Sound Runtime Assertion Checking for Memory Properties via Program Transformation

DARA LY, Université Paris-Saclay, CEA, List, France
NIKOLAI KOSMATOV, Université Paris-Saclay, CEA, List & Thales Research and Technology, France
FRÉDÉRIC LOULERGUE, Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
JULIEN SIGNOLES, Université Paris-Saclay, CEA, List, France

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 pointers and memory operations, including dynamic allocation and deallocation. The generated program instrumentation relies on an axiomatized observation memory model, which is essential to record and monitor memory-related properties. We prove the soundness of RAC verdicts with regard to the semantics of this language.

ACM Reference Format:
Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. 2023. Sound Runtime Assertion Checking for Memory Properties via Program Transformation. 1, 1 (June 2023), 46 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Runtime assertion checking (RAC) [13] 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 numerous cases of errors, typically *undefined behavior* in C [21]. Indeed, such languages, C in particular, are still widely used, e.g. in embedded software, while memory-related errors, such as invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, are among the most frequent software errors [48].

Recent tools addressing memory safety of C programs at runtime, such as Valgrind and MemCheck [36, 42], DrMemory [10] or AddressSanitizer [41], have become very popular and successful in detecting bugs when the program is running. 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 [49], such as redzoning [41]. While for a reported bug, it can be possible—at least, in

Authors' addresses: Dara Ly, Université Paris-Saclay, CEA, List, Palaiseau, France, contact@libellules.eu; Nikolai Kosmatov, Université Paris-Saclay, CEA, List & Thales Research and Technology, Palaiseau, France, nikolaikosmatov@gmail.com; Frédéric Loulergue, Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France, frederic.loulergue@univ-orleans.fr; Julien Signoles, Université Paris-Saclay, CEA, List, Palaiseau, France, julien.signoles@cea.fr.

XXXX-XXXX/2023/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. Often applied in complement to sound static verification techniques, RAC is used to check the absence of failures on parts of annotated code which were not (yet) proved [33]. Hence, ensuring the soundness of RAC tools is crucial. E-ACSL[1] is one of these tools [46], as part of the Frama-C verification platform [3] for static and dynamic analyses of C programs. A *formal proof of soundness* for E-ACSL 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* [22, 26, 50]—and to evaluate them thanks to this record. In this context, the proof of soundness is highly non-trivial: it requires one to formalize not only the semantics of the considered programming and specification languages, but also the program transformation and the observation memory model. Following Flanagan and Saxe [17], the soundness of the instrumentation for RAC can be intuitively stated as follows: the instrumentation is sound if an assertion in the original program fails (or "goes wrong" in the terminology of Flanagan and Saxe) if and only if that assertion evaluated in the instrumented program also fails.

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 pointers and dynamic memory allocation supplemented by a specification language with a complete set of memory-related predicates, including pointer validity, variable initialization, as well as memory-related functions for pointer offset, base address and size of memory blocks. Such memory properties are referred to as *block-level* memory properties [50] since they allow the user to express memory properties related to memory block boundaries. We define their semantics and formalize a runtime assertion checker for these languages, including the underlying program transformation and observation memory model. To allow more freedom for an implementation, we define the execution and the observation memory models axiomatically. In order to exclude any risk of inconsistency in the axioms, we formalize them and prove their consistency with respect to a simple implementation using the Coq proof assistant [7]. The Coq development also includes key notions (isomorphisms of contexts, subcontexts, representation of an execution memory by an observation memory) and the proof of their key properties used in the paper. Finally, we state and prove the soundness result ensuring that the resulting verdicts are correct with respect to the semantics. This paper is an extended version of an initial conference paper [31], which was deeply reworked in this version, in particular, enriched with a rigorous formalization of the execution and observation memory models and their properties in order to provide a more complete proof of the required results.

The contributions of the paper include:

- a formalization of all major steps of a runtime assertion checker for a representative imperative language with assertions;
- a definition of an observation memory model, suitable for a modular definition and verification of program transformations injecting non-interfering code for observing and monitoring block-level memory properties;
- a formalization[2] and proof of consistency of the proposed axiomatic definitions of the execution and the observation memory models and their key properties in Coq;
- a proof of soundness of a runtime verifier for block-level memory properties.

[1] available as open-source software at https://frama-c.com/eacsl.html
[2] available at https://frederic.loulergue.eu/ftp/tap_si_fac2022.zip

Outline. Section 2 gives an overview of the work and a motivating example. Section 3 defines the source language, including an assertion statement and a language of predicates, and its semantics. The target language and the transformation generating the runtime assertion checking code 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 verification). 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.

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 on line 5, written in the E-ACSL specification language [14, 43], stating that `t+mid` of type `int*` refers to a "valid memory location", ensuring that it is safe to dereference it on lines 6 and 7. For this program, the assertion is satisfied and runtime assertion checking of this program with the E-ACSL tool will not detect any failure.

To illustrate a failure, let us assume that `search` is called on line 15 with an erroneous length argument, say, 10 instead of 5. Then during the first iteration of the loop, `mid` would take the value 5 (on line 4) and the assertion on line 5 would fail because `t + 5` is out of `t`'s bounds (as `t` has only 5 elements as defined on line 14). In this case, runtime assertion checking of this program with the E-ACSL tool would halt the program execution and report the failure.

```
1  int search(int *t, int len, int x) { // search x in array t
2    int lo = 0, hi = len - 1;          // initial interval bounds
3    while (lo <= hi) {                 // while 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, 5, 7);
16 }
```

Fig. 1. An example C program with ACSL annotations.

Checking such a property at runtime is not trivial: in particular, it requires us to know at the program point of the annotation (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 [46] is 111-lines long (when deactivating its static optimization described in [32]) for tracking the program memory manipulation. In particular, for the block `t` created and initialized on line 14, E-ACSL adds the following lines of code (assuming that `sizeof(int)` = 4, so `t` is 20-byte long):

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

Each of the cells of the array is then marked as initialized as well.

Optimized implementations of such functions are also pretty complex, as explained by Vorobyov et al. [50]. In this work, assuming their correct implementation, we *formalize* the whole instrumentation performed by an 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 (theorem 5.7, semantic preservation).

3 SOURCE LANGUAGE

This section presents the source language: its syntax and semantics, including the formal model of memory we consider.

3.1 Motivation and Overview of the Main Components

From a program written in C, associated with a formal specification given as ACSL annotations (which are C comments), the code generator of the E-ACSL plugin inserts instructions that implement a monitor, whose role is to verify, at runtime, the validity of the annotations. It can be seen as a transformation from a C program to another program *written in the same language*.

The features of this monitor are prescribed by the ACSL annotations of the source program. From this perspective, the E-ACSL tool can also be seen as translating a *source* language into a distinct *target* language. From this point of view, the source language is the C language extended with ACSL annotations. A programmer using this language can for instance write a program such as the code of Fig. 1 that includes a formal E-ACSL annotation to check the validity of the access to an array cell. These programs are then translated into "pure" C where the annotations are removed. C without annotations is the target language of the translation mechanism. It should include an assert instruction to check properties. The translation of the program of Fig. 1 gives a C program where the annotation `assert` is replaced by C instructions that control the validity of the memory access (using the assert instruction). The assert annotation is left in the code but only for documentation. In the target code, it is a simple comment without any meaning for the program execution.

Such an approach has two advantages. First, considering annotations as part of the source language makes the semantic analysis of the transformation easier.

Second, it allows us to abstract away some aspects of the instrumentation. Indeed, in the E-ACSL plugin, the generated online monitors call an auxiliary library, also written in C. This library provides data structures and functions that are necessary to implement the checks corresponding

to the source annotations. Modeling the problem with a unique language would make the formal reasoning on the implementation of this library mandatory. A distinct target language can be enriched with this library API without considering its implementation but only its semantics and its properties.

In terms of programming languages, we therefore have to model three parts: a common core language representing the C language, a source language obtained by adding a logical specification language to this core language, and a target language obtained by adding instructions coming from the E-ACSL runtime library to the core language.

The focus of this work is the verification of assertions about properties related to memory states (also called memory properties). The core language should therefore be representative of the C language in terms of memory manipulation. The specification language should also be able to express interesting memory properties.

3.2 Core Imperative Language

We denote by X the set of (names of) variables possibly used in our programs. The essence of an imperative language such as C is captured by a handful of constructions, often referred to as While language [37]. In such a language, expressions and statements (or instructions) are two distinct syntactic categories. An expression can be an integer constant, a variable, the application of a unary operation to an expression, or the application of a binary operation to two expressions: these are the first four cases for expressions in Fig. 2. It is worth noting that expressions are side-effect free. Indeed, side effects are only allowed in statements, which include the instruction without any effect (`skip`), the assignment, the sequence, the conditional instruction, and the while loop. These statements are the first five cases for statements in Fig. 2.

To this core language, we add memory related operations found in C: obtaining the address of an object and deferencing a pointer. To be able to use pointer arithmetic, the set of binary operators of the language must include operators + and − that, resp., adds/subtracts an integer to/from a pointer. This completes the grammar of expressions given in Fig. 2.

One key mechanism for managing memory is dynamic allocation and deallocation. The function `malloc` allocates a block with the number of memory cells given as argument, and if it succeeds it returns a pointer to this memory block. The deallocation function `free` takes as argument a dynamically allocated pointer and deallocates the considered memory block. Our language does not offer functions,[3] therefore we model these two specific functions as two new instructions, where the specific integrated notation used for `malloc` also stores the resulting pointer in a given memory location.

In addition to explicit memory allocation in the heap, C provides a simple form of automatic memory management: declaration of local variables inside blocks. We also provide a similar feature (but without blocks) in the form of a `let` instruction. The instruction `let` $x : \tau$ `in` s `end` allocates a memory block for variable x of type τ without initializing it, executes instruction s and finally deallocates the memory block that was allocated for x. The same instruction `let`—put at the outer level and containing other program instructions inside it—can be used to model global variables, so we do not introduce other means to declare global variables. We assume without loss of generality that variables are never overloaded, that is, a `let` instruction never introduces a variable x if x is already visible in the scope (as a variable previously introduced by an external `let` instruction). This requirement can be easily satisfied by renaming of variables (also known as α-conversion). We

[3]Notice that the fact to exclude function calls in this work is a simplification assumption of the formalization and not an essential limitation inherent to the transformation approach. Indeed, the E-ACSL tool perfectly supports functions calls.

expr	$e ::=$	n	integer constant
	$\mid$	x	variable
	$\mid$	$\dagger\ e$	unary operator
	$\mid$	$e\ \ddagger\ e$	binary operator
	$\mid$	$*e$	dereferencing
	$\mid$	$\&e$	address
term	$t ::=$	e	expression
	$\mid$	$\bar{*}t$	dereferencing
	$\mid$	$\bar{\dagger}\ t$	unary operator
	$\mid$	$t\ \bar{\ddagger}\ t$	binary operator
	$\mid$	$\texttt{\textbackslash base\_address}(t)$	base address
	$\mid$	$\texttt{\textbackslash offset}(t)$	offset of a pointer
	$\mid$	$\texttt{\textbackslash block\_length}(t)$	length of a memory block
pred	$p ::=$	$\texttt{\textbackslash true} \mid \texttt{\textbackslash false}$	true, false
	$\mid$	$t \bowtie t$	comparison
	$\mid$	$p \wedge p$	conjunction
	$\mid$	$\neg p$	negation
	$\mid$	$\texttt{\textbackslash valid}(t)$	pointer validity
	$\mid$	$\texttt{\textbackslash initialized}(t)$	initialisation
stmt	$s ::=$	$\texttt{skip};$	no effect
	$\mid$	$e = e;$	assignment
	$\mid$	$s\ s$	sequence
	$\mid$	$\texttt{if}(e)\ \texttt{then}\ s\ \texttt{else}\ s$	conditionnal
	$\mid$	$\texttt{while}(e)\ s$	loop
	$\mid$	$e = \texttt{malloc}(e);$	allocation
	$\mid$	$\texttt{free}(e);$	deallocation
	$\mid$	$\texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end}$	local variable
	$\mid$	$\texttt{logical\_assert}(p);$	logical assertion
ctyp	$\tau ::=$	$\texttt{int}k,\ k \in \{8, 16, 32, 64\}$	integer type
	$\mid$	$\tau*$	pointer type

Fig. 2. Syntax of the source language. The type associated to each syntatic category is in bold.

also assume that special variables $\texttt{res}(n)$ (for all $n \geqslant 0$) are never introduced in our source programs: these variable names are reserved for auxiliary variables introduced by the translation.

The possible types in our language are basically integers of various sizes and a pointer type. The size of integers varies from 8-bit to 64-bit integers to make things easier while reasoning, without loss of generality: in E-ACSL in particular, the machine-dependent parameters such as the size of integers can easily be set by the users. For the sake of conciseness, we choose to avoid defining a static type system. Instead, we assume that each expression is labeled with a type and we define program evaluation in such a way that it takes into account type information. We write $e : \tau$ to denote that expression e is annotated with type τ. The set of types is denoted by **type**. We often omit to indicate types in the paper when they are clear from the context.

```
let t: int64*  in
let len: int64 in
let x: int64   in
  t = malloc (len * sizeof(int64));
  *t = -3; *(t+1) = 2; *(t+2) = 4; *(t+3) = 7; *(t+4) = 10;
  len = 5;
  x = 7;

  let lo: int64  in
  let hi: int64  in
  let idx: int64 in
    idx = -1;
    lo = 0;
    hi = len - 1;
    while (lo <= hi)
    let mid: int64 in
      mid = lo + (hi - lo) / 2;
      logical_assert(\valid(t + mid));
      if (*(t + mid) == x) then
        idx = mid;
        lo = hi + 1;
      else if (*(t + mid) < x) then
        lo = mid + 1;
      else
        hi = mid - 1;
    end
  end
  end
  end
end
end
end
```

Fig. 3. The example of Fig. 1 written in our source language.

3.3 Specification Language

An assertion is formalized as a statement including a predicate in a dedicated logic, which expresses a property of the program execution. If such a predicate is false, the execution stops. If it is true, the assertion has no effect and the execution continues.

The variety of properties the specification language can express depends on the language of predicates. In this work we focus on memory properties and more specifically on *spatial* properties [35], i.e. properties related to the memory state at a given time, for example, validity of a memory access to a memory location. We do not consider *temporal* properties [13], which characterize errors related to sequences of events such as some cases of use-after-free, when a pointer is used while the memory it points to was deallocated and possibly another block was allocated at the same location. Our memory model assumptions described below (in particular, the freshness condition of block allocation) will not allow modeling such situations.

Our langage of predicates is a propositional logic. For the sake of conciseness we consider only negation, conjunction and constants `\true` and `\false`. The conjunction operator is a short-circuit operator that evaluates its right-hand operand only if its left-hand operand is true. The other propositional logic operators, such as disjunction and implication, can be encoded within this minimal subset. Predicates are defined on *terms*, which represent objects related to program execution, in particular memory. We assume the usual relational predicates for comparing terms.

Two specific predicates `\valid`() and `\initialized`() express properties on pointers. Their meaning is similar to the same predicates in ACSL. If t is a term representing a pointer, `\valid`(t) expresses the validity of the pointer, i.e. the ability to deference the pointer and access the pointed location without error. `\initialized`(t) means that the memory location referred to by pointer t has been initialized, i.e. a compatible value was written with the same type at the same location. We adopt a strict version of initialization: a pointed location is seen as initialized only if a defined value can be read from this location.

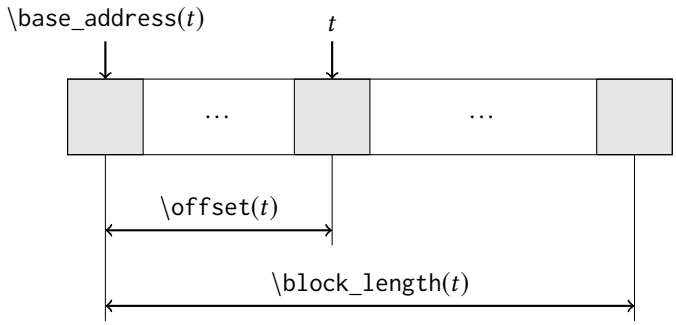

Fig. 4. Block properties.

Terms are a generalization of the source language expressions with a few additional cases needed for the predicates. We add three logic functions, similar to those existing in ACSL, which describe the memory state as depicted in Fig. 4: `\block_length`(), `\base_address`() and `\offset`(). `\block_length`(t) is the size (in bytes) of the memory block containing pointer t, `\base_address`(t) is the pointer to (the beginning of) the memory block containing pointer t, and `\offset`(t) is the offset (in bytes) of pointer t with respect to the beginning of the block. These logic functions are powerful enough to specify byte-level memory operations which typically occur in real C programs. The unary and binary operators of expressions are extended to terms and written $\dagger$ and $\ddagger$, so that it is possible to write terms such as `\block_length`(p) + 1 for some pointer p.

Figure 3 shows the example of Fig. 1 written in our source language.

3.4 Execution Memory Model

Several important statements and expressions are related to (execution) memory. The semantics of our language depends on how the interaction of these instructions with memory is described. Memory should therefore be formalized too: this is the purpose of the *(execution) memory model*. Execution memory states will be denoted by M, M', M_1, M_2,...

Basically, memory can be seen as a data structure which can be used to store data at some locations using a write instruction (store). The data can later be retrieved by a read instruction (load). Memory space is a resource: a memory block should be requested before being used. This is the role of the allocation operation (alloc). A memory block can be released using a deallocation operation (free). To remain generic, instead of providing a concrete implementation of such a data

value	$v ::=$	$\mathsf{Int}(n)$	$M_\emptyset \in$	**mem**
	$\mid$	$\mathsf{Ptr}(b, \delta)$	alloc :	$\mathbf{mem} \times \mathbb{N} \to \mathbf{block} \times \mathbf{mem}$
	$\mid$	Undef	free :	$\mathbf{mem} \times \mathbf{block} \to \mathbf{option\ mem}$
			store :	$\mathbf{mtyp} \times \mathbf{mem} \times \mathbf{block} \times \mathbb{Z} \times \mathbf{value} \to \mathbf{option\ mem}$
mtyp	$\kappa ::=$	i 8	load :	$\mathbf{mtyp} \times \mathbf{mem} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{option\ value}$
	$\mid$	i 16		
	$\mid$	i 32	$\cdot \vDash \cdot$:	$\mathbf{mem} \times \mathbf{block} \to \mathbf{bool}$
	$\mid$	i 64	length :	$\mathbf{mem} \times \mathbf{block} \to \mathbb{N}$

Fig. 5. Values and memory types.

Fig. 6. Operations of the execution memory model.

structure, we formalize it as an algebraic specification. It makes our results applicable to various implementations, as soon as they satisfy this specification.

The memory model of our source language is an adaptation of the first version of CompCert's [29] memory model [30]. The block structure of this model fits the concept of block in E-ACSL specifications. Moreover, the CompCert model was designed in the context of a C compiler, and our language can be seen as a subset of C.

In this model, at every moment of program execution, the program memory (state) consists of a set of memory blocks. Each block has a size and a unique identifier. A byte in the memory is identified by the block it belongs to and an offset being the index of this byte in the block. The set of memory states is denoted by **mem**, and **block** denotes the set of block identifiers. The predicate $\vDash$ defines the validity of a block in a given memory, while the function length gives its size. The signatures of all memory operations are given in Fig. 6. We assume we always have sufficient memory, so allocation cannot fail, while all other operations may fail. That is why they return a value of type **option t**, which adds a specific value for a failing operation. The returned value is ε in case of a failure, and $\lfloor v \rfloor$ where v has type **t** in case of success.

The initial memory of CompCert was not specified for many cases that cannot occur within the operational semantics of CompCert languages. This is not the case here: we want to express a relationship between the execution memory and the observation memory. The observation memory stores information that is more detailed that what is needed in CompCert. Therefore our execution memory is less abstract and its behaviour is specified in more cases than the CompCert memory model. This is independent of the operational semantics of our languages. Notice that many of these additional cases (such as badly typed accesses or badly aligned overlapping accesses) cannot occur for our programs due to the type and alignment constraints in our languages and semantics (further discussed below). We make nevertheless the choice to define memory models in a more general way, suitable for future extensions of the considered languages.

Data is stored as *values*. A value can be an integer, a pointer (block identifier and offset), or the meaningless value Undef, which is a default value for an uninitialized memory cell. Each of these types has a size expressed in multiples of one byte. Accessing these values is thus parametrized by a *(physical) memory type*, which basically gives only the number of bytes to read or write (contrary to the types of our source language, which distinguish integers and pointers, cf. Fig. 2). The set of *memory types*, denoted by **mtyp**, is defined in Fig. 5. We denote by $\mathrm{sizeof}(\kappa)$ the byte size of a memory type κ.

For the sake of clarity in the operational semantics, we define kind, a function from **value** ∪ **type** to the set {Num, Ptr, Undef}, as follows:

$$\begin{cases} \mathsf{kind}(\mathtt{int}k) & \triangleq \ \mathtt{Num} \text{ for } k \in \{8, 16, 32, 64\} \\ \mathsf{kind}(\tau*) & \triangleq \ \mathtt{Ptr} \text{ for } \tau \in \textbf{type} \\ \mathsf{kind}(\mathsf{Int}(n)) & \triangleq \ \mathtt{Num} \\ \mathsf{kind}(\mathsf{Ptr}(b, \delta)) & \triangleq \ \mathtt{Ptr} \\ \mathsf{kind}(\mathsf{Int}(\mathsf{Undef})) & \triangleq \ \mathtt{Undef} \end{cases}$$

The memory type of an expression type is obtained by an application of the function mtype defined as follows:

$$\begin{array}{lll} \mathsf{mtype}(\mathtt{int}k) & \triangleq & \mathrm{i}\,k \quad \text{for } k \in \{8, 16, 32, 64\}, \\ \mathsf{mtype}(\tau*) & \triangleq & \mathrm{i}\,w, \quad \text{where } w \text{ is the size of a memory word.} \end{array}$$

Definition 3.1 (Valid access). We say that an access to location (b, δ) with memory type κ in memory state M is valid, denoted by $M \vDash \kappa \mathbin{@} b, \delta$, if

$$M \vDash b \quad \wedge \quad \delta \geqslant 0 \quad \wedge \quad \delta + \mathsf{sizeof}(\kappa) \leqslant \mathsf{length}(M, b).$$

There exists an empty execution memory $M_\emptyset$ (that will be used as an initial execution memory state for evaluation of our programs) that contains no valid blocks. When a new block b is allocated in a memory M, it should respect a freshness property. This has two aspects. First, we should obviously exclude possible reuse of the identifier of an already allocated valid block. Second, we should also exclude possible reuse of the identifier of a block that is currently not valid, but was previously allocated and then deallocated while some location in this block is still referred to by an existing pointer (called a *dangling pointer* in this case). For this purpose, it will be practical to define the support of a memory.

Definition 3.2 (Support). The *support* $\mathsf{supp}(M)$ of a memory M is the set of (identifiers of) blocks that are valid in M or referred to by a pointer value in M. In other words,

$$\mathsf{supp}(M) = \{b \in \textbf{block} \mid M \vDash b\} \cup \{b \in \textbf{block} \mid \mathsf{load}(\kappa, M, b', \delta') = \lfloor \mathsf{Ptr}(b, \delta) \rfloor \text{ for some } b, b', \delta, \delta', \kappa\}.$$

The freshness condition on a new block b allocated in memory M (that will be required in Axiom (16) below) can be stated as $b \notin \mathsf{supp}(M)$. This condition—depending on the contents of the memory—can appear surprising at first glance. An alternative approach is to maintain the set of all previously allocated blocks and to require that a previously allocated block identifier is never reused again for a new allocated block. It will also exclude the risk for a dangling pointer to become valid again after such an allocation, but is stronger than necessary. Our approach is more generic: it requires a weaker but sufficient condition that does not forbid a concrete implementation to reuse a block identifier for an allocated block when there is no risk.

The equational theory of memory operations is given as the following set of axioms. For the sake of conciseness, we consider that all free variables in these axioms are universally quantified.

Definition 3.3 (Execution memory model). The execution memory model is defined by the axioms presented in Fig 7.

The axioms belong to four categories. Axioms (1), (2), (3), (4), (5) formalize the validity of blocks. A valid block is introduced by its allocation and removed by its deallocation, while other operations do not change valid blocks. Axioms (6), (7), (8), (9), (10), (11) describe the contents of the memory, basically the effect of other operations on read operations. An undefined value is read at a valid access in a newly allocated block (Axiom (6)), while an allocation and a deallocation do not change the contents of other blocks (Axiom (7), (8)). When reading a value after writing a value, the read

$$\mathsf{alloc}(M_1, n) = (b, M_2) \implies M_2 \vDash b \tag{1}$$
$$b \neq b' \wedge \mathsf{alloc}(M_1, n) = (b, M_2) \implies (M_2 \vDash b' \iff M_1 \vDash b') \tag{2}$$
$$\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \implies M_2 \nvDash b \tag{3}$$
$$b \neq b' \wedge \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \implies (M_2 \vDash b' \iff M_1 \vDash b') \tag{4}$$
$$\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \implies (M_2 \vDash b' \iff M_1 \vDash b') \tag{5}$$

$$\left.\begin{array}{l}\mathsf{alloc}(M_1, n) = (b, M_2) \wedge \\ \delta \geqslant 0 \ \wedge \ \delta + \mathsf{sizeof}(\kappa) \leqslant n\end{array}\right\} \implies \mathsf{load}(\kappa, M_2, b, \delta) = \lfloor \mathsf{Undef} \rfloor \tag{6}$$
$$b \neq b' \wedge \mathsf{alloc}(M_1, n) = (b, M_2) \implies \mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta) \tag{7}$$
$$b \neq b' \wedge \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \implies \mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta) \tag{8}$$
$$\left.\begin{array}{l}\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \wedge \\ \delta \geqslant 0 \ \wedge \ \delta + \mathsf{sizeof}(\kappa') \leqslant \mathsf{length}(M_2, b)\end{array}\right\} \implies \mathsf{load}(\kappa', M_2, b, \delta) = \lfloor \mathsf{convert}(v, \kappa, \kappa') \rfloor \tag{9}$$
$$\left.\begin{array}{l}\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \ \wedge \delta' \neq \delta \ \wedge \\ \delta + \mathsf{sizeof}(\kappa) > \delta' \wedge \delta' + \mathsf{sizeof}(\kappa') > \delta \ \wedge \\ \delta' \geqslant 0 \ \wedge \ \delta' + \mathsf{sizeof}(\kappa') \leqslant \mathsf{length}(M_2, b)\end{array}\right\} \implies \mathsf{load}(\kappa', M_2, b, \delta') = \lfloor \mathsf{Undef} \rfloor \tag{10}$$
$$\left.\begin{array}{l}\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \wedge \big(b' \neq b \vee \\ \delta + \mathsf{sizeof}(\kappa) \leqslant \delta' \vee \delta' + \mathsf{sizeof}(\kappa') \leqslant \delta\big)\end{array}\right\} \implies \mathsf{load}(\kappa', M_2, b', \delta') = \mathsf{load}(\kappa', M_1, b', \delta') \tag{11}$$

$$\mathsf{alloc}(M_1, n) = (b, M_2) \implies \mathsf{length}(M_2, b) = n \tag{12}$$
$$b \neq b' \wedge \mathsf{alloc}(M_1, n) = (b, M_2) \implies \mathsf{length}(M_2, b') = \mathsf{length}(M_1, b') \tag{13}$$
$$\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \implies \mathsf{length}(M_2, b') = \mathsf{length}(M_1, b') \tag{14}$$
$$b \neq b' \wedge \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \implies \mathsf{length}(M_2, b') = \mathsf{length}(M_1, b') \tag{15}$$

$$\mathsf{alloc}(M_1, n) = (b, M_2) \implies b \notin \mathsf{supp}(M_1) \tag{16}$$
$$M_1 \vDash \kappa \,@\, b, \delta \iff \exists M_2,\ \mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \tag{17}$$
$$M \vDash \kappa \,@\, b, \delta \iff \exists v,\ \mathsf{load}(\kappa, M, b, \delta) = \lfloor v \rfloor \tag{18}$$
$$M_1 \vDash b \iff \exists M_2, \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \tag{19}$$
$$M_\emptyset \nvDash b \tag{20}$$

Fig. 7. Axioms of execution memory model.

value remains unchanged if reading is performed at a non-overlapping location (Axiom (11)); becomes undefined if reading is performed at an overlapping valid access (Axiom (10)) or with incompatible type (Axiom (9)); and is equal to the written value if reading is at the same location with the same type (Axiom (9)). The convert operation and Axiom (9) are further detailed below. The third set of axioms, with Axioms (12), (13), (14), (15), describes the length of blocks. It is determined by the size requested during allocation, and remains unchanged by other operations. Notice that the length is not specified for a block that has not been allocated (an implementation can choose, for instance, an arbitrary default value for it). Next, Axioms (16), (17), (18), (19) specify the freshness condition and conditions of a successful read, write and free operations. In particular, reading and writing operations succeed (i.e. return a defined value or Undef) if and only if they are performed at a valid access, while a deallocation succeeds if and only if it is performed on a

valid block. Finally, Axiom 20 states that empty execution memory $M_\emptyset$ contains no valid blocks. In particular, in memory $M_\emptyset$, all deallocation, writing and reading operations fail (that is, return ε).

When writing a value to a memory, in general, the value v and the memory type κ_w used for writing may not correspond. Further, when a value is read at this location, the memory type used for reading κ_r may not correspond again. Axiom (9) deals with this situation using a function convert : **value** × **mtyp** × **mtyp** → **value**.

$$\mathsf{convert}(v,\ \kappa_w,\ \kappa_r) \triangleq \begin{cases} \mathsf{Undef} & \text{if } \kappa_w \neq \kappa_r \text{ or } v \text{ is not } \kappa_w\text{-storable,} \\ v & \text{if } \kappa_w = \kappa_r \text{ and } v \text{ is } \kappa_w\text{-storable.} \end{cases}$$

For example $\mathsf{Int}(n)$ is i8-storable if $-128 \leq n \leq 127$.

To sum up, the only way to read a defined value from memory is when this value was previously written into the same valid memory location, and not (fully or partially) overwritten since, the block was not deallocated, and the type of the written value corresponds to the memory type of the write and read operations. For simplicity, in this paper we do not consider more complex cases (of casts between integer types for which the read value in the C standard may be implementation-dependent). These extensions are left for future work.

Notice that Undef and ε should not be confused. A load operation returns ε when the given access is invalid, and some value $\lfloor v \rfloor$ when the access is valid. In the last case, the value v can be Undef (e.g. unintialized), so in some of the evaluation rules below, an additional check of the form $v \neq \mathsf{Undef}$ will be used to eliminate this case.

In order to exclude any risk of inconsistency in the axioms of the execution memory model, we formalize them and prove their consistency with respect to a simple implementation using the Coq proof assistant [7]. We implement the signatures of Fig. 6 and the axioms of Fig. 7 in a module *type* (see file `ExecutionMemoryModel.v` in our Coq formalization). To check the consistency we implement a module that respects this module type by giving definitions for the functions and proving the axioms as lemmas (see file `ExecutionMemoryImplementation.v`).

As mentioned above, Axiom (10) and the case returning Undef in Axiom (9) do not actually occur during the execution of our programs with the considered operational semantics because of well-typedness and well-alignedness of memory accesses.

3.5 Operational Semantics

The semantics of our source language is a big-step semantics adapted from CompCert's semantics [9]. We use an error-free blocking semantics: the evaluation cannot proceed in case of an error. We have five evaluation relations, all captured by the following schema:

$$\text{context} \ \vDash_{\text{mode}} \ \text{syntactic object} \ \Downarrow \ \text{result}.$$

In the source language, an *(evaluation) context* $C = (E, M)$ is a pair consisting of an environement E and a memory state M. When there is no risk of ambiguity, parentheses in (E, M) will be omitted. An axiomatic definition of memory states and operations was given in Section 3.4. An *environement* E is a partial injective mapping $E : V \to \textbf{block}$ which maps variable names to block identifiers. It is used to identify in which memory block the value of a given variable x can be found. More precisely, for a variable name x, if there exists a block b containing x, then E is defined for x and we write $E(x) = \lfloor b \rfloor$. Otherwise, we write $E(x) = \epsilon$. The subset of variables that are mapped by environment E to a block is its *domain* $\mathsf{dom}(E) = \{x \in X \mid E(x) = \lfloor b \rfloor \text{ for some } b \in \textbf{block}\}$. We also define the *image* $\mathsf{im}(E)$ of E: $\mathsf{im}(E) = \{b \in \textbf{block} \mid E(x) = \lfloor b \rfloor \text{ for some } x \in X\}$. Environements will be denoted by E, E', E_1, E_2,... We denote by $E_\emptyset$ the empty environment that maps no variable, in other words, $\mathsf{dom}(E_\emptyset) = \emptyset$. The empty context $(E_\emptyset, M_\emptyset)$ will be used as an initial evaluation

mode	type of the evaluated object	result type
expression	**expr**	**value**
left-value	**expr**	**block** $\times \mathbb{Z}$
term	**term**	**value**
predicate	**pred**	**bool**
instruction	**stmt**	**mem**

Fig. 8. Evaluation modes of the source language semantics.

context for the evaluation of our programs. Dynamically allocated blocks are not associated with a variable name in the environment and can be addressed only through a pointer to the block.

A context $C = (E, M)$ is *well-formed* if for any block b associated with a variable x by environment E, the block b is valid in M, in other words, $\forall x, b, E(x) = \lfloor b \rfloor \implies M \vDash b$. We always assume that all contexts we manipulate are well-formed. It is obvious for the empty context $(E_\emptyset, M_\emptyset)$. It can be easily checked that all our rules—in particular, adding a variable or removing a block—will preserve this property.

The syntactic objects and the result type depend on the evaluation *mode* as specified in Fig. 8 where **expr**, **term**, **pred** and **stmt** denote respectively the types of expressions, terms, predicates and instructions.

The evaluation of an expression e to a value v in a context E, M is written $E, M \vDash_{\mathrm{e}} e \Downarrow v$. Its evaluation in the same context but as a *left-value* yields a memory location (b, δ) and is written $E, M \vDash_{\mathrm{lv}} e \Downarrow b, \delta$. Like in C, left-values are expressions that describe a concrete memory location and can therefore appear on the left side of an assignment. Both relations are defined in Fig. 9.

Rule E-LVAL shows how the memory model is used. For an expression e evaluated as a left-value to a location (b, δ), we perform a memory read. If the read operation successfully returns a value v of the expected kind (a pointer if e has a pointer type or a number if e has a number type) and *this value is not undefined*, then e is correctly evaluated to v. In other cases, it cannot be evaluated. This illustrates an error-free blocking semantics: the evaluation cannot proceed in case of an error. Rule E-PARITH specifically considers pointer arithmetic and preserves well-aligned pointers.

Figure 10 shows the evaluation rules for terms. Terms are a generalization of expressions. Therefore terms which are indeed expressions are evaluated in the exact same way (rule T-EXPR).

Since terms are side-effect free, there is no rule for evaluating a term as a left-value. Therefore, the rule for dereferencing a term is different from its expression counterpart. Rule T-DEREF combines the evaluation of a term directly as a pointer and reading the memory at the location referred to by this pointer.

Evaluating the base address of a pointer (rule T-BASEADDR) is only replacing the offset component by zero, while evaluating the offset is only forgetting the bloc identifier part (rule T-OFS). The size of a block (rule T-BLOCKLEN) is obtained by calling the corresponding operation of the memory model.

The evaluation of a predicate returns a value $\top$ (true) or $\bot$ (false). This is made possible because E-ACSL specifications are executable and because of our design choices concerning execution errors in annotations. Indeed, in our operational semantics errors are not explicitly represented and manipulated. An error means that it is not possible to construct a derivation tree for the execution.

Figure 11 gives the rules to evaluate builtin predicates. The validity of a pointer (rules P-VALID and P-INVALID) rely on the corresponding operation in the memory model. Initialization (rules P-INITIALIZED and P-UNINITIALIZED) is defined depending on the fact a read operation is possible and the returned value is different from Undef (which characterizes uninitialized data). The other

$$\textsc{E-int}\ \frac{}{E, M \vDash_{\mathrm{e}} n \Downarrow \mathsf{Int}(n)} \qquad \textsc{LV-var}\ \frac{E(x) = \lfloor b \rfloor}{E, M \vDash_{\mathrm{lv}} x \Downarrow b, 0} \qquad \textsc{LV-deref}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{lv}} *e \Downarrow b, \delta} \qquad \textsc{E-addr}\ \frac{E, M \vDash_{\mathrm{lv}} e \Downarrow b, \delta}{E, M \vDash_{\mathrm{e}} \&e \Downarrow \mathsf{Ptr}(b, \delta)}$$

$$\textsc{E-lval}\ \frac{E, M \vDash_{\mathrm{lv}} e : \tau \Downarrow b, \delta \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef} \qquad \mathsf{kind}(\tau) = \mathsf{kind}(v)}{E, M \vDash_{\mathrm{e}} e \Downarrow v}$$

$$\textsc{E-unop}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow v_1 \qquad \mathsf{sem\_unop}(\dagger, v_1) = \lfloor v_2 \rfloor}{E, M \vDash_{\mathrm{e}} \dagger\ e \Downarrow v_2}$$

$$\textsc{E-binop}\ \frac{E, M \vDash_{\mathrm{e}} e_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{e}} e_2 \Downarrow v_2 \qquad \mathsf{sem\_binop}(\ddagger, v_1, v_2) = \lfloor v_3 \rfloor}{E, M \vDash_{\mathrm{e}} e_1\ \ddagger\ e_2 \Downarrow v_3}$$

$$\textsc{E-parith}\ \frac{\begin{array}{c} E, M \vDash_{\mathrm{e}} e_1 : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \\ E, M \vDash_{\mathrm{e}} e_2 \Downarrow n \qquad sz = \mathsf{sizeof}(\mathbf{mtyp}(\tau)) \qquad \delta \bmod sz = 0 \qquad \oplus \in \{+, -\} \end{array}}{E, M \vDash_{\mathrm{e}} e_1 \oplus e_2 \Downarrow \mathsf{Ptr}(b, \delta \oplus n \times sz)}$$

Fig. 9. Evaluation of expressions

predicates are comparisons of terms. We see again that the semantics is error-free. For instance, if term t in `\valid`(t) cannot be evaluated (e.g. if its evaluation requires accessing a location in a block that was deallocated or a location that is out-of-bounds), there is no way to deduce any validity value for `\valid`(t).

The evaluation of logic connectors is presented in Fig. 12. The evaluation is lazy: the second argument is evaluated only if needed. Lazy evaluation of logic connectors is not mandatory for our formalization but this choice is consistent with what happens in the E-ACSL plugin.

After the evaluation of an instruction, the environment remains the same after each rule so only the resulting memory is given as the result of evaluation. Figure 13 shows the rules for simple instructions. In our language, allocation (rule S-malloc) is the combination of two operations in the memory: a new block b' of the requested size is first allocated, the memory is then in an intermediate state M_2 ; second, a pointer to the first cell of this new block is written at the location defined by the left expression of the allocation. `free` takes as argument a pointer (to the first cell of a block) and deallocates it using the free operation of the memory model (rule S-free). To allow dynamic deallocation only for blocks that were allocated dynamically (with `malloc`) and to prevent it for a block allocated automatically (that is, for a block associated to a variable), we add the constraint $b \notin \mathsf{im}(E)$ (see rule S-free). This ensures that if the original context (E, M_1) was well-formed, this property also holds for the new context (E, M_2) (that will be used for evaluation of the following instruction, see rule S-seq in Fig. 14). The verification of an assertion (rule S-logical-assert) simply evaluates its predicate argument. If the value is $\top$ the evaluation continues, otherwise the program has no semantics.

Figure 14 gives the rules for composite instructions. Most are unsurprising. The slightly unusual instruction `let` that introduces a local variable, first allocates space in memory, then binds the

$$\textsc{T-baseaddr}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{t}} \backslash\mathtt{base\_address}(t) \Downarrow \mathsf{Ptr}(b, 0)} \qquad \textsc{T-ofs}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{t}} \backslash\mathtt{offset}(t) \Downarrow \mathsf{Int}(\delta)}$$

$$\textsc{T-blocklen}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{length}(M, b) = n}{E, M \vDash_{\mathrm{t}} \backslash\mathtt{block\_length}(t) \Downarrow \mathsf{Int}(n)} \qquad \textsc{T-expr}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow v}{E, M \vDash_{\mathrm{t}} e \Downarrow v}$$

$$\textsc{T-deref}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef} \qquad \mathsf{kind}(\tau) = \mathsf{kind}(v)}{E, M \vDash_{\mathrm{t}} \bar{*}t \Downarrow v}$$

$$\textsc{T-unop}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow v_1 \qquad \mathsf{sem\_unop}(\bar{\dagger}, v_1) = \lfloor v_2 \rfloor}{E, M \vDash_{\mathrm{t}} \bar{\dagger}\, t \Downarrow v_2}$$

$$\textsc{T-binop}\ \frac{E, M \vDash_{\mathrm{t}} t_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{t}} t_2 \Downarrow v_2 \qquad \mathsf{sem\_binop}(\bar{\ddagger}, v_1, v_2) = \lfloor v_3 \rfloor}{E, M \vDash_{\mathrm{t}} t_1 \bar{\ddagger}\, t_1 \Downarrow v_3}$$

Fig. 10. Evaluation of terms.

$$\textsc{P-true}\ \frac{}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{true} \Downarrow \top} \qquad \textsc{P-false}\ \frac{}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{false} \Downarrow \bot}$$

$$\textsc{P-valid}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad M \vDash \mathsf{mtype}(\tau)\ @\ b, \delta}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(t) \Downarrow \top} \qquad \textsc{P-invalid}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad M \nvDash \mathsf{mtype}(\tau)\ @\ b, \delta}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(t) \Downarrow \bot}$$

$$\textsc{P-initialized}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef}}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{initialized}(t) \Downarrow \top} \qquad \textsc{P-uninitialized}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) \in \{\lfloor \mathsf{Undef} \rfloor, \varepsilon\}}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{initialized}(t) \Downarrow \bot}$$

$$\textsc{P-cmp}\ \frac{E, M \vDash_{\mathrm{t}} t_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{t}} t_2 \Downarrow v_2 \qquad \mathsf{sem\_cmp}(\bar{\bowtie}, v_1, v_2) = V}{E, M \vDash_{\mathrm{p}} t_1 \bar{\bowtie} t_2 \Downarrow V}$$

Fig. 11. Evaluation of predicates: basic predicates.

variable name and the obtained block. This gives a new context. The statement s is evaluated in this new context. Assumption $x \notin \mathrm{dom}(E_1)$ states the aforementioned assumption that variables

$$\textsc{P-conj-left}\ \frac{E, M \vDash_{\mathrm{p}} p_1 \Downarrow \bot}{E, M \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow \bot} \qquad \textsc{P-conj-right}\ \frac{E, M \vDash_{\mathrm{p}} p_1 \Downarrow \top \qquad E, M \vDash_{\mathrm{p}} p_2 \Downarrow V}{E, M \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow V}$$

$$\textsc{P-neg-false}\ \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \top}{E, M \vDash_{\mathrm{p}} \neg p \Downarrow \bot} \qquad \textsc{P-neg-true}\ \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \bot}{E, M \vDash_{\mathrm{p}} \neg p \Downarrow \top}$$

Fig. 12. Evaluation of predicates: logical connectors.

$$\textsc{S-skip}\ \frac{}{E, M \vDash_{\mathrm{s}} \mathtt{skip}; \Downarrow M} \qquad \textsc{S-assign}\ \frac{E, M_1 \vDash_{\mathrm{e}} e_2 : \tau \Downarrow v \qquad E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau \Downarrow b, \delta \qquad \mathsf{store}(\mathsf{mtype}(\tau), M_1, b, \delta, v) = \lfloor M_2 \rfloor}{E, M_1 \vDash_{\mathrm{s}} e_1 = e_2; \Downarrow M_2}$$

$$\textsc{S-malloc}\ \frac{E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Int}(n) \qquad \mathsf{alloc}(M_1, n) = (b', M_2) \qquad E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau* \Downarrow b, \delta \qquad \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b, \delta, \mathsf{Ptr}(b', 0)) = \lfloor M_3 \rfloor}{E, M_1 \vDash_{\mathrm{s}} e_1 = \mathtt{malloc}(e_2); \Downarrow M_3}$$

$$\textsc{S-free}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, 0) \qquad \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \qquad b \notin \mathsf{im}(E)}{E, M_1 \vDash_{\mathrm{s}} \mathtt{free}(e); \Downarrow M_2} \qquad \textsc{S-logical-assert}\ \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \top}{E, M \vDash_{\mathrm{s}} \mathtt{logical\_assert}(p); \Downarrow M}$$

Fig. 13. Evaluation of simple instructions.

are never overloaded. Finally the block corresponding to the variable is deallocated yielding the final memory state. This rule calls auxiliary functions alloc_var and dealloc_var defined below:

$$\begin{aligned}
\mathsf{alloc_var}(E_1, M_1, x, \tau) &\triangleq (E_2, M_2), \\
\text{where } E_2(x') &\triangleq \begin{cases} \lfloor b \rfloor & \text{if } x' = x, \\ E_1(x') & \text{otherwise,} \end{cases} \\
\text{and } (b, M_2) &\triangleq \mathsf{alloc}(M_1, \mathsf{sizeof}(\mathsf{mtype}(\tau))); \\
\mathsf{dealloc_var}(E_1, M_1, x) &\triangleq \begin{cases} \lfloor M_2 \rfloor & \text{if } E_1(x) = \lfloor b \rfloor, \\ & \text{and } \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor, \\ \varepsilon & \text{otherwise.} \end{cases}
\end{aligned}$$

We also denote by $E_1 \sqcup (x, b)$ the environment E_2 (defined above) constructed from E_1 by adding an association (x, b).

In rule S-let, function alloc_var creates a new variable x together with a valid block b such that $E_2(x) = \lfloor b \rfloor$, so the resulting intermediate context (E_2, M_2) (used for evaluation of instruction s) is well-formed. Next, function dealloc_var removes the allocated block b associated with variable x, so we also have to justify that the well-formedness of the new context is preserved. Notice that the evaluation of the following instructions after the `let` instruction, if any, will continue in the resulting context (E_1, M_4) (according to the rule S-seq) with the original environment E_1 where

$$\textsc{S-seq}\ \frac{E, M_1 \vDash_{\mathrm{s}} s_1 \Downarrow M_2 \qquad E, M_2 \vDash_{\mathrm{s}} s_2 \Downarrow M_3}{E, M_1 \vDash_{\mathrm{s}} s_1\ s_2 \Downarrow M_3}$$

$$\textsc{S-if-false}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0) \qquad E, M_1 \vDash_{\mathrm{s}} s_2 \Downarrow M_2}{E, M_1 \vDash_{\mathrm{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2}$$

$$\textsc{S-if-true}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad E, M_1 \vDash_{\mathrm{s}} s_1 \Downarrow M_2}{E, M_1 \vDash_{\mathrm{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2}$$

$$\textsc{S-while-false}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{E, M \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M}$$

$$\textsc{S-while-true}\ \frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \\ E, M_1 \vDash_{\mathrm{s}} s \Downarrow M_2 \qquad E, M_2 \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M_3 \end{array}}{E, M_1 \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M_3}$$

$$\textsc{S-let}\ \frac{\begin{array}{c} (E_2, M_2) = \mathsf{alloc\_var}(E_1, M_1, x, \tau) \\ x \notin \mathsf{dom}(E_1) \qquad E_2, M_2 \vDash_{\mathrm{s}} s \Downarrow M_3 \qquad \lfloor M_4 \rfloor = \mathsf{dealloc\_var}(E_2, M_3, x) \end{array}}{E_1, M_1 \vDash_{\mathrm{s}} \texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end} \Downarrow M_4}$$

Fig. 14. Evaluation of composite instructions.

variable x was not yet present, so the well-formedness of the new context is preserved as well. Hence, the well-formedness of the resulting evaluation context is preserved in all cases possibly adding a variable or removing a block (rules S-FREE and S-LET).

From the semantics of our language and the fact that it does not contain pointer casts, it is easily seen that a badly aligned or badly typed access cannot be created. The type of the element(s) in a block is determined according to the type of the destination pointer at the moment of dynamic allocation with `malloc` (or the type of the created variable at the moment of automatic allocation with `let`), and all accesses to the block element(s) are performed with this type. Pointer arithmetic (see rule E-PARITH in Fig. 9) preserves the alignment. It explains that all memory accesses in our programs are well-typed and aligned. For instance, consider the attempt to create aliases between pointers `p16` and `p8` and overwrite the first byte of `*p16` (on lines 5–6) in the following program:

```
1 let p16: int16* in
2 let p8 : int8*  in
3   p16  = malloc(4); // array of two 16-bit integers (4 bytes)
4   *p16 = Int(420);  // writing an element (1st and 2nd bytes)
5   p8   = p16;       // attempt to cast to a byte pointer
6   *p8  = Int(42);   // attempt to write to 1st byte
7   p16  = p8 + 1;    // attempt to create a badly aligned pointer
8   *p16 = Int(420);  // attempt to write to 2nd and 3rd bytes
9 end;
```

With the considered semantics, the assignment of line 5 cannot be evaluated since the types do not correspond (see rule S-ASSIGN in Fig. 13), and the evaluation will not proceed further. For the same reason, an attempt to perform a badly aligned access (through a combination of pointer casts and pointer arithmetic) on lines 5,7,8 does not succeed either.

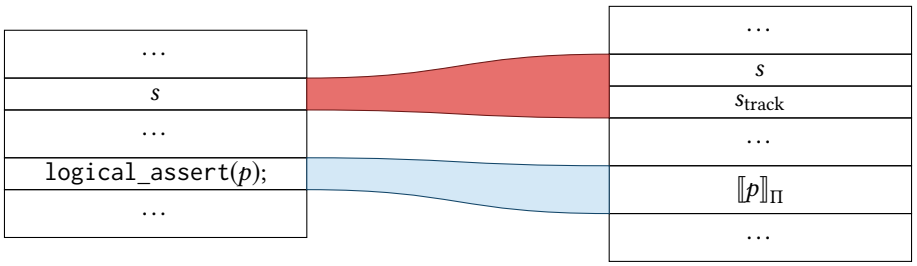

Fig. 15. Instrumentation scheme translating predicates into chunks of code (illustrated by the lower translation, shown in light blue) and adding statements for observing the memory operations (illustrated by the upper translation, shown in dark red).

Despite those restrictions, our langage remains representative of various validity issues and runtime assertion checking of memory properties of a real-life language: invalid accesses can still be created due to a dangling pointer or an out-of-bound access in a block. For example, if `malloc(2)` is replaced by `malloc(1)` on line 3 in the previous example, then accessing `*p16` on line 4 would become invalid (the allocated block contains only one byte while two bytes are needed). In that case, the evaluation of line 3 will be blocked in our semantics since the store operation in rule S-ASSIGN (see Fig. 13) cannot succeed for an invalid access (cf. Axiom (17)).

4 PROGRAM TRANSFORMATION

This section formally defines a program transformation that generates the instrumentation required for monitoring logical assertions at runtime. The purpose of the instrumentation is twofold as sketched in Fig. 15: first, it translates each logical assertion a into a chunk of executable code inserted at the very same program point as a and, second, it inserts additional statements into the original code in order to track the state of the execution memory and record it in a so-called observation memory.

Consider for instance the example of Fig 1, which contains on line 5 the logical assertion `/*@ assert \valid(t+mid); */`. The program transformation inserts two lines of code on line 14 in order to keep track in the observation memory that array `t` has been created and initialized as already explained in Section 2. Then, checking the logical assertion comes down to verifying the validity of the array access by querying the observation memory through a dedicated function call. As shown in Section 5, the whole program transformation is sound as soon as our implementation of the observation memory is correct with respect to its specification.

Before formalizing in Section 4.3 the program transformation itself, Sections 4.1 and 4.2 introduce respectively a model for the observation memory and the target language in which the code is generated.

4.1 Observation Memory Model

The observation memory is basically a data structure for the runtime monitor to store metadata about the execution memory of the program under monitoring. Observation memory states will be denoted by $\overline{M}$, $\overline{M'}$, $\overline{M_1}$, $\overline{M_2}$,... Notice that the bar symbol $\overline{\ }$ is part of the notation (so that $\overline{M}$ is not necessarily the observation memory of M). As for the execution memory type, we define the observation memory with an abstract type, named **obs**, and a set of operators, introduced in Fig. 16. There exists an empty observation memory $\overline{M_\emptyset}$ (that will be used as an initial observation memory

$$\begin{aligned}
\overline{M_\emptyset} &: \mathbf{obs} & \mathsf{is_valid_access} &: \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{bool} \\
\mathsf{store_block} &: \mathbf{obs} \times \mathbf{block} \times \mathbb{N} \to \mathbf{obs} & \mathsf{is_initialized} &: \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{bool} \\
\mathsf{delete_block} &: \mathbf{obs} \times \mathbf{block} \to \mathbf{obs} & \mathsf{initialize} &: \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{obs} \\
\mathsf{is_valid} &: \mathbf{obs} \times \mathbf{block} \to \mathbf{bool} & \mathsf{length} &: \mathbf{obs} \times \mathbf{block} \to \mathbb{N}
\end{aligned}$$

Fig. 16. Operations of the observation memory model.

state for evaluation of our target programs) that does not record any valid blocks or initialized locations.

The function is_valid_access is actually a shortcut defined as follows:

$$\begin{aligned}
\mathsf{is_valid_access}(\kappa, \overline{M}, b, \delta) = \quad & \mathsf{is_valid}(\overline{M}, b) \\
& \wedge\ \delta \geqslant 0 \\
& \wedge\ \delta + \mathsf{sizeof}(\kappa) \leqslant \mathsf{length}(\overline{M}, b).
\end{aligned}$$

It means that according to $\overline{M}$, memory block b can be accessed on $\mathsf{sizeof}(\kappa)$ bytes starting from a non-negative offset δ.

The meaning of other functions is defined by a set of axioms introduced below. Here again, we consider that all free variables in these axioms are universally quantified.

Definition 4.1 (Observation memory model). The observation memory model is defined by the axioms of Fig. 17.

The axioms are organized similarly to those of Fig. 7. Axioms (21) and (22) (resp. (23) and (24)) state that storing (resp. deleting) a block b in the observation memory means that b is now assumed to be valid (resp. invalid), and this operation has no impact on the validity of other blocks. Axiom (25) states that marking a block as being initialized has no impact on the validity of any block.

Axiom (26) states that storing a new block marks it as being currently uninitialized. Axioms (27) and (28) state that storing or deleting a block has no impact on the initialization status of any other block. Axioms (29) and (31) state that marking a block b as initialized on $\mathsf{sizeof}(\kappa)$ bytes from offset δ means that the observation memory now considers it as being initialized. It has an impact neither on the rest of the block b, nor on the other memory blocks. Like for the execution memory, we do consider possible overlapping or badly typed accesses (grouped in Axiom (30)). Even if such situations cannot occur for our programs (as we explained in Section 3.5), this choice will be useful to define a notion of representation of an execution memory by an observation memory below (Definition 5.5), independently of the operational semantics, in order to make our models suitable for future extensions of our languages.

Axiom (32) states that the length of a block b is defined when storing it. It is not lost as long as the block is not deleted. Axioms (33), (34) and (35) state that storing or deleting another block, or initializing a location, have no impact on the length of a block.

Finally, Axioms (36), (37) state that all blocks are invalid and all locations are non-initialized in $M_\emptyset$.

As for the execution memory model, in order to exclude any risk of inconsistency in the axioms of the observation memory model, we formalize them and prove their consistency with respect to a simple implementation in our Coq development in the same way as for the execution memory model. We define a module *type* for signatures and axioms and provide another module satisfying this module *type* to implement the functions and to demonstrate that they satisfy the axioms.

$$\text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \text{is\_valid}(\overline{M_2}, b) = \top \tag{21}$$

$$b \neq b' \land \text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \text{is\_valid}(\overline{M_2}, b') = \text{is\_valid}(\overline{M_1}, b') \tag{22}$$

$$\text{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \text{is\_valid}(\overline{M_2}, b) = \bot \tag{23}$$

$$b \neq b' \land \text{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \text{is\_valid}(\overline{M_2}, b') = \text{is\_valid}(\overline{M_1}, b') \tag{24}$$

$$\text{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \text{is\_valid}(\overline{M_2}, b') = \text{is\_valid}(\overline{M_1}, b') \tag{25}$$

$$\text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \text{is\_initialized}(\kappa, \overline{M_2}, b, \delta) = \bot \tag{26}$$

$$b \neq b' \ \land \ \text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \left\{ \begin{array}{l} \text{is\_initialized}(\kappa, \overline{M_2}, b', \delta') \\ = \text{is\_initialized}(\kappa, \overline{M_1}, b', \delta') \end{array} \right. \tag{27}$$

$$b \neq b' \land \text{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \left\{ \begin{array}{l} \text{is\_initialized}(\kappa, \overline{M_2}, b', \delta') = \\ = \text{is\_initialized}(\kappa, \overline{M_1}, b', \delta') \end{array} \right. \tag{28}$$

$$\text{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \text{is\_initialized}(\kappa, \overline{M_2}, b, \delta) = \top \tag{29}$$

$$\left. \begin{array}{l} \text{initialize}(\kappa, M_1, b, \delta, v) = M_2 \land \\ (\delta' = \delta \land \kappa' \neq \kappa \lor \delta' \neq \delta) \land \\ \delta + \text{sizeof}(\kappa) > \delta' \land \delta' + \text{sizeof}(\kappa') > \delta \land \\ \delta' \geqslant 0 \ \land \ \delta' + \text{sizeof}(\kappa') \leqslant \text{length}(M_2, b) \end{array} \right\} \implies \text{is\_initialized}(\kappa', M_2, b, \delta') = \bot \tag{30}$$

$$\left. \begin{array}{l} \text{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \ \land \ (b \neq b' \lor \\ \delta + \text{sizeof}(\kappa) \leqslant \delta' \ \lor \ \delta' + \text{sizeof}(\kappa') \leqslant \delta) \end{array} \right\} \implies \left\{ \begin{array}{l} \text{is\_initialized}(\kappa', \overline{M_2}, b', \delta') \\ = \text{is\_initialized}(\kappa', \overline{M_1}, b', \delta') \end{array} \right. \tag{31}$$

$$\text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \text{length}(\overline{M_2}, b) = n \tag{32}$$

$$b \neq b' \land \text{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \text{length}(\overline{M_2}, b') = \text{length}(\overline{M_1}, b') \tag{33}$$

$$\text{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \text{length}(\overline{M_2}, b') = \text{length}(\overline{M_1}, b') \tag{34}$$

$$b \neq b' \land \text{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \text{length}(\overline{M_2}, b') = \text{length}(\overline{M_1}, b') \tag{35}$$

$$\text{is\_valid}(M_\emptyset, b) = \bot \tag{36}$$

$$\text{is\_initialized}(\kappa, M_\emptyset, b, \delta) = \bot \tag{37}$$

Fig. 17. Axioms of observation memory model.

4.2 Target Language

The target language is quite close to the source language with a few differences, as shown in Fig. 18. First, the logical assertion disappears, being replaced by a program assertion `assert` similar to the C macro of the same name and taking an (executable) expression as argument. Second, the generated code relies on specific statements in order to query the observation model. In particular, they allow:

- testing pointer validity, thanks to `is_valid`;
- testing initialization of the location referred to by a given pointer, thanks to `is_initialized`;
- getting the base address of a given pointer, thanks to `base_address`;
- getting the offset (in bytes) of a given pointer from its base address, thanks to `offset`;
- getting the length of a block containing a given pointer thanks to `block_length`.

$s ::=$	`...`	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. 18. Additional statements of the target language.

Tracking memory operations requires us to update the observation model in order to store or delete a memory block thanks to `store_block` and `delete_block` respectively, or mark some memory locations as being initialized thanks to `initialize`. Recall that a valid memory location of some memory type κ is considered initialized if and only if a defined value can be read from it with memory type κ (cf. rules for initialization of Fig. 11). In this paper, we make the choice not to consider as initialized a memory location if it was overwritten with an incompatible type or by a partially overlapping writing operation (even if such situations cannot occur in our programs) because we cannot safely read such a value.

In order to give these primitives a semantics, we extend the evaluation relation of the source language with the state of the observation memory. Consequently, the evaluation relations for the target language take the following shapes:

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

Notice that the observation memory is not added into the evaluation context for expressions because their evaluation does not have any impact on the observation memory. The rules corresponding to statements that are already present in the input languages are almost unchanged: they are only updated by adding an observation memory that they propagate with no change, by replacing the prefix "S-" in the name of the rule by "S'-" and the derivation symbol "$\vDash_s$" by "$\vDash_{s'}$". Hence, these rules are omitted here. The rules corresponding to the new statements for `assert` and memory observation are presented in Fig. 19. Most of them rely on the observation memory functions introduced in Section 4.1.

The semantics of statements `store_block(`p, e`);`, `delete_block(`p`);`, and `initialize(`e`);` is directly mapped to their respective observation model's counterparts after having evaluated their arguments. When marking an expression as initialized (rule S'-INITIALIZE), its size is inferred from its type.

For a pointer e_2, rules S'-BASEADDR, S'-OFFSET, and S'-BLOCK-LENGTH respectively compute the base address of the memory block containing e_2, the length of this memory block, and the offset from the base address of e_2 to e_2. The first two rules do *not* rely on the observation memory: they only rely on how pointers are modeled in the execution memory through their base and offset. Since there is no direct way to know the length of a memory block in the execution memory, the third rule relies on the observation memory to get it. In these three cases, the resulting value is written in the execution memory, at the memory location corresponding to e_1.

Finally, the rule S'-IS-VALID (resp. S'-IS-INITIALIZED) queries the observation memory to get the validity (resp. initialization) status of a pointer and stores the resulting Boolean, as an integer 0 or 1, in the execution memory.

S'-ASSERT
$$\frac{E, M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{E, M, \overline{M} \vDash_{\mathrm{s}'} \texttt{assert}(e); \Downarrow M, \overline{M}}$$

S'-STOREBLOCK
$$\frac{E, M_1 \vDash_{\mathrm{e}} p \Downarrow \mathsf{Ptr}(b, 0) \qquad E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{store\_block}(\overline{M_1}, b, n) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{store\_block}(p, e); \Downarrow M_1, \overline{M_2}}$$

S'-DELETEBLOCK
$$\frac{E, M_1 \vDash_{\mathrm{e}} p \Downarrow \mathsf{Ptr}(b, 0) \qquad \mathsf{delete\_block}(\overline{M_1}, b) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{delete\_block}(p); \Downarrow M_1, \overline{M_2}}$$

S'-INITIALIZE
$$\frac{E, M_1 \vDash_{\mathrm{e}} e : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{initialize}(\mathsf{mtype}(\tau), \overline{M_1}, b, \delta) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{initialize}(e); \Downarrow M_1, \overline{M_2}}$$

S'-BASEADDR
$$\frac{E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau* \Downarrow b_1, \delta_1 \qquad E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathsf{mtype}(\tau*), M_1, b_1, \delta_1, \mathsf{Ptr}(b_2, 0)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \mathsf{base\_address}(e_2); \Downarrow M_2, \overline{M_1}}$$

S'-OFFSET
$$\frac{E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathsf{i64}, M_1, b_1, \delta_1, \mathsf{Int}(\delta_2)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{offset}(e_2); \Downarrow M_2, \overline{M_1}}$$

S'-BLOCK-LENGTH
$$\frac{E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{length}(\overline{M_1}, b_2) = \lfloor n \rfloor \qquad \mathsf{store}(\mathsf{i64}, M_1, b_1, \delta_1, \mathsf{Int}(n)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{block\_length}(e_2); \Downarrow M_2, \overline{M_1}}$$

S'-IS-VALID
$$\frac{E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad \mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M_1}, b_2, \delta_2) = \beta \qquad E, M_1 \vDash_{\mathrm{e}} e_2 : \tau* \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathsf{i8}, M_1, b_1, \delta_1, \mathsf{Int}(\beta)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{is\_valid}(e_2); \Downarrow M_2, \overline{M_1}}$$

S'-IS-INITIALIZED
$$\frac{E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad \mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M_1}, b_2, \delta_2) = \beta \qquad E, M_1 \vDash_{\mathrm{e}} e_2 : \tau* \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathsf{i8}, M_1, b_1, \delta_1, \mathsf{Int}(\beta)) = \lfloor M_2 \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{is\_initialized}(e_2); \Downarrow M_2, \overline{M_1}}$$

Fig. 19. Semantic Rules for Observation Memory Statements.

4.3 Formal Program Transformation

The program transformation consists of three functions: for statements, predicates and terms. Each of them returns a chunk of code in the target language, which consists of statements possibly using variables that are bounded outside of the current chunk.

Figure 20 introduces $[\![s]\!]_{\Sigma}$, which converts a statement to a chunk of code. It contains the original statement and new statements that monitor the memory operations. The first four cases, corresponding to the control-flow statements, have no such additional monitoring statements.

$$
\begin{array}{rcl}
[\![\texttt{skip;}]\!]_\Sigma & \triangleq & \texttt{skip;} \\
[\![s_1\ s_2]\!]_\Sigma & \triangleq & [\![s_1]\!]_\Sigma\ [\![s_2]\!]_\Sigma \\
[\![\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]_\Sigma & \triangleq & \texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_\Sigma\ \texttt{else}\ [\![s_2]\!]_\Sigma \\
[\![\texttt{while}\ (e)\ s]\!]_\Sigma & \triangleq & \texttt{while}\ (e)\ [\![s]\!]_\Sigma \\
[\![e_1 = e_2;]\!]_\Sigma & \triangleq & e_1 = e_2; \\
 & & \texttt{initialize}(\&e_1); \\
[\![p = \texttt{malloc}(e);]\!]_\Sigma & \triangleq & p = \texttt{malloc}(e); \\
 & & \texttt{store\_block}(p, e); \\
 & & \texttt{initialize}(\&p); \\
[\![\texttt{free}(p);]\!]_\Sigma & \triangleq & \texttt{free}(p); \\
 & & \texttt{delete\_block}(p); \\
[\![\texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end}]\!]_\Sigma & \triangleq & \texttt{let}\ x : \tau\ \texttt{in} \\
 & & \quad \texttt{store\_block}(\&x, \texttt{sizeof}(\tau)); \\
 & & \quad [\![s]\!]_\Sigma \\
 & & \quad \texttt{delete\_block}(\&x); \\
 & & \texttt{end} \\
[\![\texttt{logical\_assert}(p);]\!]_\Sigma & \triangleq & \texttt{let}\ \mathrm{res}(0) : \texttt{int8}\ \texttt{in} \\
 & & \quad [\![p]\!]^0_\Pi \\
 & & \quad \texttt{assert}(\mathrm{res}(0)); \\
 & & \texttt{end}
\end{array}
$$

Fig. 20. Statement translation.

Translating an assignment requires us to mark the left-value e_1 as being initialized. It is the same for a memory allocation, which also requires us to mark the allocated memory block as being valid, while translating a memory de-allocation must delete the block from the observation memory. Translating a local-binder needs first to mark the corresponding memory block as being allocated, and to delete it from the observation memory before leaving its scope. Translating a logical assertion requires us to translate the corresponding predicate and to generate a program assertion. Variable res(0) stores the result of the evaluation of the predicate at runtime, as explained below. Here, res is an injective function that takes an integer $n \geqslant 0$ as input and provides a variable name based on a generic prefix (say, `res_`) and the given integer suffix, so that res(0) gives the variable name `res_0`. Recall that we assumed that source program variables cannot be named res(n) (for $n \geqslant 0$).

$[\![p]\!]^n_\Pi$ is the translation function for predicates: it translates the predicate p to a truth value, encoded as an integer 0 (meaning false) or 1 (meaning true), and stores this result into the variable res(n), which is introduced right before translating the predicate. For instance, the variable res(0) is introduced right before translating the head predicate p of a logical assertion. The result for a subpredicate would be stored in res(1), which in turn can require res(n) with larger n) to store their results. To sum up, the translation scheme of a predicate p containing a sub-predicate p' is as

$$
\begin{array}{rcl}
[\![\texttt{\textbackslash true}]\!]^n_\Pi & \triangleq & \mathrm{res}(n) = 1; \\
[\![\texttt{\textbackslash false}]\!]^n_\Pi & \triangleq & \mathrm{res}(n) = 0; \\
[\![\neg p_1]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \texttt{int8 in} \\
 & & \quad [\![p_1]\!]^{n+1}_\Pi \\
 & & \quad \mathrm{res}(n) = \,!\,\mathrm{res}(n+1); \\
 & & \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
[\![p_1 \wedge p_2]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \texttt{int8 in} \\
 & & \quad [\![p_1]\!]^{n+1}_\Pi \\
 & & \quad \texttt{if } (\mathrm{res}(n+1)) \\
 & & \quad \texttt{then let } \mathrm{res}(n+2) : \texttt{int8 in} \\
 & & \quad\quad [\![p_2]\!]^{n+2}_\Pi \\
 & & \quad\quad \mathrm{res}(n) = \mathrm{res}(n+2); \\
 & & \quad \texttt{end else} \\
 & & \quad\quad \mathrm{res}(n) = 0; \\
 & & \texttt{end}
\end{array}
$$

Fig. 21. Basic predicate translation.

$$
\begin{array}{rcl}
[\![\texttt{\textbackslash valid}(t_1)]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \tau_{t_1} \texttt{ in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathrm{res}(n) = \texttt{is\_valid}(\mathrm{res}(n+1)); \\
 & & \texttt{end} \\
[\![\texttt{\textbackslash initialized}(t_1)]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \tau_{t_1} \texttt{ in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathrm{res}(n) = \texttt{is\_initialized}(\mathrm{res}(n+1)); \\
 & & \texttt{end} \\
[\![t_1 \bar{\bowtie} t_2]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \tau_{t_1} \texttt{ in} \\
 & & \texttt{let } \mathrm{res}(n+2) : \tau_{t_2} \texttt{ in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad [\![t_2]\!]^{n+2}_{\mathrm{T}} \\
 & & \quad \mathrm{res}(n) = \mathrm{res}(n+1) \ \bowtie \ \mathrm{res}(n+2); \\
 & & \texttt{end} \\
 & & \texttt{end}
\end{array}
$$

Fig. 22. Term-based predicate translation.

follows.

$$
\begin{array}{rcll}
[\![p]\!]^n_\Pi & \triangleq & \texttt{let } \mathrm{res}(n+1) : \texttt{int8 in} & \text{will store the result of } p' \\
 & & \quad [\![p']\!]^{n+1}_\Pi & \text{translate } p' \text{ and store its result to } \mathrm{res}(n+1) \\
 & & \quad \ldots\ \mathrm{res}(n+1)\ \ldots & \text{use the result of } p' \\
 & & \quad \mathrm{res}(n) = \ \ldots & \text{compute and store the result for } p \\
 & & \texttt{end} &
\end{array}
$$

Figure 21 defines $[\![p]\!]^n_\Pi$ for the basic predicates. Translating the truth value is trivial. For the logical connectors, we introduce a fresh local variable for each sub-predicate before translating them. Translating a conjunction relies on a classical encoding through a conditional.

Figure 22 translates predicates based on terms. Translating $\texttt{\textbackslash valid}(t_1)$ and $\texttt{\textbackslash initialized}(t_1)$ relies on the corresponding functions of the observation memory, while translating relational operators directly uses the corresponding operators of the target language. We denote by τ_{t_1} and τ_{t_2} the types of t_1 and t_2.

$$
\begin{array}{rcl}
[\![e]\!]^n_{\mathrm{T}} & \triangleq & \mathsf{res}(n) = e; \\
[\![\bar{*}t_1]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = *\mathsf{res}(n+1); \\
 & & \mathtt{end} \\
[\![\ \bar{\dagger}\ t_1]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \ \dagger\ \mathsf{res}(n+1); \\
 & & \mathtt{end}
\end{array}
\qquad
\begin{array}{rcl}
[\![t_1\ \bar{\ddagger}\ t_2]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \mathtt{let}\ \mathsf{res}(n+2) : \tau_{t_2}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad [\![t_2]\!]^{n+2}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \mathsf{res}(n+1)\ \ddagger\ \mathsf{res}(n+2); \\
 & & \mathtt{end} \\
 & & \mathtt{end}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\backslash\mathtt{base\_address}(t_1)]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \mathtt{base\_address}(\mathsf{res}(n+1)); \\
 & & \mathtt{end} \\
[\![\backslash\mathtt{offset}(t_1)]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \mathtt{offset}(\mathsf{res}(n+1)); \\
 & & \mathtt{end} \\
[\![\backslash\mathtt{block\_length}(t_1)]\!]^n_{\mathrm{T}} & \triangleq & \mathtt{let}\ \mathsf{res}(n+1) : \tau_{t_1}\ \mathtt{in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \mathtt{block\_length}(\mathsf{res}(n+1)); \\
 & & \mathtt{end}
\end{array}
$$

Fig. 23. Term translation.

Figure 23 translates terms by relying on the same translation scheme as predicates, where the resulting type depends on the term. For each case, the translation also uses the corresponding operators of the target language or the corresponding function of the observation memory.

Figure 24 shows the result of this transformation applied on the program of Fig. 3 (slightly simplified by merging initialize statements for each array element assignment into one statement for the whole array, see lines 7–8). The inserted statements are written mostly on the right side for clarity. Lines 21–31 are the translation of logical assertion `logical_assert(\valid(t + mid));`, which is the only line removed from the source program.

5 SOUNDNESS OF THE TRANSFORMATION

The transformation soundness theorem is illustrated by Fig. 25. Given some program s, on the one hand, the evaluation of s in the initial source context $(\widehat{E_1}, \widehat{M_1})$ results in the final source context $(\widehat{E_2}, \widehat{M_2})$. On the other hand, the transformed program $[\![s]\!]_\Sigma$ evaluated in the initial target context $(E_1, M_1, \overline{M_1})$ results in the final target context $(E_2, M_2, \overline{M_2})$. In order to distinguish between source-related elements (environments, memory states, values…) and target-related ones, the former are hereafter denoted with a hat symbol. Notice that the hat symbol $\widehat{\ }$ is part of the notation (so that, for instance, $\widehat{M}$ is not necessarily the source memory state related to the target memory state M).

```
let t: int64*  in                store_block(&t, sizeof(int64*));
let len: int64 in                store_block(&len, sizeof(int64));
let x: int64   in                store_block(&x, sizeof(int64));
  t = malloc (len * sizeof(int64));
                                 store_block(t, len*sizeof(int64));
                                 initialize(&t);
  *t = -3; *(t+1) = 2; *(t+2) = 4; *(t+3) = 7; *(t+4) = 10;
                                 initialize(&*t);
  len = 5;                       initialize(&len);
  x = 7;                         initialize(&x);

  let lo: int64  in              store_block(&lo, sizeof(int64));
  let hi: int64  in              store_block(&hi, sizeof(int64));
  let idx: int64 in              store_block(&idx, sizeof(int64));
    idx = -1;                    initialize(&idx);
    lo = 0;                      initialize(&lo);
    hi = len - 1;                initialize(&hi);
    while (lo <= hi)
    let mid: int64 in            store_block(&mid, sizeof(int64));
      mid = lo + (hi - lo) / 2; initialize(&mid);
                                 let res(0): int8 in
                                   let res(1): int8 in
                                     let res(2): int8 in
                                     let res(3): int8 in
                                     res(2) = t; res(3) = mid;
                                     res(1) = res(2) + res(3);
                                     end
                                     end
                                   res(0) = is_valid(res(1));
                                   assert(res(0));
                                 end
      if (*(t + mid) == x) then
        idx = mid;             initialize(&idx);
        lo = hi + 1;           initialize(&lo);
      else if (*(t + mid) < x) then
        lo = mid + 1;          initialize(&lo);
      else
        hi = mid - 1;          initialize(&hi);
    delete_block(&mid); end
  delete_block(&idx); end
  delete_block(&hi);  end
  delete_block(&lo);  end
delete_block(&x);    end
delete_block(&len); end
delete_block(&t);    end
```

Fig. 24. The transformation applied on the program of Fig. 3.

The soundness theorem states that, for a certain relation $\mathcal{R}$, if the initial source and target contexts are related by $\mathcal{R}$, then the target program may indeed evaluate in the initial target context, and the resulting final target context is also related to the final source context by $\mathcal{R}$. A keystone of the theorem is thus the invariant relation $\mathcal{R}$, which involves a number of auxiliary relations, formally defined in the following subsections.

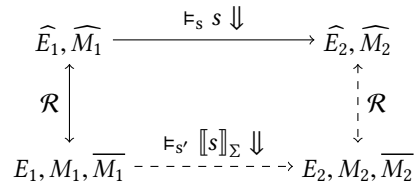

Fig. 25. Commutation diagram of the soundness theorem.

First, we introduce context *isomorphisms* as a way to reason about execution contexts up to a block permutation. This relation enables us to match source and target contexts with essentially the same structure and contents without requiring them to have the same block identifiers.

However, source and target memory states do not always have the same structure and contents, as can be guessed by reading, for instance, the predicate translation defined by Fig. 21. As the translation introduces additional variables, target memory states may become "bigger" than their source counterpart. Therefore, we need a notion of *subcontext* to express inclusion of a context into another one.

We also define *representation* of an execution memory by an observation memory, expressing that metadata on validity and initialization of memory locations in the observation memory accurately represents the execution memory contents.

Finally, we introduce *equivalence* as a stronger case of isomorphism, allowing systematic equational reasoning on memory states.

5.1 Isomorphisms between Execution Contexts

This section introduces the notion of an isomorphism that will be useful to compare execution contexts in programs before and after the transformation and to ensure that the source context and (a subcontext of) the target context have essentially the same structure and contents. Two contexts are isomorphic if they have the same contents, up to a permutation of blocks (or, more precisely, of block identifiers). A *permutation* (of blocks) is a bijective function $\sigma : \mathbf{block} \to \mathbf{block}$. Let us denote by id the identity function, by σ^{-1} the inverse of permutation σ, and by $(x \leftrightarrow y)$ the transposition of x and y, that is, the permutation that exchanges x and y and leaves untouched all other blocks. A permutation of blocks induces a function on values which exchanges the blocks inside pointers accordingly to the permutation, and leaves untouched all other values.

Definition 5.1 (Induced function). Let σ be a permutation of blocks. The *function induced by* σ is a mapping of values $\dot{\sigma} : \mathbf{value} \to \mathbf{value}$ defined as follows:

$$\begin{array}{rcl} \dot{\sigma}(\mathsf{Int}(n)) & \triangleq & \mathsf{Int}(n), \\ \dot{\sigma}(\mathsf{Undef}) & \triangleq & \mathsf{Undef}, \\ \dot{\sigma}(\mathsf{Ptr}(b, \delta)) & \triangleq & \mathsf{Ptr}(\sigma(b), \delta). \end{array}$$

We can now define isomorphic contexts.

Definition 5.2 (Isomorphic contexts). Let $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ be two contexts, and $\sigma : \mathbf{block} \to \mathbf{block}$ a permutation of blocks. We say that context C_1 is *isomorphic* to context C_2

(with an isomorphism induced by permutation σ), and write $C_1 \sim C_2$ (or $C_1 \sim_\sigma C_2$ when σ is relevant), if

(1) environments E_1 and E_2 are defined on the same variables and map them to the corresponding blocks, that is, $\forall x, b, E_1(x) = \lfloor b \rfloor \iff E_2(x) = \lfloor \sigma(b) \rfloor$;
(2) a block b in M_1 and the corresponding block $\sigma(b)$ in M_2 are valid at the same time and in this case have the same length, that is, $\forall b,\ M_1 \vDash b \iff M_2 \vDash \sigma(b)$ and $\forall b, M_1 \vDash b \implies \mathsf{length}(M_1, b) = \mathsf{length}(M_2, \sigma(b))$;
(3) a value can be read from a location in M_1 if and only if the corresponding value can be read from the corresponding location in M_2, that is, $\forall \kappa, b, \delta, v, \mathsf{load}(\kappa, M_1, b, \delta) = \lfloor v \rfloor \iff \mathsf{load}(\kappa, M_2, \sigma(b), \delta) = \lfloor \dot{\sigma}(v) \rfloor$.

If $C_1 \sim_\sigma C_2$, it follows from the last condition of Def. 5.2 and Axiom (18) that an access in memory M_1 is valid if and only if the same access in the corresponding block in M_2 is valid:

Property 1. *If $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ are two contexts such that $C_1 \sim_\sigma C_2$, then $\forall \kappa, b, \delta,\ M_1 \vDash \kappa \mathbin{@} b, \delta \iff M_2 \vDash \kappa \mathbin{@} \sigma(b), \delta$.*

In addition to a simultaneous validity of accesses, the last condition of Def. 5.2 can be applied to pointer values to deduce that a block b' is referred to by an existing pointer in M_1 if and only if the corresponding block $\sigma(b')$ is referred to by the corresponding pointer in M_2. More formally, $\forall \kappa, b, b', \delta, \delta', \mathsf{load}(\kappa, M_1, b, \delta) = \lfloor \mathsf{Ptr}(b', \delta') \rfloor \iff \mathsf{load}(\kappa, M_2, \sigma(b), \delta) = \lfloor \mathsf{Ptr}(\sigma(b'), \delta') \rfloor$. Together with a simultaneous validity of the corresponding blocks stated in the second condition of Def. 5.2, it allows us to deduce an interesting property for the supports of isomorphic contexts.

Property 2. *Let $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ be two contexts such that $C_1 \sim_\sigma C_2$. Then we have $b \in \mathit{supp}(M_1)$ if and only if $\sigma(b) \in \mathit{supp}(M_2)$. Moreover, permutation σ inducing the isomorphism $C_1 \sim_\sigma C_2$ is not unique: it matters only on $\mathit{supp}(M_1)$, on which it defines a bijection with $\mathit{supp}(M_2)$, and it can arbitrarily associate the elements of $\mathbf{block} \setminus \mathit{supp}(M_1)$ to the elements of $\mathbf{block} \setminus \mathit{supp}(M_2)$.*

Proof. The proof of both statements is similar and follows from Def. 5.2 and the definition of support. Let us show here the second statement. Indeed, the only blocks that are involved in the conditions of the definition are blocks $b \in \mathrm{supp}(M_1)$ for context C_1 and their counterparts $\sigma(b) \in \mathrm{supp}(M_2)$ for C_2. By definition of support, if we take a block $b \notin \mathrm{supp}(M_1)$, then it is neither a valid block in M_1 nor a block referred to by an existing pointer value in M_1. Since our contexts are well-formed, the non-validity condition also implies that block b is not associated with a variable by E_1. Similarly, if we consider a block $\sigma(b) \notin \mathrm{supp}(M_2)$, then $\sigma(b)$ is neither a valid block in M_2 nor a block referred to by an existing pointer value in M_2. Since our contexts are well-formed, the non-validity condition also implies that block $\sigma(b)$ is not associated with a variable by E_2. Therefore, the definition of permutation σ inducing an isomorphism of contexts $C_1 \sim_\sigma C_2$ matters only on $\mathrm{supp}(M_1)$, on which it creates a bijection with $\mathrm{supp}(M_2)$. It can arbitrarily associate the elements of $\mathbf{block} \setminus \mathrm{supp}(M_1)$ to the elements of $\mathbf{block} \setminus \mathrm{supp}(M_2)$. □

This property will be useful in the proof of the next property, which expresses that the isomorphism of contexts is preserved by block allocation.

Property 3. *An isomorphism of contexts is preserved after allocation of a block of the same size in both contexts. More precisely,*

$$\left.\begin{array}{l} \forall E_1, E_2, M_1, M_2, M_1', M_2', b_1, b_2, \sigma, n \\ \quad (E_1, M_1) \sim_\sigma (E_2, M_2) \\ \quad \mathit{alloc}(M_1, n) = (b_1, M_1') \\ \quad \mathit{alloc}(M_2, n) = (b_2, M_2') \end{array}\right\} \implies (E_1, M_1') \sim_{\sigma'} (E_2, M_2')$$

where $\sigma' \triangleq \sigma \circ (b_1 \leftrightarrow \sigma^{-1}(b_2))$.

Proof. By Axiom (16) for the newly allocated blocks we know that $b_1 \notin \mathrm{supp}(M_1)$ and $b_2 \notin \mathrm{supp}(M_2)$. By Property 2, we can deduce that $\sigma^{-1}(b_2) \notin \mathrm{supp}(M_1)$ and $\sigma(b_1) \notin \mathrm{supp}(M_2)$. Permutation σ' defined in the statement differs from σ only on b_1 and $\sigma^{-1}(b_2)$ that do not belong to $\mathrm{supp}(M_1)$: their images are swapped to become, resp., b_2 and $\sigma(b_1)$ after σ', instead of, resp., $\sigma(b_1)$ and b_2 after σ. Its definition can be rewritten as follows:

$$\sigma'(b) = \begin{cases} b_2 & \text{if } b = b_1, \\ \sigma(b_1) & \text{if } b = \sigma^{-1}(b_2), \\ \sigma(b) & \text{otherwise.} \end{cases}$$

By Property 2 again, permutation σ' induces the same isomorphism as σ between contexts (E_1, M_1) and (E_2, M_2), and in addition σ' associates the newly associated blocks together: it maps b_1 to b_2. It is easily seen by verifying the conditions of Def. 5.2 that permutation σ' induces an isomorphism of contexts $(E_1, M_1') \sim_{\sigma'} (E_2, M_2')$. The detailed verification is straightforward and left to the reader. □

Property 4. *An isomorphism of contexts is preserved after deallocation of corresponding blocks in both contexts. More precisely,*

$$\left.\begin{array}{l} \forall E_1, E_2, M_1, M_2, b, \sigma, \\ \quad (E_1, M_1) \sim_\sigma (E_2, M_2) \\ \quad M_1 \vDash b \\ \quad b \notin \mathit{im}(E_1) \end{array}\right\} \implies \left\{\begin{array}{l} \exists M_1',\ \mathit{free}(M_1, b) = \lfloor M_1' \rfloor \\ \exists M_2',\ \mathit{free}(M_2, \sigma(b)) = \lfloor M_2' \rfloor \\ (E_1, M_1') \sim_\sigma (E_2, M_2') \end{array}\right.$$

Proof. The proof follows from Def. 5.2 and the axioms of the memory model (Def. 3.3). The verification of the required conditions is straightforward and left to the reader. □

Property 5. *An isomorphism of contexts preserves the possibility to store corresponding values into corresponding locations in both contexts, and when the storing succeeds, the resulting contexts are isomorphic again. More precisely,*

$$\forall E_1, E_2, M_1, M_2, \kappa, b, \delta, v, \sigma, \ \textit{such that}\ (E_1, M_1) \sim_\sigma (E_2, M_2), \textit{ then we have:}$$
$$\left\{\begin{array}{l} \textit{if } \exists M_1',\ \mathit{store}(\kappa, M_1, b, \delta, v) = \lfloor M_1' \rfloor, \textit{ then} \\ \quad \exists M_2',\ \mathit{store}(\kappa, M_2, \sigma(b), \delta, \dot\sigma(v)) = \lfloor M_2' \rfloor \wedge (E_1, M_1') \sim_\sigma (E_2, M_2'); \\ \textit{if } \mathit{store}(\kappa, M_1, b, \delta, v) = \varepsilon, \textit{ then } \mathit{store}(\kappa, M_2, \sigma(b), \delta, \dot\sigma(v)) = \varepsilon. \end{array}\right.$$

Proof. The proof follows from Property 1, Def. 5.2 and the axioms of the memory model (Def. 3.3). The verification of the required conditions is straightforward and left to the reader. □

Along with context isomorphisms, we also need the notion of subcontext.

Definition 5.3 (Subcontext). Let $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ be two contexts. We say that C_1 is a subcontext of C_2, and write $C_1 \subseteq C_2$, if

(1) E_2 extends E_1, that is, $\mathrm{dom}(E_1) \subseteq \mathrm{dom}(E_2)$, and $\forall x, b, E_1(x) = \lfloor b \rfloor \implies E_2(x) = \lfloor b \rfloor$;
(2) any valid block of M_1 is also a valid block of M_2 and has the same length, that is, $\forall b, M_1 \vDash b \implies M_2 \vDash b \wedge \mathsf{length}(M_1, b) = \mathsf{length}(M_2, b)$;
(3) whenever a value can be read from a location in M_1, the same value is read from the same location in M_2, that is, $\forall \kappa, b, \delta, v, \mathsf{load}(\kappa, M_1, b, \delta) = \lfloor v \rfloor \implies \mathsf{load}(\kappa, M_2, b, \delta) = \lfloor v \rfloor$.

If $(E_1, M_1) \subseteq (E_2, M_2)$, it follows from the last condition and Axiom (18) that a valid access of memory M_1 is also a valid access in M_2:

Property 6. *If $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ are two contexts such that $C_1 \subseteq C_2$, then* $\forall \kappa, b, \delta,\ M_1 \vDash \kappa \,@\, b, \delta \implies M_2 \vDash \kappa \,@\, b, \delta$.

5.2 Observational Equivalence for Execution Memory States

Observational equivalence (or simply equivalence) between two memory states means that no difference can be observed regarding validity, length and contents of blocks via the available access functions. As our definition of memory model is axiomatic and abstracts away all implementation details (such as order of blocks, spacing between blocks, etc.), we cannot (and do not need to) know that two memory states are physically equal: observational equivalence is sufficient for our results. This approach allows more flexibility for an actual implementation.

Definition 5.4 (Observational equivalence). Let M_1 and M_2 be two execution memory states. We say that they are *(observationally) equivalent*, and write $M_1 \equiv M_2$, if

(1) a block b is valid in M_1 and in M_2 at the same time, and in this case it has the same length in both, that is, $\forall b,\ M_1 \vDash b \iff M_2 \vDash b$ and $\forall b, M_1 \vDash b \implies \mathsf{length}(M_1, b) = \mathsf{length}(M_2, b)$;
(2) a value can be read from a location in M_1 if and only if the same value can be read from the same location in M_2, that is, $\forall \kappa, b, \delta, v, \mathsf{load}(\kappa, M_1, b, \delta) = \lfloor v \rfloor \iff \mathsf{load}(\kappa, M_2, b, \delta) = \lfloor v \rfloor$.

5.3 Representation of an Execution Memory by an Observation Memory

The purpose of the observation memory is to store metadata on validity and initialization of memory locations in the execution memory. To ensure that this metadata correctly represents the state of the execution memory, we introduce the notion of representation.

Definition 5.5 (Representation). Let M be an execution memory and $\overline{M}$ an observation memory. We say that observation memory $\overline{M}$ *represents* execution memory M, and write $M \triangleright \overline{M}$, if

(1) a block b is valid in M if and only if it is recorded as valid in $\overline{M}$, and in this case it has the same length in both, that is, $\forall b,\ M \vDash b \iff \mathsf{is\_valid}(\overline{M}, b)$ and $\forall b, M \vDash b \implies \mathsf{length}(M, b) = \mathsf{length}(\overline{M}, b)$;
(2) for a valid access in M, a defined value (different from Undef) can be read from it in M if and only if the corresponding location is recorded as initialized in $\overline{M}$, that is, $\forall \kappa, b, \delta,\ M \vDash \kappa \,@\, b, \delta \implies ((\exists v, v \neq \mathsf{Undef} \wedge \mathsf{load}(\kappa, M_1, b, \delta) = \lfloor v \rfloor) \iff \mathsf{is\_initialized}(\kappa, \overline{M}, b, \delta) = \top)$.

Property 7 (representation of access validity). *If $M \triangleright \overline{M}$, then any memory access is valid in M if and only if it is marked as valid in $\overline{M}$. In other words,*

$$\forall \kappa, b, \delta,\ \ M \vDash \kappa \,@\, b, \delta \iff \mathit{is\_valid\_access}(\kappa, \overline{M}, b, \delta) = \top.$$

Proof. This property is a direct consequence of the first condition of the definition. □

Property 8 (preservation of representation across allocation). *Let M_1 be an execution memory and $\overline{M_1}$ an observation memory such that $M_1 \triangleright \overline{M_1}$. Let n be an integer, and $(b, M_2) = \mathit{alloc}(M_1, n)$ the result of allocating n bytes in M_1. Then $M_2 \triangleright \mathit{store\_block}(\overline{M_1}, b, n)$.*

Proof. In order to prove both conditions of Def. 5.5, consider some block b'. If $b' = b$, then the representation conditions are satisfied, by virtue of execution and observation memory axioms expressing properties of newly allocated (or stored) blocks. If $b' \neq b$, then after applying allocation (or block registration) related axioms, the case comes down to the hypothesis $M_1 \triangleright \overline{M_1}$. □

Property 9. *Let M_1 be an execution memory and $\overline{M_1}$ an observation memory such that $M_1 \triangleright \overline{M_1}$. Let b be a block, and $\lfloor M_2 \rfloor = \mathit{free}(M_1, b)$ the result of deallocating b from M_1. Then $M_2 \triangleright \mathit{delete\_block}(\overline{M_1}, b)$.*

Proof. The proof structure is the same as for the previous property: consider some block b', and distinguish between cases $b' = b$ and $b' \neq b$. In the former, the conditions of Def. 5.5 follow from deallocation (resp. block deletion) related axioms, specifying invalidity of deallocated block b. For other blocks $b' \neq b$, the representation conditions are preserved from hypothesis $M_1 \triangleright \overline{M_1}$. □

Property 10 (preservation of representation across store). *Let M_1 be an execution memory and $\overline{M_1}$ an observation memory such that $M_1 \triangleright \overline{M_1}$. Let (b, δ) be a memory location, κ a memory type, $v \neq$ Undef a defined κ-storable value, and $\lfloor M_2 \rfloor = store(\kappa, M_1, b, \delta, v)$ the result of storing v into M_1 at location (b, δ) with type κ. Then $M_2 \triangleright initialize(\kappa, \overline{M_1}, b, \delta)$.*

Proof. The proof is similar to that of the two previous properties. The first condition of Def. 5.5 follows from Axioms (5), (14), (25), (34). For the second condition, consider a valid memory access $M_2 \vDash \kappa' @ b', \delta'$ and let us show that $(\exists v, v \neq \text{Undef} \wedge \text{load}(\kappa', M_1, b', \delta') = \lfloor v \rfloor) \iff \text{is\_initialized}(\kappa', \overline{M}, b', \delta') = \top$. The case $b' \neq b$ and the case $b' = b \wedge (\delta + \text{sizeof}(\kappa) \leqslant \delta' \vee \delta' + \text{sizeof}(\kappa') \leqslant \delta)$, where the writing and reading operations are disjoint, are straightforward from Axioms (11), (31). The remaining non-disjoint cases follow from Axioms (9), (10), (29), (30). □

To ensure that the execution and observation memory models are consistent with one another, the aforementioned notions and properties of Section 5 were formalized and proved in our Coq development.

5.4 Intermediate Lemma

The following lemma will be helpful to prove our main results.

Lemma 5.6. *Let σ be a block permutation and $C = (E, M)$ and $C' = (E', M')$ two isomorphic contexts with an isomorphism induced by σ. If an expression e is evaluated in context C to value v, then e is evaluated in context C' to value $\dot{\sigma}(v)$. In other words, $\forall e, v,\ C \vDash_{\text{e}} e \Downarrow v \implies C' \vDash_{\text{e}} e \Downarrow \dot{\sigma}(v)$. If a left value l is evaluated in context C to memory location (b, δ), then l is evaluated in context C' to memory location $(\sigma(b), \delta)$. In other words, $\forall l, v,\ C \vDash_{\text{lv}} l \Downarrow b, \delta \implies C' \vDash_{\text{lv}} l \Downarrow \sigma(b), \delta$.*

Proof. Assume we have contexts $C = (E, M)$ and $C' = (E', M')$ with $C \sim_\sigma C$. Since the definition of evaluation for expressions and left values is mutually recursive (cf. Fig. 9), both statements of the lemma are proved simultaneously by structural induction.

If e is an integer constant n, it is evaluated to $\text{Int}(n)$ independently of the context, and by definition $\dot{\sigma}$ does not modify integer values.

For a variable x evaluated as a left value, the result follows from the definition of $C \sim_\sigma C'$ (see Def. 5.2) for environments: if $E(x) = \lfloor b \rfloor$ then $E'(x) = \lfloor \sigma(b) \rfloor$.

For evaluation of an expression e that is a left value, the rule is as follows:

$$\text{E-LVAL}\ \frac{E, M \vDash_{\text{lv}} e : \tau \Downarrow b, \delta \qquad \text{load}(\text{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \text{Undef} \qquad \text{kind}(\tau) = \text{kind}(v)}{E, M \vDash_{\text{e}} e \Downarrow v}$$

By induction hypothesis and the definitions of $C \sim_\sigma C'$ (see Def. 5.2) and $\dot{\sigma}$, we have the assumptions of the following derivation which provides the required evaluation:

$$\text{E-LVAL}\ \frac{E', M' \vDash_{\text{lv}} e : \tau \Downarrow \sigma(b), \delta \qquad \text{load}(\text{mtype}(\tau), M', \sigma(b), \delta) = \lfloor \dot{\sigma}(v) \rfloor \qquad \dot{\sigma}(v) \neq \text{Undef} \qquad \text{kind}(\tau) = \text{kind}(\dot{\sigma}(v))}{E', M' \vDash_{\text{e}} e \Downarrow \dot{\sigma}(v)}$$

The remaining cases are treated similarly. □

5.5 Main Theorems

The first theorem characterizes the transformation of instructions by preservation of two invariants: an isomorphism of source and target contexts and a representation of execution memory by the observation memory.

Theorem 5.7 (semantic preservation). *Let s be a source program evaluated in source initial context $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}})$ to final memory state $\widehat{M_\mathrm{f}}$. Let $(E_\mathrm{i}, M_\mathrm{i}, \overline{M_\mathrm{i}})$ be a target initial context such that $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}})$ is isomorphic to $(E_\mathrm{i}, M_\mathrm{i})$ and $\overline{M_\mathrm{i}}$ represents M_i. More formally,*

$$\begin{cases} \widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vDash_\mathrm{s} s \Downarrow \widehat{M_\mathrm{f}}, \\ (\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}}) \sim (E_\mathrm{i}, M_\mathrm{i}), \\ M_\mathrm{i} \triangleright \overline{M_\mathrm{i}}. \end{cases}$$

We assume that $res(n) \notin dom(\widehat{E_\mathrm{i}}) = dom(E_\mathrm{i})$ for all $n \geqslant 0$. Then there exist memory states M_f and $\overline{M_\mathrm{f}}$ such that the evaluation of $[\![s]\!]_\Sigma$ in the target initial context leads to $(M_\mathrm{f}, \overline{M_\mathrm{f}})$, such that $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{f}})$ is isomorphic to $(E_\mathrm{i}, M_\mathrm{f})$ and $\overline{M_\mathrm{f}}$ represents M_f. More formally,

$$\begin{cases} E_\mathrm{i}, M_\mathrm{i}, \overline{M_\mathrm{i}} \vDash_{s'} [\![s]\!]_\Sigma \Downarrow M_\mathrm{f}, \overline{M_\mathrm{f}}, \\ (\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{f}}) \sim (E_\mathrm{i}, M_\mathrm{f}), \\ M_\mathrm{f} \triangleright \overline{M_\mathrm{f}}. \end{cases}$$

As explained in the beginning of Section 5, this theorem can be illustrated by Fig. 25, where the relation $\mathcal{R}$ is defined by the isomorphism and representation conditions. The proof of this theorem proceeds by structural induction on the derivation of the evaluation of the source program instruction. In most cases, it examines the source program operations and constructs their counterpart for its transformation in the target program and establishes the required invariants. An important exception is the evaluation of assertions with predicates, for which the transformation introduces extra instructions. We state a special theorem for predicates. We encode boolean values by integers in the usual way: $\mathsf{Int}(\top) = \mathsf{Int}(1)$ and $\mathsf{Int}(\bot) = \mathsf{Int}(0)$.

Theorem 5.8 (soundness of predicate translation). *Let p be a predicate evaluated in source initial context $(\widehat{E}, \widehat{M})$ to a Boolean value β and $n \geqslant 0$. Let $(E, M, \overline{M})$ be a target starting context such that $res(n) \notin dom(E)$, and let (E^p, M^p) be a subcontext of (E, M) such that $(\widehat{E}, \widehat{M})$ is isomorphic to (E^p, M^p) and $\overline{M}$ represents M^p. More formally,*

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_\mathrm{p} p \Downarrow \beta, \\ (E^p, M^p) \subseteq (E, M), \\ (\widehat{E}, \widehat{M}) \sim (E^p, M^p), \\ M^p \triangleright \overline{M}. \end{cases}$$

Assume in addition that $res(n') \notin dom(\widehat{E})$ for all $n' \geqslant 0$, and $res(n') \notin dom(E)$ for all $n' \geqslant n$. Let us define $(E_\mathrm{i}, M_\mathrm{i}) \triangleq alloc\_var(E, M, res(n), \mathtt{int8})$ such that $E_\mathrm{i}(res(n)) = \lfloor b_{res(n)} \rfloor$, and $\lfloor M'_\mathrm{i} \rfloor \triangleq store(\mathrm{i8}, M_\mathrm{i}, b_{res(n)}, 0, Int(\beta))$. Then there exists a target memory state M_f such that:

- *the translation of p at level n in target initial context $(E_\mathrm{i}, M_\mathrm{i}, \overline{M})$ is evaluated to $(M_\mathrm{f}, \overline{M})$;*
- *the result variable $res(n)$ in target final context $(E_\mathrm{i}, M_\mathrm{f})$ is evaluated to the integer encoding of β;*
- *final memory M_f is equivalent to M'_i.*

In other words:

$$\begin{cases} E_\mathrm{i}, M_\mathrm{i}, \overline{M} \vDash_{s'} [\![p]\!]^n_\Pi \Downarrow M_\mathrm{f}, \overline{M}, \\ E_\mathrm{i}, M_\mathrm{f} \vDash_\mathrm{e} \mathit{res}(n) \Downarrow \mathit{Int}(\beta), \\ M_\mathrm{f} \equiv M'_\mathrm{i}. \end{cases}$$

Compared to Theorem 5.7 (that can be, as we explained in the beginning of Section 5, illustrated by the diagram of Fig. 25), we observe several important differences in the last statement. First, for the source program, we do not evaluate any statement but only a predicate, while for the target program the predicate is translated into a program that is evaluated. Second, as we already mentioned, during the evaluation of the predicate translation, the target context can be richer than the source context: it can contain extra variables and memory blocks (introduced by the translation), hence we have a weaker condition: an isomorphism between the source initial context and a subcontext (E^p, M^p) of the target starting context. Third, final memory M_f is defined up to an equivalence and indicates the expected value for the resulting variable. Overall, this theorem is clearly tuned for the translation scheme of a non-trivial predicate, which starts by introducing a new variable, as illustrated for instance by the translation of `logical_assert`(p) in Fig. 20 and rule S-LET of Fig. 13: from a target starting context $(E, M, \overline{M})$, variable $\mathrm{res}(n)$ is allocated to obtain a new target context $(E_\mathrm{i}, M_\mathrm{i}, \overline{M})$, then the predicate translation is evaluated and the memory of the target final context (with the resulting value of the predicate computed in $\mathrm{res}(n)$) is characterized up to an equivalence. To distinguish target contexts before and after the allocation of $\mathrm{res}(n)$, that is, $(E, M, \overline{M})$ and $(E_\mathrm{i}, M_\mathrm{i}, \overline{M})$, they are referred to, resp., as the target starting context and the target initial context.

Since predicates also involve terms, we will need a similar theorem for terms.

Theorem 5.9 (soundness of term translation). *Let t be a term of type τ evaluated in source initial context $(\widehat{E}, \widehat{M})$ to a value $\widehat{v}$ and $n \geqslant 0$. Let $(E, M, \overline{M})$ be a target starting context such that $\mathit{res}(n) \notin \mathit{dom}(E)$, and let (E^p, M^p) be a subcontext of (E, M) such that $(\widehat{E}, \widehat{M})$ is isomorphic to (E, M^p) with an isomorphism induced by a permutation σ, and $\overline{M}$ represents M^p. In other words,*

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_\mathrm{t} t \Downarrow \widehat{v}, \\ (E^p, M^p) \subseteq (E, M), \\ (\widehat{E}, \widehat{M}) \sim_\sigma (E^p, M^p), \\ M^p \triangleright \overline{M}. \end{cases}$$

Assume in addition that $\mathit{res}(n') \notin \mathit{dom}(\widehat{E})$ for all $n' \geqslant 0$, and $\mathit{res}(n') \notin \mathit{dom}(E)$ for all $n' \geqslant n$. Let us define $(E_\mathrm{i}, M_\mathrm{i}) \triangleq \mathit{alloc\_var}(E, M, \mathit{res}(n), \tau)$ such that $E_\mathrm{i}(\mathit{res}(n)) = \lfloor b_{\mathit{res}(n)} \rfloor$, and $\lfloor M'_\mathrm{i} \rfloor \triangleq \mathit{store}(\mathit{mtype}(\tau), M_\mathrm{i}, b_{\mathit{res}(n)}, 0, v)$, where $v \triangleq \dot{\sigma}(\widehat{v})$. Then there exists a target memory state M_f such that:

- *the translation of t at level n in target initial context $(E_\mathrm{i}, M_\mathrm{i}, \overline{M})$ is evaluated to $(M_\mathrm{f}, \overline{M})$;*
- *the result variable $\mathit{res}(n)$ in target final context $(E_\mathrm{i}, M_\mathrm{f})$ is evaluated to v;*
- *final memory M_f is equivalent to M'_i.*

In other words:

$$\begin{cases} E_\mathrm{i}, M_\mathrm{i}, \overline{M} \vDash_{s'} [\![t]\!]^n_\mathrm{T} \Downarrow M_\mathrm{f}, \overline{M}, \\ E_\mathrm{i}, M_\mathrm{f} \vDash_\mathrm{e} \mathit{res}(n) \Downarrow v, \\ M_\mathrm{f} \equiv M'_\mathrm{i}. \end{cases}$$

We state the three theorems and present their (partial) proofs in the order that makes it easier for the reader to see why we need them. Logically, the proof of the three theorems proceeds in the inverse order: Theorem 5.9 is proved first (without relying on Theorems 5.7 and 5.8), then

Theorem 5.8 is proved using Theorem 5.9 (without relying on Theorem 5.7), and finally Theorem 5.7 is proved using Theorem 5.8. A machine-checked Coq proof of these results is left as future work.

5.6 Proof of Theorem 5.7

To prove Theorem 5.7, we proceed by structural induction on the evaluation of the source program s and consider various cases of the final rule in the evaluation of s. Assuming that the result holds for the previous steps of evaluation, we construct the evaluation of the transformed program $[\![s]\!]_\Sigma$ and prove that it satisfies the required properties.

We detail two representative cases, for rules S-malloc and S-logical-assert. The proof of other cases is similar and is omitted. As explained above, we assume here that Theorem 5.8 holds (its proof will be presented below and will not rely on Theorem 5.7).

Case S-malloc. Assume that the final rule in the source program evaluation is S-malloc, that is, the source program s has the form $l = \texttt{malloc}(e)$; and the source program evaluation has the following form (cf. rule S-malloc in Fig. 13):

$$\text{S'-malloc}\ \frac{\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vdash_\mathrm{e} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{alloc}(\widehat{M_\mathrm{i}}, n) = (\widehat{b'}, \widehat{M'_\mathrm{i}}) \qquad \widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vdash_\mathrm{lv} l : \tau* \Downarrow (\widehat{b}, \delta) \qquad \mathsf{store}(\mathsf{mtype}(\tau*), \widehat{M'_\mathrm{i}}, \widehat{b}, \delta, \mathsf{Ptr}(\widehat{b'}, 0)) = \lfloor \widehat{M_\mathrm{f}} \rfloor}{\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vdash_\mathrm{s} l = \texttt{malloc}(e); \Downarrow \widehat{M_\mathrm{f}}}$$

The translation $[\![l = \texttt{malloc}(e);]\!]_\Sigma$ has the form (cf. Fig. 20):

```
1 l = malloc(e);
2 store_block(l,e);
3 initialize(&l);
```

To save room in derivations (here and below), we will use the following notation. The target context after line k of the translation will be denoted by $\mathscr{C}_k = (E_k, M_k, \overline{M_k})$. Accordingly, $\mathscr{C}_0 = (E_0, M_0, \overline{M_0})$ will denote the context before the first line.

Let us construct an evaluation of the translation $[\![l = \texttt{malloc}(e);]\!]_\Sigma$ as the following derivation (by applying suitable rules of Fig. 13 and 19, and for suitable values of contexts that we describe below):

$$\dfrac{\dfrac{\begin{array}{c}(E_0, M_0) \vdash_\mathrm{e} e \Downarrow \mathsf{Int}(n) \\ \mathsf{alloc}(M_0, n) = (b', M'_0) \\ (E_0, M_0) \vdash_\mathrm{lv} l : \tau* \Downarrow (b, \delta) \\ \mathsf{store}(\mathsf{mtype}(\tau*), M'_0, b, \delta, \mathsf{Ptr}(b', 0)) = \lfloor M_1 \rfloor\end{array}}{\mathscr{C}_0 \vdash_\mathrm{s} l = \texttt{malloc}(e); \Downarrow \mathscr{C}_1} \qquad \dfrac{\dfrac{\cdots}{\mathscr{C}_1 \vdash_\mathrm{s} \texttt{store\_block}(l, e); \Downarrow \mathscr{C}_2} \qquad \dfrac{\cdots}{\mathscr{C}_2 \vdash_\mathrm{s} \texttt{initialize}(\&l); \Downarrow \mathscr{C}_3}}{\mathscr{C}_1 \vdash_\mathrm{s} \texttt{store\_block}(l, e); \texttt{initialize}(\&l); \Downarrow \mathscr{C}_3}}{\mathscr{C}_0 \vdash_\mathrm{s} [\![l = \texttt{malloc}(e);]\!]_\Sigma \Downarrow M_3, \overline{M_3}}$$

We define the contexts $\mathscr{C}_k$ and block b' as follows:

$E_0 \triangleq E_\mathrm{i}$	$M_0 \triangleq M_\mathrm{i}$	$\overline{M_0} \triangleq \overline{M_\mathrm{i}}$
	$(b', M'_0) \triangleq \mathsf{alloc}(M_0, n)$	
$E_1 \triangleq E_0$	$\lfloor M_1 \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M'_0, b, \delta, \mathsf{Ptr}(b', 0))$	$\overline{M_1} \triangleq \overline{M_0}$
$E_2 \triangleq E_1$	$M_2 \triangleq M_1$	$\overline{M_2} \triangleq \mathsf{store\_block}(\overline{M_1}, b', 0, n)$
$E_3 \triangleq E_2$	$M_3 \triangleq M_2$	$\overline{M_3} \triangleq \mathsf{initialize}(\tau*, \overline{M_2}, b, \delta)$

By assumption, $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}}) \sim_\sigma (E_0, M_0)$. By Lemma 5.6 and the definition of $\dot{\sigma}$ (Def. 5.1), we see indeed that expression e is evaluated to $\mathsf{Int}(n)$, and that left-value l is evaluated to (b, δ), where $b \triangleq \sigma(\widehat{b})$. It follows from the rules of Fig. 9 that $\widehat{b} \in \mathsf{supp}(\widehat{M_\mathrm{i}})$ because since this block is the result of evaluation of a left-value, it is either associated to a variable (hence a valid block, since our contexts

are well-formed) or referred to by an existing pointer. Similarly, we also have $b \in \mathrm{supp}(M_0)$. By Axiom (16), $\widehat{b'} \notin \mathrm{supp}(\widehat{M_\mathrm{i}})$ and $b' \notin \mathrm{supp}(M_0)$, therefore we necessarily have $\widehat{b} \neq \widehat{b'}$ and $b \neq b'$. By Property 2, since $b' \notin \mathrm{supp}(M_0)$, we also have $\sigma^{-1}(b') \notin \mathrm{supp}(\widehat{M_\mathrm{i}})$, therefore we necessarily have $\widehat{b} \neq \sigma^{-1}(b')$.

Further, by Property 3, the isomorphism of contexts is preserved after the allocation, hence we have $(\widehat{E_\mathrm{i}}, \widehat{M'_\mathrm{i}}) \sim_{\sigma'} (E_0, M'_0)$ with permutation $\sigma' = \sigma \circ (\widehat{b'} \leftrightarrow \sigma^{-1}(b'))$. In particular, by definition of σ' we have $\sigma'(\widehat{b'}) = b'$ and $\sigma'(\widehat{b}) = b$ (since $\widehat{b} \neq \widehat{b'}$, $\widehat{b} \neq \sigma^{-1}(b')$ and $\sigma(\widehat{b}) = b$). The two store operations $\mathrm{store}(\mathrm{mtype}(\tau*), \widehat{M'_\mathrm{i}}, \widehat{b}, \delta, \mathrm{Ptr}(\widehat{b'}, 0))$ and $\mathrm{store}(\mathrm{mtype}(\tau*), M'_0, b, \delta, \mathrm{Ptr}(b', 0))$ write two corresponding values (since $\dot{\sigma'}(\mathrm{Ptr}(\widehat{b'}, 0)) = \mathrm{Ptr}(b', 0)$) into two corresponding locations (since $\sigma'(\widehat{b}) = b$), hence by Property 5 we deduce that $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{f}}) \sim_{\sigma'} (E_1, M_1) = (E_3, M_3)$.

The remaining steps update the observation memory without changing the execution memory. Since $M_0 \triangleright \overline{M_0}$ by assumption, we can deduce by Property 8 that $M'_0 \triangleright \overline{M_2}$, and further by Property 10 that $M_3 = M_1 \triangleright \overline{M_3}$, which finishes the proof of the required statement in this case.

Cas S-LOGICAL-ASSERT. This case relies on the evaluation rule for `logical_assert(`p`)`, the translation of `logical_assert(`p`)` and Theorem 5.8 (applied for level $n = 0$).

The source program evaluation has the following form (cf. S-LOGICAL-ASSERT in Fig. 13):

$$\textsc{S-logical-assert}\ \frac{\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vDash_\mathrm{p} p \Downarrow \top}{\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}} \vDash_\mathrm{s} \texttt{logical\_assert}(p); \Downarrow \widehat{M_\mathrm{f}}}$$

where $\widehat{M_\mathrm{f}} = \widehat{M_\mathrm{i}}$.

The translation $[\![\texttt{logical\_assert}(p)]\!]_\Sigma$ has the form (cf. Fig. 20):

```
1 let res(0) : int8 in
2   [[p]]^0_Π
3   assert(res(0));
4 end
```

Recall that we maintain the same notation convention $\mathscr{C}_k = (E_k, M_k, \overline{M_k})$ for target context before line k as in the previous case. Let us construct an evaluation of the translation $[\![\texttt{logical\_assert}(p)]\!]_\Sigma$ as the following derivation (by applying suitable rules of Fig. 13 and 19, and for suitable values of contexts that we describe below):

$$\textsc{S'-let}\ \frac{(E_1, M_1) = \mathrm{alloc\_var}(E_0, M_0, \mathrm{res}(0), \mathtt{int8}) \qquad x \notin \mathrm{dom}(E_1) \qquad \lfloor M_4 \rfloor = \mathrm{dealloc\_var}(E_3, M_3, \mathrm{res}(0))}{\mathscr{C}_0 \vDash_{s'} \texttt{let res}(0) : \texttt{int8 in } [\![p]\!]^0_\Pi\ \texttt{assert}(\mathrm{res}(0)); \texttt{end} \Downarrow M_4, \overline{M_4}}$$

$$\textsc{S'-seq}\ \frac{\mathscr{C}_1 \vDash_{s'} [\![p]\!]^0_\Pi \Downarrow M_2, \overline{M_2} \qquad \textsc{S'-assert}\ \dfrac{E_2, M_2 \vDash_\mathrm{e} \mathrm{res}(0) \Downarrow \mathrm{Int}(1) \qquad 1 \neq 0}{\mathscr{C}_2 \vDash_{s'} \texttt{assert}(\mathrm{res}(0)); \Downarrow M_3, \overline{M_3}}}{\mathscr{C}_1 \vDash_{s'} [\![p]\!]^0_\Pi\ \texttt{assert}(\mathrm{res}(0)); \Downarrow M_3, \overline{M_3}}$$

We define the contexts $\mathscr{C}_k$ and block $b_{\mathrm{res}(0)}$ as follows:

E_0	$\triangleq$	E_i	M_0	$\triangleq$	M_i	$\overline{M_0}$	$\triangleq$	$\overline{M_\mathrm{i}}$
E_1	$\triangleq$	$E_0 \sqcup (\mathrm{res}(0), b_{\mathrm{res}(0)})$	$(b_{\mathrm{res}(0)}, M_1)$	$\triangleq$	$\mathrm{alloc}(M_0, \mathrm{sizeof}(\mathtt{int8}))$	$\overline{M_1}$	$\triangleq$	$\overline{M_0}$
E_2	$\triangleq$	E_1	M_2	such that	$\mathscr{C}_1 \vDash_{s'} [\![p]\!]^0_\Pi \Downarrow M_2, \overline{M_2}$	$\overline{M_2}$	$\triangleq$	$\overline{M_1}$
E_3	$\triangleq$	E_2	M_3	$\triangleq$	M_2	$\overline{M_3}$	$\triangleq$	$\overline{M_2}$
E_4	$\triangleq$	E_0	$\lfloor M_4 \rfloor$	$\triangleq$	$\mathrm{free}(M_3, b_{\mathrm{res}(0)})$	$\overline{M_4}$	$\triangleq$	$\overline{M_3}$

Let us show the existence of M_2 and M_4. Theorem 5.8 can be applied for level $n = 0$, $\beta = \top$, source initial context $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}})$, target starting context $\mathscr{C}_0 = (E_0, M_0, \overline{M_0})$ and a trivial subcontext

$(E^p, M^p) = (E_0, M_0)$, which verify the assumptions of the theorem. We deduce from Theorem 5.8 for target initial context $(E_1, M_1, \overline{M_1})$ and $\overline{M_2} \triangleq \overline{M_1}$ that there exists M_2 such that

$$\begin{cases} E_1, M_1, \overline{M_1} \vDash_{s'} [\![p]\!]^0_\Pi \Downarrow M_2, \overline{M_2}, \\ E_1, M_2 \vDash_e \mathsf{res}(0) \Downarrow \mathsf{Int}(1), \\ M_2 \equiv M_1', \end{cases}$$

where M_1' is defined by $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathrm{i}\,8, M_1, b_{\mathsf{res}(0)}, 0, \mathsf{Int}(1))$. The first of these properties is also used as an assumption in the application of rule S'-SEQ in the derivation above.

Further, we have $M_1 \vDash b_{\mathsf{res}(0)}$ by Axiom (1), $M_3 = M_2 \equiv M_1'$ and $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathrm{i}8, M_1, b_{\mathsf{res}(0)}, 0, \mathsf{Int}(1))$, hence it follows from Axiom (5) that $M_1' \vDash b_{\mathsf{res}(0)}$ and by Def. 5.4 that $M_3 \vDash b_{\mathsf{res}(0)}$. Therefore, by Axiom (19) M_4 is also well-defined.

To deduce the required statement of Theorem 5.7, it remains to show that M_4 and $\overline{M_4}$ satisfy the properties $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{f}}) \sim (E_4, M_4)$ and $M_4 \triangleright \overline{M_4}$. Since $\widehat{M_\mathrm{f}} = \widehat{M_\mathrm{i}}$ and $E_4 = E_0$, the first property can be rewritten as $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}}) \sim (E_0, M_4)$. The required properties can be deduced from the assumptions of Theorem 5.7 $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}}) \sim (E_\mathrm{i}, M_\mathrm{i}) = (E_0, M_0)$ and $M_0 \triangleright \overline{M_\mathrm{i}} = \overline{M_0}$. Intuivively, memory M_4 is constructed from M_0 (up to equivalence at step M_2) by allocating a bloc b_1 for variable $\mathsf{res}(0)$, writing its value and deallocating block b_1. These operations keep untouched other existing valid blocks and their contents, which allows us to deduce $(\widehat{E_\mathrm{i}}, \widehat{M_\mathrm{i}}) \sim (E_0, M_4)$. They have no impact on representation by $M_0 = M_4$, which allows us to deduce $M_4 \triangleright \overline{M_4}$. The detailed verification of these properties is straightforward using the definitions and is left to the reader. □

5.7 Proof of Theorem 5.8

To prove Theorem 5.8, we proceed by structural induction on the evaluation of the source program predicate p and consider various cases of the final rule in its evaluation. Assuming that the result holds for the previous steps of evaluation, we construct the evaluation of the translation $[\![p]\!]^n_\Pi$ and prove that it satisfies the required properties. As previously, to save room in derivations, the target context after line k of the translation will be denoted by $\mathscr{C}_k = (E_k, M_k, \overline{M})$. From target starting context $(E, M, \overline{M})$, the target initial context $(E_\mathrm{i}, M_\mathrm{i}, \overline{M}) = (E_0, M_0, \overline{M})$ is defined by $(E_0, M_0) \triangleq \mathsf{alloc\_var}(E, M, \mathsf{res}(n), \texttt{int8})$ according to the statement, such that $E_\mathrm{i}(\mathsf{res}(n)) = \lfloor b_{\mathsf{res}(n)} \rfloor$, and $M_\mathrm{i}' = M_0'$ is defined by $\lfloor M_\mathrm{i}' \rfloor = \lfloor M_0' \rfloor \triangleq \mathsf{store}(\mathrm{i}8, M_\mathrm{i}, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(\beta))$.

The induction will proceed from predicate p to its subpredicate(s) (if any) with the same source initial context $(\widehat{E}, \widehat{M})$, the same subcontext (E^p, M^p) of the target starting context and the same observation memory $\overline{M}$ inside the target starting context. Indeed, in the source program, the environment and execution memory are not modified during a predicate evaluation. In the translated program—where the predicate is translated into additional instructions—the environment and execution memory are modified for variables and blocks added by the translation, but the observation memory is only read and is not modified.

We detail three representative cases, for rules P-TRUE, P-CONJ-LEFT and S-VALID. The proof of other cases is similar and is omitted. As explained earlier, we assume here that Theorem 5.9 holds (its proof does not rely on Theorem 5.8).

Case P-TRUE. Assume that the final rule in the source program evaluation is P-TRUE, that is, the source program evaluation has the following form (cf. rule P-TRUE in Fig. 11):

$$\text{P-TRUE}\ \frac{}{\widehat{E}, \widehat{M} \vDash_p \backslash\texttt{true} \Downarrow \top}$$

The translation $[\![\backslash\texttt{true}]\!]^n_\Pi$ has the form (cf. Fig. 21):

```
1  res(n) = 1;
```

Let us construct an evaluation of the translation $[\![\texttt{\textbackslash true}]\!]^n_\Pi$ as the following derivation (by applying suitable rules of Fig. 9 and 13 and for suitable values of contexts that we describe below):

$$\textsc{s'-assign}\ \frac{E_0, M_0 \vDash_{\mathrm{e}} 1 : \texttt{int8} \Downarrow \mathsf{Int}(1) \qquad \textsc{lv-var}\ \dfrac{E_0(\mathsf{res}(n)) = \lfloor b_{\mathsf{res}(n)} \rfloor}{E_0, M_0 \vDash_{\mathrm{lv}} \mathsf{res}(n) : \texttt{int8} \Downarrow b_{\mathsf{res}(n)}, 0} \qquad \mathsf{store}(\mathsf{i8}, M_0, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(1)) = \lfloor M_1 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathsf{res}(n) = 1; \Downarrow M_1, \overline{M}}$$

We define the contexts $\mathscr{C}_k = (E_k, M_k, \overline{M})$ as follows (recall that the observation memory remains unchanged):

$$\begin{array}{lcl|lcl} E_0 & \triangleq & E_{\mathrm{i}} & M_0 & \triangleq & M_{\mathrm{i}} \\ E_1 & \triangleq & E_0 & \lfloor M_1 \rfloor & \triangleq & \mathsf{store}(\mathsf{i8}, M_{\mathrm{i}}, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(1)) \end{array}$$

By construction, $M_{\mathrm{i}} \vDash \mathsf{i8} \mathbin{@} b_{\mathsf{res}(n)}, 0$, so memory state M_1 is well-defined by Axiom (17).

Let show that the required properties of Theorem 5.8 for $M_{\mathrm{f}} \triangleq M_1$:

$$\begin{cases} E_0, M_0, \overline{M} \vDash_{s'} [\![\texttt{\textbackslash true}]\!]^n_\Pi \Downarrow M_1, \overline{M}, \\ E_0, M_1 \vDash_{\mathrm{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(\beta), \\ M_1 \equiv M'_0. \end{cases}$$

The derivation above gives the required evaluation for the translation $[\![\texttt{\textbackslash true}]\!]^n_\Pi$. The second property is shown by the following derivation, directly based on definitions:

$$\textsc{e-lval}\ \frac{E_{\mathrm{i}}, M_1 \vDash_{\mathrm{lv}} \mathsf{res}(n) : \texttt{int8} \Downarrow b_{\mathsf{res}(n)}, 0 \qquad \mathsf{load}(\mathsf{i8}, M_1, b_{\mathsf{res}(n)}, 0) = \lfloor \mathsf{Int}(1) \rfloor \qquad \mathsf{Int}(1) \neq \mathsf{Undef} \qquad \mathsf{kind}(\texttt{int8}) = \mathsf{kind}(\mathsf{Int}(1))}{E_{\mathrm{i}}, M_1 \vDash_{\mathrm{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(1)}$$

Finally, $M_{\mathrm{f}} \equiv M'_{\mathrm{i}}$ since $\lfloor M_{\mathrm{f}} \rfloor = \mathsf{store}(\mathsf{i8}, M_{\mathrm{i}}, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(1)) = \lfloor M'_{\mathrm{i}} \rfloor$.

Cas P-CONJ-LEFT. Assume that the final rule in the source program evaluation is P-CONJ-LEFT, that is, the source program evaluation has the following form (cf. rule P-CONJ-LEFT in Fig. 12):

$$\textsc{p-conj-left}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_1 \Downarrow \bot}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow \bot}$$

The translation $[\![p_1 \wedge p_2]\!]^n_\Pi$ has the form (cf. Fig. 21):

```
1  let res(n + 1) : int8 in
2    [[p1]]^{n+1}_Π
3    if (res(n + 1))
4    then let res(n + 2) : int8 in
5      [[p2]]^{n+2}_Π
6      res(n) = res(n + 2);
7    end else
8      res(n) = 0;
9  end
```

Let us construct an evaluation of the translation $[\![p_1 \wedge p_2]\!]^n_\Pi$ as the following derivation (by applying suitable rules of Fig. 9, 13 and 14 and for suitable values of contexts that we describe below):

$$\text{S'-LET}\ \frac{\text{alloc\_var}(E_0, M_0, \text{res}(n+1), \texttt{int8}) = E_1, M_1 \qquad \text{res}(n+1) \notin \text{dom}(E_0) \qquad \text{dealloc\_var}(E_8, M_8, \text{res}(n+1)) = \lfloor M_9 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \texttt{let res}(n+1) : \texttt{int8 in} \ldots \texttt{end} \Downarrow M_9, \overline{M}}$$

$$\text{S'-SEQ}\ \frac{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]^{n+1}_\Pi \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]^{n+1}_\Pi\ \texttt{if}(\text{res}(n+1)) \ldots \Downarrow M_8, \overline{M}}$$

$$\text{S'-IF-FALSE}\ \frac{E_2, M_2 \vDash_e \text{res}(n+1) \Downarrow \text{Int}(0)}{E_2, M_2, \overline{M} \vDash_{s'} \texttt{if}(\text{res}(n+1)) \ldots \texttt{else res}(n) = 0; \Downarrow M_8, \overline{M}}$$

$$\text{S'-ASSIGN}\ \frac{\text{store}(\text{i8}, M_2, b_{\text{res}(n)}, 0, \text{Int}(0)) = \lfloor M_8 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \text{res}(n) = 0; \Downarrow M_8, \overline{M}}$$

$$\text{LV-VAR}\ \frac{E_2(\text{res}(n)) = \lfloor b_{\text{res}(n)} \rfloor}{E_2, M_2 \vDash_{lv} \text{res}(n) \Downarrow b_{\text{res}(n)}, 0} \qquad E_2, M_2 \vDash_e 0 \Downarrow \text{Int}(0)$$

We define the contexts $\mathscr{C}_k = (E_k, M_k, \overline{M})$ and block $b_{\text{res}(n+1)}$ as follows (notice that the then branch of the conditional in the translation is not executed so only $\mathscr{C}_0, \mathscr{C}_1, \mathscr{C}_2, \mathscr{C}_8, \mathscr{C}_9$ are relevant, and that the observation memory remains unchanged):

E_0	$\triangleq$	E_i	M_0	$\triangleq$	M_i
E_1	$\triangleq$	$E_0 \sqcup (\text{res}(n+1), b_{\text{res}(n+1)})$	$(b_{\text{res}(n+1)}, M_1)$	$\triangleq$	$\text{alloc}(M_0, \text{sizeof}(\texttt{int8}))$
E_2	$\triangleq$	E_1	M_2	such that	$E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]^{n+1}_\Pi \Downarrow M_2, \overline{M}$
E_8	$\triangleq$	E_2	$\lfloor M_8 \rfloor$	$\triangleq$	$\text{store}(\text{i8}, M_2, b_{\text{res}(n)}, 0, \text{Int}(0))$
E_9	$\triangleq$	E_0	$\lfloor M_9 \rfloor$	$\triangleq$	$\text{free}(M_8, b_{\text{res}(n+1)})$

Let us show that all these elements are well-defined. To show the existence of M_2, we use the induction hypothesis by applying the statement of Theorem 5.8 for predicate p_1, level $n+1$, target starting context $(E_0, M_0, \overline{M})$, target initial context $(E_1, M_1, \overline{M})$, the same source initial context $(\widehat{E}, \widehat{M})$ and the same subcontext (E^p, M^p). The hypotheses for this application of the theorem are satisfied, in particular, we have:

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_\text{p} p_1 \Downarrow \bot, \\ (E^p, M^p) \subseteq (E_0, M_0), \\ (\widehat{E}, \widehat{M}) \sim (E^p, M^p), \\ M^p \triangleright \overline{M}. \end{cases}$$

where the property $(E^p, M^p) \subseteq (E_0, M_0)$ can be easily deduced from the assumption $(E^p, M^p) \subseteq (E, M)$ by construction of (E_0, M_0) and by definitions. Hence, we deduce from this application of the theorem the existence of M_2 such that:

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]^{n+1}_\Pi \Downarrow M_2, \overline{M}, \\ E_2, M_2 \vDash_\text{e} \text{res}(n+1) \Downarrow \text{Int}(0), \\ M_2 \equiv M_1', \end{cases}$$

where $\lfloor M_1' \rfloor \triangleq \text{store}(\text{i8}, M_1, b_{\text{res}(n+1)}, 0, \text{Int}(0))$. The first and second of these properties are also used as assumptions in the application of rules S'-SEQ and S'-IF-FALSE above.

Let us show that M_8 and M_9 are well-defined. By construction, $M_0 \vDash \text{i8} \mathbin{@} b_{\text{res}(n)}, 0$, and after the following steps—namely, allocation of block $b_{\text{res}(n+1)}$ to obtain M_1 and storing a value for $b_{\text{res}(n+1)}$ to obtain (up to equivalence) M_2—we also have $M_2 \vDash \text{i8} \mathbin{@} b_{\text{res}(n)}, 0$, so memory state M_8 is well-defined by Axiom (17). By construction, $M_1 \vDash b_{\text{res}(n+1)}$, and the following steps—namely, storing a value for $b_{\text{res}(n+1)}$ to obtain (up to equivalence) M_2, then storing a value for $b_{\text{res}(n)}$ to obtain M_8—do not impact this validity, so $M_8 \vDash b_{\text{res}(n+1)}$, hence memory state M_9 is well-defined by Axiom (19). The verification of details is straightforward by definitions.

We define the required final memory as $M_{\dagger} \triangleq M_9$. To finish the proof, we have to show:

$$\begin{cases} E_0, M_0, \overline{M} \vDash_{s'} [\![p_1 \wedge p_2]\!]^n_{\Pi} \Downarrow M_9, \overline{M}, \\ E_0, M_9 \vDash_{e} \mathsf{res}(n) \Downarrow \mathsf{Int}(0), \\ M_9 \equiv M'_0, \end{cases}$$

where $\lfloor M'_0 \rfloor \triangleq \mathsf{store}(\mathsf{i8}, M_0, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(0))$. The first property is shown by the derivation above. The last two properties are justified by the fact that M_9 is constructed from M_0 by allocation of block $b_{\mathsf{res}(n+1)}$ to obtain M_1, storing a value for $b_{\mathsf{res}(n+1)}$ to obtain (up to equivalence) M_2, storing a value 0 for $\mathsf{res}(n)$ to obtain M_8, and finally removing block $b_{\mathsf{res}(n+1)}$. The detailed verification of these properties is straightforward using the definitions and is left to the reader.

Remark. Notice that our axioms do not allow us to deduce that $M_9 = M'_0$ but only their observational equivalence $M_9 \equiv M'_0$ (which preserves the set of valid blocks, their sizes and contents), and only this observational equivalence is required in the theorem. We admit that in a real-life implementation, memory datastructures can be different for M_9 whose construction from M_0 goes through additional steps (e.g. allocation and deallocation of block $b_{\mathsf{res}(n+1)}$) compared to the construction of M'_0. This approach leaves more freedom for an actual implementation to choose, for instance, the next block identifier to be allocated, the way to maintain (an overapproximation of) the support of the execution memory, the moment to call a garbage collector (to recompute this overapproximation more precisely), etc. This choice leaves more freedom for a real-life tool in implementing the underlying memory model.

Case P-VALID. Assume that the final rule in the source program evaluation is P-VALID, that is, the source program evaluation has the following form (cf. rule P-VALID in Fig. 11, where t must be of a pointer type, say, $\tau_t = \tau*$):

$$\text{P-VALID}\ \frac{\widehat{E}, \widehat{M} \vDash_{t} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad \widehat{M} \vDash \mathsf{mtype}(\tau)\ @\ \widehat{b}, \widehat{\delta}}{\widehat{E}, \widehat{M} \vDash_{p} \backslash\mathtt{valid}(t) \Downarrow \top}$$

The translation $[\![\backslash\mathtt{valid}(t)]\!]^n_{\Pi}$ has the form (cf. Fig. 22):

```
1 let res(n+1) : τ* in
2   [[t]]^{n+1}_T
3   res(n) = is_valid(res(n+1));
4 end
```

Let us construct an evaluation of the translation $[\![\backslash\mathtt{valid}(t)]\!]^n_{\Pi}$ as the following derivation (by applying suitable rules of Fig. 9, 14 and 19 and for suitable values of contexts that we describe below):

$$\text{S'-LET}\ \frac{\text{S'-SEQ}\ \dfrac{\text{S'-IS-VALID}\ \dfrac{\text{LV-VAR}\ \dfrac{E_2(\mathsf{res}(n)) = \lfloor b_{\mathsf{res}(n)} \rfloor}{E_2, M_2 \vDash_{lv} \mathsf{res}(n) \Downarrow b_{\mathsf{res}(n)}, 0} \qquad E_2, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta) \\ \mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \top \\ \mathsf{store}(\mathsf{i8}, M_2, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(1)) = \lfloor M_3 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}} \\ E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]^{n+1}_{\mathrm{T}} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]^{n+1}_{\mathrm{T}}\ \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}} \\ \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau*) = (E_1, M_1) \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let}\ \mathsf{res}(n+1) : \tau*\ \mathtt{in} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

We define the contexts $\mathscr{C}_k = (E_k, M_k, \overline{M})$ and block $b_{\mathsf{res}(n+1)}$ as follows:

$$\begin{array}{lcl|rcl} E_0 & \triangleq & E_{\mathsf{i}} & M_0 & \triangleq & M_{\mathsf{i}} \\ E_1 & \triangleq & E_0 \sqcup (\mathsf{res}(n+1), b_{\mathsf{res}(n+1)}) & (b_{\mathsf{res}(n+1)}, M_1) & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau*)) \\ E_2 & \triangleq & E_1 & M_2 & \text{such that} & E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_3 & \triangleq & E_2 & \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathsf{i8}, M_2, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(1)) \\ E_4 & \triangleq & E_0 & \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_{\mathsf{res}(n+1)}) \end{array}$$

Let us show that all these elements are well-defined. To show the existence of M_2, we use Theorem 5.9 for term t, level $n+1$, target starting context $(E_0, M_0, \overline{M})$, target initial context $(E_1, M_1, \overline{M})$, the same source initial context $(\widehat{E}, \widehat{M})$ and the same subcontext (E^p, M^p). The hypotheses for this application of Theorem 5.9 are satisfied, in particular, we have for some permutation σ:

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_{\mathsf{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}), \\ (E^p, M^p) \subseteq (E_0, M_0), \\ (\widehat{E}, \widehat{M}) \sim_\sigma (E^p, M^p), \\ M^p \triangleright \overline{M}, \end{cases}$$

where the property $(E^p, M^p) \subseteq (E_0, M_0)$ can be easily deduced from the assumption $(E^p, M^p) \subseteq (E, M)$ by construction of (E_0, M_0) and by definitions. Hence, we deduce from this application of the theorem the existence of M_2 such that:

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M}, \\ E_2, M_2 \vDash_{\mathsf{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta), \\ M_2 \equiv M_1', \end{cases}$$

where $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_1, b_{\mathsf{res}(n+1)}, 0, \mathsf{Ptr}(b, \delta))$, where $\mathsf{Ptr}(b, \delta) = \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, that is, $b = \sigma(\widehat{b})$ and $\delta = \widehat{\delta}$. The first and second of these properties are also used as assumptions in the application of rules S'-SEQ and S'-IS-VALID above.

Let us show that M_3 and M_4 are well-defined. By construction, $M_0 \vDash \mathsf{i8} \mathbin{@} b_{\mathsf{res}(n)}, 0$, and after the following steps—namely, allocation of block $b_{\mathsf{res}(n+1)}$ to obtain M_1 and storing a value for $b_{\mathsf{res}(n+1)}$ to obtain (up to equivalence) M_2—we also have $M_2 \vDash \mathsf{i8} \mathbin{@} b_{\mathsf{res}(n)}, 0$, so memory state M_3 is well-defined by Axiom (17). By construction, $M_1 \vDash b_{\mathsf{res}(n+1)}$, and the following steps—namely, storing a value for $b_{\mathsf{res}(n+1)}$ to obtain (up to equivalence) M_2, then storing a value for $b_{\mathsf{res}(n)}$ to obtain M_8—do not impact this validity, so $M_3 \vDash b_{\mathsf{res}(n+1)}$, hence memory state M_4 is well-defined by Axiom (19). The verification of details is straightforward by definitions.

By Property 1 we deduce from the assumption for the source language $\widehat{M} \vDash \mathsf{mtype}(\tau) \mathbin{@} \widehat{b}, \widehat{\delta}$ and $(\widehat{E}, \widehat{M}) \sim_\sigma (E^p, M^p)$ that $M^p \vDash \mathsf{mtype}(\tau) \mathbin{@} b, \delta$. Since $M^p \triangleright \overline{M}$, it follows by Property 7 that $\mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \top$, which is used as an assumption in the application of rule S'-IS-VALID above.

We define the required final memory as $M_{\mathsf{f}} \triangleq M_4$. To finish the proof, we have to show:

$$\begin{cases} E_0, M_0, \overline{M} \vDash_{s'} [\![\backslash\mathtt{valid}(t)]\!]_{\Pi}^{n} \Downarrow M_4, \overline{M}, \\ E_0, M_4 \vDash_{\mathsf{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(1), \\ M_4 \equiv M_0', \end{cases}$$

where $\lfloor M_0' \rfloor \triangleq \mathsf{store}(\mathsf{i8}, M_0, b_{\mathsf{res}(n)}, 0, \mathsf{Int}(0))$. The first property is shown by the derivation above. The last two properties are justified by the fact that M_4 is constructed from M_0 by allocation of block $b_{\mathsf{res}(n+1)}$ to obtain M_1, storing a value for $b_{\mathsf{res}(n+1)}$ to obtain (up to equivalence) M_2, storing a value 0 for $\mathsf{res}(n)$ to obtain M_3, and finally removing block $b_{\mathsf{res}(n+1)}$. The detailed verification of these properties is straightforward using the definitions and is left to the reader. □

5.8 Proof of Theorem 5.9

The proof of Theorem 5.9 proceeds similarly to that of Theorem 5.8 and does not rely on Theorem 5.7 and Theorem 5.8. It is left to the reader and is omitted here. □

5.9 Soundness of a Runtime Assertion Checker

Assume we have a runtime assertion checker implementing the translation presented in Sec. 4. The evaluation of (the first instruction of) a given program starts with an empty context, and the runtime assertion checker starts the evaluation of (the first instruction of) the translated program with an empty observation memory. Let us show how Theorem 5.7 can be applied to show its soundness.

Let $(\widehat{E_\emptyset}, \widehat{M_\emptyset})$ be the trivial source initial context composed of an empty environment (with no variables) and an empty execution memory (with no valid blocks). Let $(E_\emptyset, M_\emptyset, \overline{M_\emptyset})$ be the trivial target initial context composed of an empty environment, an empty execution memory and an empty observation memory $\overline{M_\emptyset}$. We trivially have

$$\begin{cases} (\widehat{E_\emptyset}, \widehat{M_\emptyset}) \sim (E_\emptyset, M_\emptyset), \\ M_\emptyset \triangleright \overline{M_\emptyset}. \end{cases}$$

So the following corollary follows from Theorem 5.7.

Corollary 5.10 (soundness of a runtime assertion checker). *Let s be a source program evaluated in the trivial source initial context to final memory state $\widehat{M_f}$, in other words, $\widehat{E_\emptyset}, \widehat{M_\emptyset} \vDash_s s \Downarrow \widehat{M_f}$. Then there exist memory states M_f and $\overline{M_f}$ such that the evaluation of $[\![s]\!]_\Sigma$ in the trivial target initial context leads to $(M_f, \overline{M_f})$, such that $(\widehat{E_\emptyset}, \widehat{M_f})$ is isomorphic to $(E_\emptyset, M_f)$ and $\overline{M_f}$ represents M_f. In other words,*

$$\begin{cases} E_\emptyset, M_\emptyset, \overline{M_\emptyset} \vDash_{s'} [\![s]\!]_\Sigma \Downarrow M_f, \overline{M_f}, \\ (\widehat{E_\emptyset}, \widehat{M_f}) \sim (E_\emptyset, M_f), \\ M_f \triangleright \overline{M_f}. \end{cases}$$

6 DISCUSSION AND RELATED WORK

Discussion. This paper formally presents and proves the soundness of a runtime assertion checker for memory-related properties. It is inspired by the program transformation implemented in the E-ACSL plug-in [46] of Frama-C [3] for tracking memory-related properties of its formal specification language. However, compared to its current implementation, we made a number of significant changes and simplifications.

First, we simplify both the programming and the specification languages. Such a reduction is quite standard when presenting a formalization work in a paper. In practice, the program transformation of the E-ACSL tool operates on basically the same source and target programming language, which is C, with the source language being additionally extended with ACSL annotations[4]. This language is arguably a much larger language than ours. However, the observation memory model introduced in this paper is representative of spatial memory errors occurring in C (without inline assembly code): we only need to know the memory effects done by each C statement or expression in order to decide how to instrument the code. Our programming language also includes a dedicated `let` construct to model global and local variables, and this choice simplifies our formalization. Support for local blocks and global and local variables instead of this construct is easy to implement in

[4]In a few cases outside the scope of this paper, E-ACSL may generate some new ACSL annotations in addition to the generated C code.

practice. Additionally, we also rely on a memory model broadly inspired by CompCert [30] to model the execution memory of our C-like language. Our memory models support badly typed and badly aligned accesses, even if our current source and target languages do not include them. An extension of the languages for such features is future work.

The specification language is also much simpler than the E-ACSL specification language [43]. However, the most important block-level memory properties are already considered in this paper and modeled thanks to our observation memory. In practice, the observation memory is based on a former work [50] and implemented as a C library. Extending the present formalization to make it fully consistent with the real-life library is future work. It is challenging because of its low-level bitwise implementation, even if recent efforts on verification of complex C libraries [8, 40] give us hope it would be possible. Supporting additional memory-related properties (e.g. the `\separated` predicate stating that a given set of pointers refer to pairwise disjoint memory locations) should not require modifying the observation memory implementation: it only requires adding the corresponding operation in the observation memory model and associated axioms.

The most important simplification made in the paper with respect to the specification language's semantics is related to undefined terms and predicates, such as `\valid(*p)` when `p` is not a valid pointer. Handling such undefined constructs is often referred to as the *undefinedness problem* [12]. Executing the resulting C code would result in an undefined behavior, which should never happen. Therefore, generating such a code would be a bug in the code generator. To prevent this issue, E-ACSL generates additional guards (typically, checking the validity of `p` before dereferencing it) while, in our formalization, we do not take care of them and just consider they have no meaning (i.e., there is no derivation tree in our operational semantics). Formalizing RAC in presence of undefinedness is future work.

The above-mentioned simplifications and changes with respect to C, the E-ACSL specification language and the E-ACSL plug-in of Frama-C allow us to focus on a particularly difficult part of the program transformation made by E-ACSL, namely memory properties, while keeping the study tractable in a research paper. It is part of a larger ongoing effort about formalizing E-ACSL, step by step [5, 6, 25]. Therefore, this paper focuses on the theoretical properties of the program transformation rather than its practical usages. In particular, the source and target languages have been designed to simplify the theoretical study and highlight the salient part of the transformation, and not for being used in practice. However, the interested readers may refer to previous papers for concrete applications [38] and experimental evaluations [49, 50] of E-ACSL regarding checking memory properties at runtime. They should also refer to the E-ACSL user manual [45] for using it in practice.

This paper also *assumes* we have an implementation of the observation memory respecting the axioms in Section 4.1. Such an implementation has been proposed by Vorobyov et al. [50] and is provided within E-ACSL. However, it has *not* been proved correct with respect to our set of axioms. That is left for future work. For being used in untrusted security-sensitive contexts, additional security properties, such as control-flow integrity [1], are often desirable in order to ensure (for instance) that the executed program does not bypass the memory check at runtime. Adding such security mechanisms is complementary to our work.

Related Work. More and more languages include a notion of contract. Design-by-contract is one of the main features of Eiffel [34], contracts have been introduced in Java through JML [27] in 1999, in C# through Spec# in 2011 [2], in Ada 2012 [18], and in OCaml through GOSPEL in 2019 [11]. 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. [51] study 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 proof using the Coq proof assistant [7]. Findler et al. [15, 16] study runtime checking in the context of object-oriented language such as Java, focusing in evaluating contracts' pre- and post-conditions in presence of method calls and inheritance. They prove the soundness of their approach. Cheon [12] formalizes RAC of JML, but provides no proof of soundness, while Lehner [28] 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 accesses correspond to invalid accesses to the memory, and are therefore handled in ACSL by the predicate `\valid`. Such memory properties have been formerly referred to as block-level memory properties [50]. 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. [51] propose and verify such a phase. It is also the case for our approach, which can be combined with a sound dataflow analysis [32]. Such optimizations are thus related to the verification of static analysis [23] and are not directly related to verification of runtime checkers.

Our contribution targets the C language [21], the Frama-C framework [3], the ACSL specification language [4] and the E-ACSL plug-in [46]. In particular we focus on memory properties. In Frama-C, the plug-in RTE [20] 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. There is an interest in formally verifying that the type system of Rust indeed provides strong guarantees [24], that the Rust language also provides *unsafe* pointers, and there exist Rust libraries to provide rudimentary support to express contracts, so 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. Several works have been done in that direction, including a formalization of a type system used as a pre-analysis before generating the code [25], a formalization of a GMP-based runtime assertion checker for mathematical properties [5, 6], or developing a sound GMP-equivalent library [40]. Grouping together all these formalization efforts is future work.

Regarding formal RAC of block-level memory properties, this work is an extension of a former work [31]. In particular, this version carefully reworks and details the axiomatization of the observation memory, presents a clearer design of the program transformation, and provides more proof details. The input language is also slightly larger. In particular, it now includes local variable declarations. Earlier, Petiot et al [39] also presented a formalization of a program transformer for E-ACSL-like properties. However, they simplify the transformation of memory properties by not

taking into account the observation memory. In particular, they do not explain how to instrument the input program for recording necessary pieces of information in the observation memory.

One 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 [19] for our extended source language. Finally, the E-ACSL plug-in currently does not support several interesting features of the specification language [44], such as axiomatized predicates. For them, a possible *verified* extension of E-ACSL could be based on the work of Tollitte et al. [47].

7 CONCLUSION

Runtime assertion checking of memory-related properties for a mainstream language like C relies on a complex program transformation that needs to record memory block metadata in a non-trivial, often highly optimized observation memory model. This work makes a significant step toward a formally proved runtime assertion checker for such memory properties.

We have presented a formalization of this program transformation for a representative programming language with pointers and dynamic memory allocation and proved the soundness of the resulting verification verdicts. We have proposed necessary machinery and proved that the generated monitor does not interfere with the original code. Our formalization and proofs are based on an observation memory model, which is particularly suitable for a modular definition and verification of the program transformation. The consistency of memory models and some of their key properties were proved in the Coq proof assistant [7].

Future work includes an extension of the present proof to real-life programming and specification languages, like C and ACSL respectively. It also includes a complete formalization and a mechanized proof of the runtime assertion checker in Coq.

Acknowledgment. The authors thank the reviewers for their helpful comments. The first author was partially funded by a grant of the French Ministry of Defense.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/1102120.1102165

[2] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (June 2011), 81–91. https://doi.org/10.1145/1953122.1953145

[3] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM* 64, 8 (2021), 56–68. https://doi.org/10.1145/3470569

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2022. *ACSL: ANSI/ISO C Specification Language*. http://frama-c.com/download/acsl.pdf.

[5] Thibaut Benjamin, Félix Ridoux, and Julien Signoles. 2022. Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution. In *Journées Francophones des Langages Applicatifs (JFLA'22)*. In French.

[6] Thibaud Benjamin and Julien Signoles. 2023. Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates. (2023). To appear in International Symposium on Applied Computing (2023).

[7] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[8] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10811)*. Springer, 37–53. https://doi.org/10.1007/978-3-319-77935-5_3

[9] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[10] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer

Society, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[11] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. 2019. GOSPEL - Providing OCaml with a Formal Specification Language. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11800)*. Springer, 484–501. https://doi.org/10.1007/978-3-030-30942-8_29

[12] Yoonsik Cheon. 2003. *A runtime assertion checker for the Java Modeling Language.* Ph. D. Dissertation. Iowa State University.

[13] Lori A. Clarke and David S. Rosenblum. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 25–37. https://doi.org/10.1145/1127878.1127900

[14] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. 2013. Common specification language for static and dynamic analysis of C programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. 1230–1235. https://doi.org/10.1145/2480362.2480593

[15] Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for Object-Oriented Languages. https://doi.org/10.1145/504282.504283

[16] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. 2001. Behavioral Contracts and Behavioral Subtyping. https://doi.org/10.1145/503209.503240

[17] Cormac Flanagan and James B. Saxe. 2001. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001)*. ACM, 193–205. https://doi.org/10.1145/360204.360220

[18] ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group. 2012. *ADA Reference Manual.* http://www.ada-auth.org/standards/ada12.html

[19] Paolo Herms. 2013. *Certification of a Tool Chain for Deductive Program Verification. (Certification d'une chaine de vérification déductive de programmes).* Ph. D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00789543

[20] Philippe Herrmann and Julien Signoles. 2022. *Annotation generation: Frama-C's RTE plug-in.* http://frama-c.com/download/frama-c-rte-manual.pdf.

[21] ISO/IEC 9899:1999 1999. *Programming languages – C.* ISO/IEC 9899:1999.

[22] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. 2016. Fast as a shadow, expressive as a tree: Optimized memory monitoring for C. *Sci. Comput. Program.* 132 (2016), 226–246. https://doi.org/10.1016/j.scico.2016.09.003

[23] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. *SIGPLAN Not.* 50, 1 (2015), 247–259. https://doi.org/10.1145/2775051.2676966

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2017). https://doi.org/10.1145/3158154

[25] Nikolai Kosmatov, Fonenantsoa Maurica, and Julien Signoles. 2020. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12399)*. Springer, 310–322. https://doi.org/10.1007/978-3-030-60508-7_17

[26] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. 2013. An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. 167–182. https://doi.org/10.1007/978-3-642-40787-1_10

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38. https://doi.org/10.1145/1127878.1127884

[28] Hermann Lehner. 2011. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.* Ph. D. Dissertation. ETH Zurich.

[29] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[30] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[31] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. 2020. Verified Runtime Assertion Checking for Memory Properties. In *Tests and Proofs - 14th International Conference, TAP@STAF 2020, Bergen, Norway, June 22-23, 2020, Proceedings [postponed] (Lecture Notes in Computer Science, Vol. 12165)*. Springer, 100–121. https://doi.org/10.1007/978-3-030-50995-8_6

[32] Dara Ly, Nikolai Kosmatov, Julien Signoles, and Frédéric Loulergue. 2019. Soundness of a Dataflow Analysis for Memory Monitoring. *Ada Lett.* 38, 2 (dec 2019), 97–108. https://doi.org/10.1145/3375408.3375416

[33] Fonenantsoa Maurica, David R. Cok, and Julien Signoles. 2018. Runtime Assertion Checking and Static Verification: Collaborative Partners. In *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Verification (ISoLA 2018) (LNCS, Vol. 11245)*. Springer, 75–91. https://doi.org/10.1007/978-3-030-03421-4_6

[34] Bertrand Meyer. 1991. *Eiffel: The Language*. Prentice-Hall. http://www.eiffel.com/doc/#etl

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[36] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*. 65–74. https://doi.org/10.1145/1254810.1254820

[37] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[38] Dillon Pariente and Julien Signoles. 2017. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*.

[39] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 105–114. https://doi.org/10.1109/SCAM.2014.19

[40] Raphaël Rieu-Helft. 2019. A Why3 proof of GMP algorithms. *J. Formaliz. Reason.* 12, 1 (2019), 53–97. https://doi.org/10.6092/issn.1972-5787/9730

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. USENIX Association, 309–318.

[42] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 17–30. http://www.usenix.org/events/usenix05/tech/general/seward.html

[43] Julien Signoles. 2022. *E-ACSL: Executable ANSI/ISO C Specification Language*. http://frama-c.com/download/e-acsl/e-acsl.pdf.

[44] Julien Signoles. 2022. *E-ACSL Version 1.18. Implementation in Frama-C Plug-in E-ACSL 26.0*. http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[45] Julien Signoles, Basile Desloges, and Kostyantyn Vorobyov. 2022. *E-ACSL User Manual*. https://www.frama-c.com/download/e-acsl/e-acsl-manual.pdf.

[46] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*. 164–173. http://www.easychair.org/publications/paper/t6tV

[47] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. 2012. Producing Certified Functional Code from Inductive Specifications. In *Certified Programs and Proofs (CPP) (LNCS)*. Springer, Berlin, Heidelberg, 76–91. https://doi.org/10.1007/978-3-642-35308-6_9

[48] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7462)*. Springer, 86–106. https://doi.org/10.1007/978-3-642-33338-5_5

[49] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. 2018. Detection of Security Vulnerabilities in C Code Using Runtime Verification: An Experience Report. In *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*. 139–156. https://doi.org/10.1007/978-3-319-92994-1_8

[50] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. 2017. Shadow state encoding for efficient monitoring of block-level properties. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, Barcelona, Spain, June 18, 2017*. 47–58. https://doi.org/10.1145/3092255.3092269

[51] Zhi Zhang, Robby, John Hatcliff, Yannick Moy, and Pierre Courtieu. 2017. Focused Certification of an Industrial Compilation and Static Verification Toolchain. In *Software Engineering and Formal Methods (SEFM) (LNCS, Vol. 10469)*. Springer, 17–34. https://doi.org/10.1007/978-3-319-66197-1_2