$$

$$\text{E\_BASE\_ADDR:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} a \Rightarrow \texttt{Ptr}(b,\delta)}{\widehat{E},\widehat{M} \vDash_{\mathrm{t}} \backslash\texttt{base\_address}(a) \Rightarrow \texttt{Ptr}(b,0)} \qquad \text{E\_OFS:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} a \Rightarrow \texttt{Ptr}(b,\delta)}{\widehat{E},\widehat{M} \vDash_{\mathrm{t}} \backslash\texttt{offset}(a) \Rightarrow \texttt{Int}(\delta)}$$

$$\text{E\_BLOCK\_LENGTH:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} a \Rightarrow \texttt{Ptr}(b,\delta) \qquad \mathrm{bounds}(\widehat{M},b) = \lfloor lo, hi \rfloor}{\widehat{E},\widehat{M} \vDash_{\mathrm{t}} \backslash\texttt{block\_length}(a) \Rightarrow \texttt{Int}(hi - lo)} \qquad \text{E\_EXPR:}\quad \frac{\widehat{E},\widehat{M} \vDash_{\mathrm{e}} e \Rightarrow v}{\widehat{E},\widehat{M} \vDash_{\mathrm{t}} e \Rightarrow v}$$

Fig. 5. Semantics of predicates and terms.

$s ::=$	...	source lang. stmts	| `e = is_initialized(`e`);`	is $(*e)$ initialized
|	~~`logical_assert(`p`);`~~	no assert. over pred.	| `initialize(`e`);`	mark $*e$ as initialized
|	`assert(`e`);`	assert. over exp.	| `e = base_address(`e`);`	e's block base address
|	`store_block(`e,e`);`	record new block	| `e = offset(`e`);`	get pointer offset
|	`delete_block(`e`);`	remove recrorded bl.	| `e = block_length(`e`);`	e's block length
|	`e = is_valid(`e`);`	is e valid		

Fig. 6. Additional statements of the destination language.

cates (and terms) into chunks of executable code evaluating them; and second, inserting statements into the original code, in order to track the state of the execution memory; that is, updating the observation memory whenever a memory related operation occurs.

The general idea underlying this transformation is the following: atomic predicates and terms are translated into dedicated primitives of the target language, while composite ones (logical connectors, comparison operators...) are encoded with non-logical constructs of the source language. The translation of each term and predicate introduces a specific variable *res* that stores its results for later use by subsequent computations.

Formally, we express the transformation as a set of three recursive functions over statements (denoted $[\![\cdot]\!]_s$), predicates ($[\![\cdot]\!]_p$) and terms ($[\![\cdot]\!]_t$). Notice that indices s, p, t are here part of notation (and not a reference to a specific statement s, predicate p or term t). These functions have the following types: $[\![\cdot]\!]_s$: statement $\rightarrow$ statement; $[\![\cdot]\!]_p$: predicate $\rightarrow \{code : \text{statement}; res : \text{variable}\}$; $[\![\cdot]\!]_t$: term $\rightarrow \{code : \text{statement}; res : \text{variable}\}$. While $[\![\cdot]\!]_s$ is a straightforward translation from statement to statement, the other two translation functions return records; their fields are a statement (the *code* field of the record type) performing computation of the translated term or predicate, and distinguished variable *res* to store the result of the computation.

$$\text{E\_STOREBLOCK:}\quad \frac{E, M_1 \vDash_{\text{e}} p \Rightarrow \text{Ptr}(b, 0) \qquad E, M_1 \vDash_{\text{e}} e \Rightarrow n \qquad \text{store\_block}(\overline{M_1}, b, 0, n) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} \texttt{store\_block}(p, e); \Rightarrow M_1, \overline{M_2}}$$

$$\text{E\_DELETEBLOCK:}\quad \frac{E, M_1 \vDash_{\text{e}} p \Rightarrow \text{Ptr}(b, 0) \qquad \text{delete\_block}(\overline{M_1}, b) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} \texttt{delete\_block}(p); \Rightarrow M_1, \overline{M_2}}$$

$$\text{E\_ISVALID:}\quad \frac{E, M_1 \vDash_{\text{lv}} e_1 \Rightarrow (b_1, \delta_1) \qquad E, M_1 \vDash_{\text{e}} e_2 \Rightarrow \text{Ptr}(b_2, \delta_2) \qquad \text{typeof}(e_2) = \tau* \qquad \text{is\_valid}(\tau, \overline{M_1}, b_2, \delta_2) = n \qquad \text{store}(\text{int}, M_1, b_1, \delta_1, n) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} e_1 = \texttt{is\_valid}(e_2); \Rightarrow M_2, \overline{M_1}}$$

$$\text{E\_ISINITIALIZED:}\quad \frac{E, M_1 \vDash_{\text{lv}} e_1 \Rightarrow (b_1, \delta_1) \qquad E, M_1 \vDash_{\text{e}} e_2 \Rightarrow \text{Ptr}(b_2, \delta_2) \qquad \text{typeof}(e_2) = \tau* \qquad \text{is\_initialized}(\tau, \overline{M_1}, b_2, \delta_2) = n \qquad \text{store}(\text{int}, M_1, b_1, \delta_1, n) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} e_1 = \texttt{is\_initialized}(e_2); \Rightarrow M_2, \overline{M_1}}$$

$$\text{E\_INITIALIZE:}\quad \frac{\text{typeof}(e) = \tau* \qquad E, M_1 \vDash_{\text{e}} e \Rightarrow \text{Ptr}(b, \delta) \qquad \text{initialize}(\tau, \overline{M_1}, b, \delta) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} \texttt{initialize}(e); \Rightarrow M_1, \overline{M_2}}$$

$$\text{E\_BASEADDR:}\quad \frac{E, M_1 \vDash_{\text{lv}} e_1 \Rightarrow (b_1, \delta_1) \qquad E, M_1 \vDash_{\text{e}} e_2 \Rightarrow \text{Ptr}(b_2, \delta_2) \quad \text{typeof}(e_1) = \tau* \qquad \text{store}(\tau*, M_1, b_1, \delta_1, \text{Ptr}(b_2, 0)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} e_1 = \texttt{base\_address}(e_2); \Rightarrow M_2, \overline{M_1}}$$

$$\text{E\_BLOCKLENGTH:}\quad \frac{E, M_1 \vDash_{\text{lv}} e_1 \Rightarrow (b_1, \delta_1) \qquad E, M_1 \vDash_{\text{e}} e_2 \Rightarrow \text{Ptr}(b_2, \delta_2) \qquad \text{bounds}(\overline{M_1}, b_2) = \lfloor 0, n \rfloor \qquad \text{store}(\text{int}, M_1, b_1, \delta_2, n) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} e_1 = \texttt{block\_length}(e_2); \Rightarrow M_2, \overline{M_1}}$$

$$\text{E\_OFFSET:}\quad \frac{E, M_1 \vDash_{\text{lv}} e_1 \Rightarrow (b_1, \delta_1) \qquad E, M_1 \vDash_{\text{e}} e_2 \Rightarrow \text{Ptr}(b_2, \delta_2) \qquad \text{store}(\text{int}, M_1, b_1, \delta_1, \delta_2) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\text{s}} e_1 = \texttt{offset}(e_2); \Rightarrow M_2, \overline{M_1}}$$

Fig. 7. Semantics of destination-specific statements.

4.1 Statement Translation

The statement translation (see Figure 8) is the top-level transformation function. It simply follows the structure of the source program, only adding observation memory manipulation primitives where execution memory operations occur. Therefore, besides logical assertions, the only statements actually transformed are assignements, memory allocation, deallocation, and code blocks (to account for automatic allocation and deallocation of local variables).

When translating a logical assertion over a predicate p, a block of code is generated, ending with a C-like assertion over a local variable, $[\![p]\!]_p.res$, that will receive the result of p's translation. Its declaration is generated from its name (and, implicitly, type) using a dedicated function mkdecl. As for the code, $[\![p]\!]_p.code$, it is inserted just before the final assertion. The execution of such a block therefore follows these steps: first, the control enters the block and $[\![p]\!]_p.res$ is allocated; then $[\![p]\!]_p.code$ executes, computing p's truth value and writing the result (0 or 1) into $[\![p]\!]_p.res$; finally, the assertion is evaluated, halting the program if $[\![p]\!]_p.res$ is zero (meaning that p is false in the source program), and resuming otherwise; in the latter case, the control exits the block and $[\![p]\!]_p.res$ is automatically deallocated, returning the memory to its previous state.

4.2 Predicate Translation

The predicate translation is the main component of the program transformation as a whole. Its purpose is to convert a logical predicate into code reflecting

$[\![\texttt{skip};]\!]_s = \quad \texttt{skip};$
$[\![p = \texttt{malloc}(e);]\!]_s =$
  $p = \texttt{malloc}(e);$
  $\texttt{store\_block}(p, e);$
  $\texttt{initialize}(\&p);$
$[\![\texttt{free}(p);]\!]_s = \quad \texttt{free}(\texttt{p});$
  $\texttt{delete\_block}(p);$
$[\![l = e;]\!]_s = \quad l = e;$
  $\texttt{initialize}(\&l);$
$[\![\texttt{logical\_assert}(p);]\!]_s = \ \{$
  mkdecl($[\![p]\!]_p.res$)
  $[\![p]\!]_p.code;$
  $\texttt{assert}([\![p]\!]_p.res);\ \}$

$[\![s_1\ s_2]\!]_s = \quad [\![s_1]\!]_s\ [\![s_2]\!]_s$
$[\![\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]_s =$
  $\texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_s\ \texttt{else}\ [\![s_2]\!]_s$
$[\![\texttt{while}(e)\ s]\!]_s = \quad \texttt{while}(e)\ [\![s]\!]_s$
$[\![\{\tau_1\ x_1; \ldots \tau_n\ x_n;\ s\}]\!]_s = \ \{$
  $\tau_1\ x_1; \ldots \tau_n\ x_n;$
  $\texttt{store\_block}(\&x_1, \text{sizeof}(\tau_1));$
  $\ldots$
  $\texttt{store\_block}(\&x_n, \text{sizeof}(\tau_n));$
  $[\![s]\!]_s$
  $\texttt{delete\_block}(\&x_1);$
  $\ldots$
  $\texttt{delete\_block}(\&x_n);\ \}$

Fig. 8. Translation of statements.

$[\![\backslash\texttt{false}]\!]_p.code = \quad [\![p]\!]_p.res = 0;$
$[\![t_1 \bowtie t_2]\!]_p.code = \quad \{$
  mkdecl($[\![t_1]\!]_t.res$)
  mkdecl($[\![t_2]\!]_t.res$)
  $[\![t_1]\!]_t.code$
  $[\![t_2]\!]_t.code$
  $[\![p]\!]_p.res = [\![t_1]\!]_t.res\ \bowtie\ [\![t_2]\!]_t.res;\ \}$
$[\![p_1 \vee p_2]\!]_p.code = \quad \{$
  mkdecl($[\![p_1]\!]_p.res$)
  $[\![p_1]\!]_p.code$
  $\texttt{if}([\![p_1]\!]_p.res)\ \texttt{then}$
    $[\![p]\!]_p.res = 1;$
  $\texttt{else}\ \{$
    mkdecl($[\![p_2]\!]_p.res$)
    $[\![p_2]\!]_p.code$
    $[\![p]\!]_p.res = [\![p_2]\!]_p.res;\ \}\ \}$
$[\![\neg p]\!]_p.code =$
  $[\![p]\!]_p.code$
  $[\![p]\!]_p.res = 1 - [\![p]\!]_p.res;$
$[\![\backslash\texttt{valid}(t)]\!]_p.code = \quad \{$
  mkdecl($[\![t]\!]_t.res$)
  $[\![t]\!]_t.code$
  $[\![p]\!]_p.res = \texttt{is\_valid}([\![t]\!]_t.res);\ \}$
$[\![\backslash\texttt{initialized}(t)]\!]_p.code = \quad \{$
  mkdecl($[\![t]\!]_t.res$)
  $[\![t]\!]_t.code$
  $[\![p]\!]_p.res = \texttt{is\_initialized}([\![t]\!]_t.res);\ \}$

Fig. 9. Translation of predicates, where p denotes the currently translated predicate for short. Omitted cases are similar to those displayed.

the evaluation of this predicate. Figure 9 presents the definition of $[\![p]\!]_p.code$, inductively defined on the structure of p. Regarding the result variable ($[\![p]\!]_p.res$), we only require the transformation to generate a *fresh* name for each predicate. The *code* field of the resulting record is expected to be inserted at a program point at which its result variable, the *res* field, has already been declared and allocated with an adequate memory block.

Our translation introduces many intermediate variables (cf. Figure 9). To minimize the impact of these variables, we introduce them only when needeed, and deallocate them as soon as they are no longer used. Therefore, in all but the most simple cases (\true,\false, and $\neg p$), *code* is a block that limits the scope of the intermediate variable(s) *res*.

4.3 Term Translation

The translation function for terms (see Figure 10) is quite similar to that of predicates, the main difference being that the type of the result variable depends

$[\![e]\!]_t.code = \quad [\![t]\!]_t.res = e;$

$[\![\bar{*}t_1]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t]\!]_t.res = *[\![t_1]\!]_t.res;\ \}$

$[\![\bar{\&}t_1]\!]_t.code = \{\dots\}$ // similar to $[\![\bar{*}t_1]\!]_t$

$[\![t_1 \mathbin{\bar{\ddagger}} t_2]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
mkdecl($[\![t_2]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t_2]\!]_t.code$
$[\![t]\!]_t.res = [\![t_1]\!]_t.res \ \ddagger\ [\![t_2]\!]_t.res;\ \}$

$[\![\bar{\dagger}\, t_1]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t]\!]_t.res = \dagger\ [\![t_1]\!]_t.res\ \}$

$[\![\backslash\texttt{base\_address}(t_1)]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t]\!]_t.res = \texttt{base\_address}([\![t_1]\!]_t.res);\ \}$

$[\![\backslash\texttt{offset}(t_1)]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t]\!]_t.res = \texttt{offset}([\![t_1]\!]_t.res);\ \}$

$[\![\backslash\texttt{block\_length}(t_1)]\!]_t.code = \quad \{$
mkdecl($[\![t_1]\!]_t.res$)
$[\![t_1]\!]_t.code$
$[\![t]\!]_t.res = \texttt{block\_length}([\![t_1]\!]_t.res);\ \}$

Fig. 10. Translation of terms, where t denotes the currently translated term for short.

on the translated term, while it is always a Boolean for predicates. As with predicates, the only requirement for generated variables is freshness.

5 Soundness

Preliminary Notation Convention. Statements in the source language evaluate in some *evaluation context* $\widehat{C} = (\widehat{E}, \widehat{M})$, consisting of a variable environment $\widehat{E}$ and an execution memory state $\widehat{M}$. In the destination language, an evaluation context $\mathcal{C} = (E, M, \overline{M})$ has an additional third component: the observation memory $\overline{M}$. In both languages, statement evaluation only affects memory states, and does not alter environments. Therefore, an evaluation such as $\widehat{C_i} \vDash_{\mathrm{s}} s \Rightarrow \widehat{M_f}$ actually links the initial context $\widehat{C_i} = (\widehat{E_i}, \widehat{M_i})$ to a final context $\widehat{C_f} = (\widehat{E_f}, \widehat{M_f})$, where $\widehat{E_f} = \widehat{E_i}$. For the sake of conciseness, we assume that any memory state $\widehat{M_k}$ at some program point k is implicitly extended to a context $\widehat{C_k}$ by the current environement $\widehat{E_k}$. Reciprocally, any context $\widehat{C_k}$ may implicitly be decomposed into its components $\widehat{E_k}$ and $\widehat{M_k}$. The same holds for the destination language.

5.1 Definitions

Let us elaborate a notion of semantics preservation for our program transformation. Assume a source program s sucessfully evaluates from the initial evaluation context $\widehat{C_i}$: we have $\widehat{C_i} \vDash_{\mathrm{s}} s \Rightarrow \widehat{M_f}$. We want to relate this evaluation of s and that of its associated transformed program $[\![s]\!]_s$. The preservation property states that if the initial evaluation context of the source program $\widehat{C_i}$ and that of the transformed program $\mathcal{C}_i$ are related according to a certain relation $\mathcal{R}$, then evaluating $[\![s]\!]_s$ in $\mathcal{C}_i$ should succeed and terminate in a final context $\mathcal{C}_f$ that is also related to $\widehat{C_f}$ by $\mathcal{R}$. More formally, our transformation soundness theorem states:

$$\forall s, \widehat{C_i}, \mathcal{C}_i, \widehat{C_f}, \begin{cases} \widehat{C_i} \vDash_{\mathrm{s}} s \Rightarrow \widehat{C_f} \\ \widehat{C_i}\ \mathcal{R}\ \mathcal{C}_i \end{cases} \implies \exists\, \mathcal{C}_f, \begin{cases} \mathcal{C}_i \vDash_{\mathrm{s}} [\![s]\!]_s \Rightarrow M_f, \overline{M_f} \\ \widehat{C_f}\ \mathcal{R}\ \mathcal{C}_f \end{cases}$$

We now have to define an appropriate relation $\mathcal{R}$ between a source context $\widehat{C}$ and an associated destination context $\mathcal{C}$. They have the following differences. First, the content of the destination execution memory M is larger than its source counterpart $\widehat{M}$, because in addition to the memory of the source program, it also stores the intermediate variables introduced by the instrumentation (those generated by predicates and terms translation). M can thus be divided into two distinct regions, the original program memory M^p and the monitor memory M^m, such that no pointer value stored in M^p points to a location in M^m (because the monitored program should not refer to the memory of the monitor). We call this property *separation* and extend it to contexts.

Definition 1 (Context separation). *A context C is separated into two subcontexts C^p and C^m (denoted $C = C^p \uplus C^m$) if:*

- *E is the disjoint union of maps E^p and E^m;*
- *the set of valid blocks in M is the disjoint union of those of M^p and M^m;*
- *any valid block in M, which is also valid in either M^p or M^m, has the same content in M^p or M^m as in M;*
- *no value in M^p is a pointer to a block in M^m.*

Second, the destination context $\mathcal{C}$ includes an observation memory $\overline{M}$. The requirement for $\overline{M}$ is to be an accurate description of the monitored program memory M^p. $\overline{M}$ is then said to *represent* M^p.

Definition 2 (Representation). *An observation memory $\overline{M}$ represents an execution memory M (denoted $M \triangleright \overline{M}$) if:*

$$\begin{cases} \forall \tau, b, \delta,\ M \vDash \tau \mathbin{@} b, \delta \implies \text{is\_valid}(\tau, \overline{M}, b, \delta) = \text{true} \\ \forall \tau, b, \delta,\ \text{load}(\tau, M, b, \delta) = \lfloor v \rfloor \wedge v \neq \text{Undef} \implies \text{is\_initialized}(\tau, \overline{M}, b, \delta) = \text{true} \\ \forall b,\ \text{bounds}(M, b) = \text{bounds}(\overline{M}, b) \end{cases}$$

where $M \vDash \tau \mathbin{@} b, \delta$ means that data of type τ may be safely accessed at (b, δ) in M.

Third, in our memory model, blocks are identifiers. Therefore two memory states may have the same content up to block permutation. This *isomorphism* extends to environments and contexts.

Definition 3 (Isomorphism). *Two execution memories M_1 and M_2 are isomorphic (denoted $M_1 \sim M_2$) if there is a permutation σ on the set of blocks such that $\forall \tau, b, \delta,\ \widetilde{\sigma}(\text{load}(\tau, M_1, b, \delta)) = \text{load}(\tau, M_2, \sigma(b), \delta)$, where $\widetilde{\sigma}$ is the function over values (more precisely over value options) that applies σ to pointers: $\text{Ptr}(b, \delta) \mapsto \text{Ptr}(\sigma(b), \delta)$.*

Definition 4 (Context monitoring). *The monitoring relation $\mathcal{R}$ between a source context $\widehat{C}$ and a destination context $\mathcal{C}$ is defined as follows: $\widehat{C}\ \mathcal{R}\ \mathcal{C}$ iff $\exists\, C^p, C^m, \mathcal{C} = C^p \uplus C^m$ and $\widehat{C} \sim C^p$ and $M \triangleright \overline{M^p}$.*

5.2 Soundness Theorem

Theorem 1 (Soundness of program transformation). *Let* $\widehat{C_i} \vDash_{\mathrm{s}} s \Rightarrow \widehat{C_f}$ *be the evaluation of a source program* s*, from initial context* $\widehat{C_i}$ *to final context* $\widehat{C_f}$*, and* $\mathcal{C}_i$ *a destination context that monitors* $\widehat{C_i}$*, i.e.* $\widehat{C_i} \vDash_{\mathrm{s}} s \Rightarrow \widehat{C_f}$ *and* $\widehat{C_i}\ \mathcal{R}\ \mathcal{C}_i$*. Then* $[\![s]\!]_s$ *evaluates from* $\mathcal{C}_i$ *to a final destination context* $\mathcal{C}_f$ *that monitors* $\widehat{C_f}$*, that is,* $\exists \mathcal{C}_f,\ \mathcal{C}_i \vDash_{\mathrm{s}} [\![s]\!]_s \Rightarrow M_f, \overline{M_f}$ *and* $\widehat{C_f}\ \mathcal{R}\ \mathcal{C}_f$.

Proof. We proceed by induction on the evaluation of s. The proof is straightforward for all cases but that of `logical_assert`(), which requires a specific lemma. To give a flavor of the proof, we present the case of assignments. Throughout the proof, we manipulate many execution contexts and their components (execution and observation memories, and environments). In order to help relating them together, we index them according to the intuitive notion of program point: the initial context C_i is also C_0; after execution of an atomic statement, the next one is C_1, and so on.

Case E_ASSIGN**.** If s is an assignement $l = e$; then its translation is the sequence $l = e;$`initialize`$(\&l);$, and its evaluation is:

$$\frac{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{e}} e \Rightarrow v \qquad \mathrm{store}(\tau, \widehat{M_i}, \widehat{b}, \widehat{\delta}, v) = \lfloor \widehat{M_f} \rfloor \qquad \widehat{E}, \widehat{M_i} \vDash_{\mathrm{lv}} l \Rightarrow (\widehat{b}, \widehat{\delta})}{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{s}} l = e; \Rightarrow \widehat{M_f}}$$

We want to prove the existence of a destination evaluation context $\mathcal{C}_f$ such that $\mathcal{C}_i \vDash_{\mathrm{s}} l = e;$`initialize`$(\&l); \Rightarrow M_f, \overline{M_f}$ and $\widehat{C_f}\ \mathcal{R}\ \mathcal{C}_f$. Let us build an evaluation derivation for $[\![l = e;]\!]_s$, and then prove preservation of $\mathcal{R}$. We want to build a derivation such as this one, for appropriate values of memory states:

$$\frac{\dfrac{\mathrm{store}(\tau, M_i, b, \delta, v) = \lfloor M_1 \rfloor}{\dfrac{E, M_i \vDash_{\mathrm{e}} e \Rightarrow v \qquad E, M_i \vDash_{\mathrm{lv}} l \Rightarrow (b, \delta)}{\mathcal{C}_i \vDash_{\mathrm{s}} l = e; \Rightarrow M_1, \overline{M_1}}} \qquad \dfrac{\mathrm{initialize}(\tau, \overline{M_1}, b, \delta) = \overline{M_2}}{\mathcal{C}_1 \vDash_{\mathrm{s}} \texttt{initialize}(\&l); \Rightarrow M_2, \overline{M_2}}}{\mathcal{C}_i \vDash_{\mathrm{s}} l = e; \texttt{initialize}(\&l); \Rightarrow M_2, \overline{M_2}}$$

Since $\widehat{C_i}\ \mathcal{R}\ \mathcal{C}_i$, C_i may be separated into $C_i^p \uplus C_i^m$, with $\widehat{C_i} \sim C_i^p$. As a consequence e evaluates to the same value in C_i as in $\widehat{C_i}$: $C_i \vDash_{\mathrm{e}} e \Rightarrow v$. Now, let (b, δ) be the result of the left-value evaluation of l in the destination program: $C_i \vDash_{\mathrm{lv}} l \Rightarrow b, \delta$. Define $\lfloor M_1 \rfloor = \mathrm{store}(\tau, M_i, b, \delta)$; this store operation is valid for M_i, because the corresponding store is valid in the source memory $\widehat{M_i}$, and $\widehat{M_i}$ is isomorph to M_i^p, which is a subpart of M_i. M_1 defines a destination context $\mathcal{C}_1$; C_1 can be separated as $C_1^p \uplus C_1^m$, with $C_1^m = C_i^m$ (since the only memory operation was performed in the M^p part), and the isomorphism $\widehat{C_i} \sim C_i^p$ was preserved since the same store operation was performed in both contexts. Therefore $\widehat{C_f} \sim C_1^p$. The representation property, however, no longer holds: indeed (b, δ) now contains initialized data, but this was not reported to the observation memory $\overline{M_1} = \overline{M_i}$. Now, if we define $\overline{M_2} = \mathrm{initialize}(\tau, \overline{M_1}, b, \delta)$ and $M_2 = M_1$, the representation property is restored: $\widehat{C_2}\ \mathcal{R}\ \mathcal{C}_2$.

Case E_LOGICAL_ASSERT**.** If s is a logical assertion, the evaluation judgement is $\widehat{C_i} \vDash_{\mathrm{s}} \texttt{logical\_assert}(p); \Rightarrow \widehat{M_f}$, with premise $\widehat{C_i} \vDash_{\mathrm{p}} p \Rightarrow \text{true}$.

The generated code is: $\{\text{mkdecl}\,(\llbracket p \rrbracket_p.res); \llbracket p \rrbracket_p.code;\ \texttt{assert}(\llbracket p \rrbracket_p.res);\}$. Let C_i be an initial destination evaluation context, and C_1 the context after allocation of $\llbracket p \rrbracket_p.res$. By applying the soundness lemma 2 we get $\exists C_2$ s.t. $\mathcal{C}_1 \vDash_{\mathrm{s}} \llbracket p \rrbracket_p.code \Rightarrow M_f, \overline{M}$ and $C_2 \vDash_{\mathrm{e}} \llbracket p \rrbracket_p.res \Rightarrow \text{int}(\text{true})$. The evaluation derivation of $\llbracket p \rrbracket_p.code$ may then be completed by using the rules for C-like assertion and for code block. Preservation of $\mathcal{R}$ follows from lemma 1.

Lemma 1 (Preservation of context monitoring by predicate translation). *Let p be a predicate, $\widehat{C}$ a source context, and $\mathcal{C}_i$ and $\mathcal{C}_f$ destination contexts. If $\mathcal{C}_i \vDash_{\mathrm{s}} \llbracket p \rrbracket_p \Rightarrow M_f$ and $\widehat{C}\ \mathcal{R}\ C_i$, then $\widehat{C}\ \mathcal{R}\ C_f$.*

Proof. (sketch) The code generated by predicate translation does not modify the observation memory (it only reads from it). Moreover, since the only assignments performed in the generated code write to result variables, any modification of the execution memory takes place in the *monitoring* part of the execution memory (the M^m in the definition of $\mathcal{R}$), leaving the *program* part untouched. This ensures preservation of $\mathcal{R}$.

Lemma 2 (Soundness of predicates translation). *Let $\widehat{C} \vDash_{\mathrm{p}} p \Rightarrow \mathbf{b}$ be the evaluation of a predicate p; let C, C_i and $\overline{M}$ be such that $\widehat{C}\ \mathcal{R}\ (C, \overline{M})$ and $C_i = \text{alloc\_vars}(\llbracket p \rrbracket_p.res, C)$.*

Then $\exists C_f$ s.t. $\mathcal{C}_i \vDash_{\mathrm{s}} \llbracket p \rrbracket_p.code \Rightarrow M_f, \overline{M}$ and $C_f \vDash_{\mathrm{e}} \llbracket p \rrbracket_p.res \Rightarrow \text{int}(b)$ (where int() *is the usual encoding from boolean to integers, mapping* false *on 0 and* true *on 1).*

Proof. We prove lemma 2 by induction on p's evaluation. Base cases of the induction correspond to predicates such as validity, initialization, or term comparison; these cases are proved using a lemma expressing the soundness of *terms* translation, which is very similar to lemma 2 both conceptually and technically. Therefore, we do not prove it here. To give an intuition of the proof on other cases (logical connectives), we present one of the two cases for disjunction.

Case E_OR2**.** The considered predicate evaluation is $\widehat{C} \vDash_{\mathrm{p}} p_1 \vee p_2 \Rightarrow \mathbf{b}$, with premises $\widehat{C} \vDash_{\mathrm{p}} p_1 \Rightarrow \text{false}$ and $\widehat{C} \vDash_{\mathrm{p}} p_2 \Rightarrow \mathbf{b}$. Let us build a derivation for $\llbracket p_1 \vee p_2 \rrbracket_p.code$ (defined Figure 9). We start from context C_i as defined by the lemma's hypothesis, and build step by step every memory state the generated code is going through. Let $C_1 = \text{alloc\_vars}(\llbracket p_1 \rrbracket_p.res, C_i)$. By induction hypothesis on p_1 (instantiating C_i with C_1), there exists C_2 s.t. $\mathcal{C}_1 \vDash_{\mathrm{s}} \llbracket p_1 \rrbracket_p.code \Rightarrow M_2, \overline{M}$ and $C_2 \vDash_{\mathrm{e}} \llbracket p_1 \rrbracket_p.res \Rightarrow 0$ (since p_1 evaluates to false). Now, let $C_5 = \text{alloc\_vars}(\llbracket p_2 \rrbracket_p.res, C_2)$. By induction on p_2, there exists $C_5 = (E_5, M_6)$ s.t. $\mathcal{C}_5 \vDash_{\mathrm{s}} \llbracket p_2 \rrbracket_p.code \Rightarrow M_6, \overline{M}$ and $C_6 \vDash_{\mathrm{e}} \llbracket p_2 \rrbracket_p.res \Rightarrow \text{int}(\mathbf{b})$. Finally, let us define the following memories and associated contexts:

$$M_7 = \text{store}(\text{int}, M_6, E_6(\llbracket p \rrbracket_p.res), 0, \text{int}(\mathbf{b}))$$

$$C_8 = \text{dealloc\_vars}(\llbracket p_2 \rrbracket_p.res, C_7) \qquad C_9 = \text{dealloc\_vars}(\llbracket p_1 \rrbracket_p.res, C_8)$$

Let us prove that C_9 satisfies the expected properties for the C_f of the proof goal. Using the above definition, we can derive the following derivation for $[\![p_1 \vee p_2]\!]_p.code$ (in this derivation tree, for lack of space, the res field is abbreviated to r, $code$ to c, and $\overline{M}$ is omitted):

$$\dfrac{\dfrac{\dfrac{}{C_1 \vDash_{\mathrm{s}} [\![p_1]\!]_p.c \Rightarrow M_2} \quad \dfrac{\dfrac{}{C_2 \vDash_{\mathrm{e}} [\![p_1]\!]_p.r \Rightarrow 0} \quad \dfrac{\dfrac{\dfrac{}{C_5 \vDash_{\mathrm{s}} [\![p_2]\!]_p.c \Rightarrow M_6} \quad \dfrac{\dfrac{}{C_6 \vDash_{\mathrm{e}} [\![p_2]\!]_p.r \Rightarrow \mathrm{int}(\mathbf{b})} \quad \dfrac{}{C_6 \vDash_{\mathrm{lv}} [\![p]\!]_p.r \Rightarrow (E_6([\![p]\!]_p.r), 0)}}{C_6 \vDash_{\mathrm{s}} [\![p]\!]_p.r = [\![p_2]\!]_p.r \Rightarrow M_7}}{C_5 \vDash_{\mathrm{s}} [\![p_2]\!]_p.c; [\![p]\!]_p.r = [\![p_2]\!]_p.r \Rightarrow M_7}}{C_2 \vDash_{\mathrm{s}} else\_block \Rightarrow M_8}}{C_2 \vDash_{\mathrm{s}} \texttt{if (...) then ... else ...} \Rightarrow M_8}}{C_1 \vDash_{\mathrm{s}} [\![p_1]\!]_p.c; \texttt{if...} \Rightarrow M_8}}{C_i \vDash_{\mathrm{s}} [\![p_1 \vee p_2]\!]_p.c \Rightarrow M_9}$$

All that is left to do now is to prove $C_9 \vDash_{\mathrm{e}} [\![p]\!]_p.res \Rightarrow \mathrm{int}(b)$. This follows from the definitions of M_7, C_8 and C_9: M_7 results from storing $\mathrm{int}\, b$ at location $(E_6([\![p]\!]_p.res), 0)$ therefore $C_7 \vDash_{\mathrm{e}} [\![p]\!]_p.res \Rightarrow \mathrm{int}(v)$; since C_8 and C_9 are obtained by deallocating variables other than $[\![p]\!]_p.res$, this evaluation also holds for C_9: $C_9 \vDash_{\mathrm{e}} [\![p]\!]_p.res \Rightarrow \mathrm{int}(b)$.

6 Related Work

More and more languages include a notion of contract. Design-by-contract is one of the main features of Eiffel [22], contracts have been introduced in Java through JML [18] in 1999, in Ada 2012 [1], and the C++ standardization committee considered contracts for C++ 20, although this new feature has been finally deferred to a later standard. In Eiffel, assertions are Boolean expressions written in the programming language. In Ada 2012, it is also the case, but the language has been extended with *quantified expressions* to allow bounded universal and existential quantification. These new expressions have been inspired by Spark, a well-defined subset of Ada, extended to express contracts for static and dynamic verification.

Zhang et al. [33] studies verified runtime checking in the context of Spark: the checks to be performed are however not explicitly stated as assertions in the source language, but are implicit (e.g. division by zero). The authors provide a formalization and proofs using the Coq proof assistant [3]. Cheon [6] formalizes runtime assertion checking of JML, but provides no proof of soundess, while Lehner [19] formalizes the semantics of a large subset of JML and proves in Coq an algorithm that checks `assignable` clauses at runtime. Such clauses are memory properties that do not require memory observation. As our work focuses on memory observation, it is related but complementary to these works. Indeed, in the context of Java and Ada, even runtime checks for out-of-bounds accesses are related to arithmetic inequalities. In the case of C, however, as the bounds of an array are not attached to the array itself, out-of-bound access corresponds to an invalid access to the memory, and is therefore handled in ACSL by the

predicate `\valid`. More generally, the formal verification efforts on languages such as Eiffel, Java, Ada and Spark do not consider such properties because the design of the language prevents most memory problems that can arise in the context of C.

As runtime checking is costly, most approaches rely on an optimization phase, based on static analysis. Zhang et al. propose and verify such a phase. It is also the case for our approach and prior work [21]. Such optimizations are thus related to the verification of static analysis [14].

Our contribution targets the C language, the Frama-C framework, the ACSL specification language and the E-ACSL plug-in. In particular we focus on memory properties. In Frama-C, the plug-in RTE [11] generates ACSL assertions for runtime errors, and the E-ACSL plug-in can translate these assertions into C code. As C++ includes C, in the long term, the work presented in this paper could contribute to the verified compilation of a future standard of C++ including contracts. It is interesting to note that a recent language, Rust, that aims at combining the high-efficiency of C with strong guarantees, does not include contracts. As there is an interest in formally verifying that the type system of Rust indeed provides strong guarantees [15], that the Rust language also provides *unsafe* pointers, and there exist Rust libraries to provide rudimentary support to express contracts, our contribution may be interesting in the context of future iterations of Rust.

We aim at extending the proposed approach to consider a larger subset of E-ACSL, such as support of mathematical integers and their translation using a library such as GMP. It makes the correctness of such a library a related topic [24]. One of the strength of Frama-C is the use of the common ACSL language by all plug-ins. For the verification of RAC, it means reusing existing formalizations of ACSL designed in the context of the verification of deductive verification [10] for our extended source language. Finally, the E-ACSL plug-in currently does not support the translation of axiomatized predicates. A possible *verified* extension of E-ACSL could be based on the work of Tollitte et al. [30].

7 Conclusion

Runtime assertion checking of memory related properties for a mainstream language like C is a complex task involving various program transformation steps with additional recording of memory block metadata in a non-trivial dedicated observation memory model. This work makes a significant step toward a formally proved runtime assertion checker. We have presented a formalization of the underlying program transformation for a representative programming language with dynamic memory allocation and proved the soundness of the resulting verification verdicts. Future work includes an extension of the present proof to a real-life language like C, as well as a formalization and a mechanized proof of the runtime assertion checker in the Coq proof assistant [3].

References

1. Ada reference manual, 2012 edition, http://www.ada-auth.org/standards/ada12.html
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and Verification: The Spec# Experience. Commun. ACM (Jun 2011)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
4. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43** (2009)
5. Bruening, D., Zhao, Q.: Practical memory checking with Dr. Memory. In: Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011). pp. 213–223. IEEE Computer Society (2011)
6. Cheon, Y.: A runtime assertion checker for the Java Modeling Language. Ph.D. thesis, Iowa State University (2003)
7. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. Software Engineering Notes **31** (2006)
8. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: Formal Methods for Industrial Case Studies (FMICS). Springer (2012)
9. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Symposium on Applied Computing (SAC). ACM (2013)
10. Herms, P.: Certification of a Tool Chain for Deductive Program Verification. (Certification d'une chaine de vérification déductive de programmes). Ph.D. thesis, University of Paris-Sud, Orsay, France (2013), https://tel.archives-ouvertes.fr/tel-00789543
11. Herrmann, P., Signoles, J.: Annotation generation: Frama-C's RTE plug-in, http://frama-c.com/download/frama-c-rte-manual.pdf
12. ISO/IEC 9899:1999: Programming languages – C (1999)
13. Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a shadow, expressive as a tree: optimized memory monitoring for C. Science of Computer Programming **132** (2016)
14. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A Formally-Verified C Static Analyzer. SIGPLAN Not. **50**(1), 247–259 (2015). https://doi.org/10.1145/2775051.2676966
15. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158154
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27** (2015)
17. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: RV. LNCS, vol. 8174, pp. 328–333. Springer (2013)
18. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
19. Lehner, H.: A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking. Ph.D. thesis, ETH Zurich (2011)

20. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008), http://xavierleroy.org/publi/memory-model-journal.pdf
21. Ly, D., Kosmatov, N., Loulergue, F., Signoles, J.: Soundness of a dataflow analysis for memory monitoring. In: Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT). ACM (2018)
22. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991)
23. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: International Conference on Virtual Execution Environments (VEE 2007). pp. 65–74. ACM (2007)
24. Rieu-Helft, R., Marché, C., Melquiond, G.: How to Get an Efficient yet Verified Arbitrary-Precision Integer Library. In: Verified Software. Theories, Tools, and Experiments (VSTTE). LNCS, vol. 10712, pp. 84–101. Springer (2017). https://doi.org/10.1007/978-3-319-72308-2_6
25. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: USENIX Annual Technical Conference (USENIX). USENIX Association (2012)
26. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: USENIX Annual Technical Conference. pp. 17–30. USENIX (2005)
27. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf
28. Signoles, J., Kosmatov, N., Vorobyov, K.: 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) (2017)
29. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability: a study of field failures in operating systems. In: Fault Tolerant Computing (FTCS). IEEE (1991)
30. Tollitte, P.N., Delahaye, D., Dubois, C.: Producing certified functional code from inductive specifications. In: Certified Programs and Proofs (CPP). pp. 76–91. LNCS, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_9
31. Vorobyov, K., Kosmatov, N., Signoles, J.: Detection of security vulnerabilities in C code using runtime verification: an experience report. In: Tests and Proofs (TAP). Springer (2018)
32. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: International Sympoisum on Memory Management (ISMM). ACM (2017)
33. Zhang, Z., Robby, Hatcliff, J., Moy, Y., Courtieu, P.: Focused Certification of an Industrial Compilation and Static Verification Toolchain. In: Software Engineering and Formal Methods (SEFM). LNCS, vol. 10469, pp. 17–34. Springer (2017). https://doi.org/10.1007/978-3-319-66197-1_2