Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking

Thibaut Benajmin[1][0000−0002−9481−1896] and Julien Signoles[1][0000−0001−9266−0820]

Université Paris-Saclay, CEA, List, Palaiseau, France
thibaut.benjamin@gmail.com julien.signoles@cea.fr

Abstract. Runtime Assertion Checking (RAC) is a lightweight formal method for verifying at runtime code properties written in a formal specification language. One of the main challenge of RAC is to check the properties efficiently, while emitting sound verdicts. In particular, arithmetic properties are only efficiently verified using machine integers, yet soundness can only be achieved by using an exact but slower exact arithmetic library. This paper presents how E-ACSL, a RAC tool for C programs, applies abstract interpretation for efficiently and soundly supporting arithmetic properties. Abstract interpretation provides sound static information regarding the size of terms involved in runtime assertions in order to choose at compile time whether machine integers or exact arithmetic will be used at runtime on a case by case basis. Our specification language includes recursive user-defined logic functions and predicates, for which we rely on fast fixpoint operators based on widening of abstract values.

1 Introduction

Runtime Assertion Checking (RAC) is a lightweigth formal method that consists in checking at runtime formal properties written as code annotations [7]. For this purpose, a RAC tool usually takes a source code (or bytecode) program p as input and generates as output an inline monitor that observes each p's execution. An inline monitor means that the (source, byte or binary) code of the monitor is part of the observed program: the generated chunks of code interleave with pieces of code of the original program [10]. This paper focuses on E-ACSL [20], the RAC tool of Frama-C [1], an analysis framework for code written in C. The formal properties are written in a variant of the ACSL specification language [2], also named E-ACSL and dedicated to runtime checking [9]. The E-ACSL tool takes as input a C program annotated with E-ACSL specifications and generates a new C program in which the formal annotations have been converted to C code.

E-ACSL aims at satisfying four key properties, that are quite usual for RAC tools: *expressivity*, *transparency*, *soundness*, and *efficiency*. *Expressivity* means that the more properties a RAC tool can check the better. *Transparency* means that the inline monitor must not modify the functional behavior of the original program: when the checked properties are all satisfied, the monitored program must produce the same output as the unmonitored program. Transparency is out-of-scope in this paper. *Soundness* means that the inline monitor must emit

correct verdicts when checking properties. *Efficiency* means that the inline monitor must run as efficiently as possible: the time and memory overheads of the monitored program with respect to the unmonitored program must remain as low as possible. It also means that generating the code of the monitor must be efficient enough. In our context, being "efficient enough" for generating the code means being as efficient as standard optimizing compilers.

Regarding expressivity, we focus here on integer properties only. E-ACSL is based on integer arithmetics in $\mathbb{Z}$, the set of mathematical integers, which allows users to specify arithmetic properties without implementation details in mind: assuming `x` is a C variable of type `int` and `n` is an integer constant, `x + n` can never overflow in a formal property, although it might in C code. The formal properties can also call user-defined possibly-recursive logic functions and predicates. Such definitions allow users to specify once complex parameterized computations and properties and use them several times.

Such an expressivity leads to an issue regarding efficiency and soundness. Indeed, using mathematical integers requires to rely on a dedicated exact arithmetic library, such as GMP[1] in C, for generating correct code, while using machine bounded integers would be much better for efficiency. For taking the best of both worlds and being both correct and efficient, E-ACSL relies on a dedicated static analysis that allows it to use efficient machine bounded integers when it is safe to do so and inefficient-yet-correct exact arithmetics otherwise. The static analysis is itself efficient and thus allows E-ACSL to generate the efficient code efficiently: even if based on abstract interpretation [8], it only uses the simple interval domain and a fast widening operator that scale extremely well, yet is precise enough for our need. Therefore, it reaches the goal of being as efficient as standard optimizing compilers, contrary to most existing abstract interpreters that target proving properties such as absence of bugs. **This paper presents this static analysis in the presence of recursive logic definitions, proves its correctness and shows experimentally how it helps E-ACSL to generate efficient C code**. Although focusing on E-ACSL, our contributions can be applied on other contexts: they can be applied to any runtime system that must deal with mathematical numbers, including other runtime assertion checkers, different kinds of runtime verification tools, or simulators.

This work is the last item of a series of works about formalizing E-ACSL. In particular, it extends the work of Kosmatov et al [13] to logic definitions. To do so, it requires to move from a type-system based setting to an abstract-interpretation based setting to deal with recursive definitions soundly and precisely. This way, we improve a recently published paper [3] that formalizes the E-ACSL's code generator for the very same language fragment, assuming a sound static analysis. Indeed, the static analysis is presented in detail here, while we also prove the assumptions of [3] about soundness of the analysis and push forward the experimental evaluation about efficiency. As far as we know, no other prior work targets recursive logic definitions for runtime assertion checking.

[1] http://gmplib.org

Ly et al formalized another subset of E-ACSL, targeting memory properties [17,16]. Their works are complementary to ours. Beyond E-ACSL, Cheon [6] was the first to formally study RAC, in the context of JML [14], a formal specification language for Java. He did not focus his work on integer arithmetic since, at that time, the JML's arithmetic was exactly the machine arithmetic. Later, Lehner [15] formalized in Coq a large subset of the JML's semantics. He also formalized a RAC algorithm for the JML's `assignable` clause, which is independent from, but compatible with, our integer properties. More recently, Filliâtre and Pascutto [11] proposed Ortac, a RAC tool for OCaml. It relies on a similar mechanism to ours for generating efficient arithmetic code, but without details nor formalization for that part. They also do not deal with logic definitions. Recently, they formally studied how to optimize referring to the pre-state from the post-state of a function [12]. This work is complementary to ours.

Section 2 presents an overview of our work on a concrete example. Section 3 introduces the programming and specification languages supporting our formalization. Section 4 details our static analysis without considering logic definitions. Section 5 extends it to the whole considered languages and presents our formal results. Section 6 presents our experimental evaluation.

2 Illustrated Overview

Fig 1 shows an example of an annotated program together with a simplified version of the instrumented code generated by E-ACSL. For the sake of simplicity, we assume that the program is executed on an 8-bit architecture, where the type `int` ranges from -128 to 127, and that there is no machine integer type greater than this. In this example, three assertions are translated. For the first one, the translation is straightforward, as it suffices to replace the assertion with the exact same assertion in C. The second one is more complex: since the addition it involves overflows in the machine integers, we rely on the GMP library, which provides exact integer arithmetic. The last assertion is the most complex since it calls a user-defined recursive function. Its translation generates a C function that specializes this ACSL function, while keeping track of the size of the numbers involved to use either machine integers or GMP. This article presents a static analysis based on abstract interpretation whose purpose is to provide the information required to decide whether a particular term can soundly be translated using machine integer or must rely on inefficient GMP integers.

3 Language Definition

The formal presentation of the paper focuses on a core arithmetic subset of the C language, called mini-C. mini-C programs may contain formal annotations written in a subset of the ACSL specification language [2], called mini-ACSL. Its main feature is the support of user-defined logic functions and predicates, including mutually recursive ones.

3.1 Formal Syntax

Figure 2 presents the syntax of the languages mini-C and mini-ACSL together, as they mutually depend on each other. An annotated mini-C program is a sequence

```
/*@ logic integer f (integer x) = x <= 0 ? 0 : f(x - 1) + 1; */
void main () {
  //@ assert 10 + 20 == 30;
  //@ assert 120 + 30 == 150;
  //@ assert f (50) == 50;
}
```

(a) Annotated Program

```
void _f(mpz_t *_res, int x);
void main(void) {
  assert(10 + 20 == 30);
  { mpz_t _a, _b, _c, _add; int eq;
    mpz_init_set_si(_a, 120); mpz_init_set_si(_b, 30);
    mpz_init_set_str(_c,"30"); mpz_init(_add);
    mpz_add(_add, _a, _b);
    eq = mpz_cmp(_add, _c);
    assert (eq == 0);
    mpz_clear(_a); mpz_clear(_b); mpz_clear(_c); mpz_clear(_add); }
  { _mpz_t _f_1, _x_1; int eq_2;
    _f(& _f_1,50);
    mpz_init_set_si(_x_3,50L);
    eq_2 = mpz_cmp(_f_1,x_1);
    assert (eq_2 == 0);
    mpz_clear(_f_1); mpz_clear(_x_1); }
}
void _f(_mpz_t *_res, int x) {
  if (x <= 0) { mpz_init_set_si(*res, 0); }
  else { mpz_t _f_2, _x;
    _f(& _f_2,x - 1);
    mpz_init_set_si(_x,1L); mpz_init(*res);
    mpz_add(*res,_f_2, _x);
    mpz_clear(_f_2); mpz_clear(_x); }
}
```

(b) Instrumented Program

Fig. 1: Example of an Annotated Program and its Instrumented Version.

of program variables declarations, followed by a sequence of function definitions, which may be either a mini-C function, or a user-defined logic function or predicate epxressed in the mini-ACSL language. For simplicity, we assume that the only type of mini-C is `int`, i.e. bounded machine integers: our results can easily be extended to a language with more bounded integer types. The program functions are made of statements that include standard control flow structures, such as loops and conditionals, as well as arithmetic operations. The statements also include logical assertions, expressed in the mini-ACSL specification language. Assertions are propositional predicates over mathematical (unbounded) integer terms. Terms and predicates may include calls to user-defined logic functions and predicates, which can be (mutually) recursive. Syntactically, no restriction is put on the recursion scheme of functions and predicates.

3.2 Program Structure

We assume that all the programs given as inputs are syntactically well-formed and properly typed, even if the type system is omitted here. We denote $\mathcal{V}$ the set of program variables and $\mathcal{S}$ the set of statements, as well as $\mathfrak{L}$ the set of logic binders (i.e., the logic variables introduced as parameters of user-defined logic functions and predicates), $\mathfrak{Z}$ the set of logical terms and $\mathfrak{B}$ the set of predicates of the program. For the sake of simplicity, we consider any program function identifier as being a particular program variable, and any logic function and predicate identifier as being a particular logic binder. The partial function $\mathcal{F} : \mathcal{V} \rightharpoonup \mathcal{V}^* \times \mathcal{S}$, associates to each variable denoting a program function, the list

$p ::= d^*\ f^*$	annotated program		$\delta ::= \tau$ id		logic declaration
$d ::=$ `int` id;	program declaration		$p ::=$ `\true` | `\false`		truth values
$f ::=$ `int` id$(d^*)\{d^*; s_c\}$	program function		$\mid t \triangleleft t$	$\triangleleft \in \{<; \leq; >; \geq; \overset{?}{=}; \neq\}$	
$\mid$ `/*@ logic` τ id$(\delta^*) = t$ `*/`	logic function		$\mid$ `!` t		negation
$\mid$ `/*@ predicate` id$(\delta^*) = p$ `*/`	predicate		$\mid p$ `||` p		disjunction
$s_c ::=$ `skip;`	empty statement		$\mid$ id(δ^*)		predicate call
$\mid$ id `=` e;	assignment		$t ::= z$		integer in $\mathbb{Z}$
$\mid$ id `=` id(e^*);	function call		$\mid$ id		variable access
$\mid s\ s$	sequence		$\mid t \diamond t$	$\diamond \in \{+; -; \times; /\}$	
$\mid$ `if`$(e)\ s$ `else` s	conditional		$\mid p$ `?` t `:` t		conditional term
$\mid$ `while`$(e)\ s$	loop		$\mid$ id(δ^*)		function call
$\mid$ `assert`(e);	program assertion		$\kappa ::=$ `int` | `integer`		logic types
$\mid$ `/*@ assert` p `*/`	logic assertion				
$\mid$ `return`(e);	return statement				
$e ::= z_m$	machine integer				
$\mid$ id	variable access				
$\mid e\ \diamond_{\mathtt{C}}\ e$	$\diamond_{\mathtt{C}} \in$ {+; -; *; /}				
$\mid e\ \triangleleft_{\mathtt{C}}\ e$	$\triangleleft_{\mathtt{C}} \in$ {<; <=; >; >=; ==; !=}				

Fig. 2: Syntax of mini-C (left) and mini-ACSL (right).

of variables corresponding to its parameters together with the statement defining its body. Similarly, we assume two partial functions $\mathfrak{F} : \mathfrak{L} \rightharpoonup \mathfrak{L}^* \times \mathfrak{Z}$ and $\mathfrak{P} : \mathfrak{L} \rightharpoonup \mathfrak{L}^* \times \mathfrak{B}$ modeling respectively the set of user-defined logic functions and the set of user-defined predicates. In practice, the assumptions made are guaranteed by Frama-C [1], which also computes the functions $\mathcal{F}, \mathfrak{F}$ and $\mathfrak{P}$. For any partial function f, $f\{x \backslash v\}$ is defined as $f\{x \backslash v\}(x) = v$ and $f\{x \backslash v\}(y) = f(y)$ for any $y \neq x$. It is also worth noting the following key remark about mini-ACSL.

Remark 1 (Accessibility of logic bindings). The only logic variables in $\mathfrak{L}$ bounded in a function or predicate body are its formal parameters, although global program variables in $\mathcal{V}$ may also be bounded.

3.3 Concrete Semantics

This section defines the concrete semantics of mini-C and mini-ACSL. Let $\mathrm{m}_{\mathtt{int}}$ and $\mathrm{M}_{\mathtt{int}}$ be respectively the smallest and biggest integer representable in the type `int` and $\mathbb{V} = \mathsf{Int} \cup \mathbb{U}$ be the set of values that a mini-C expression may evaluate to, where Int is the set of possible values of a variable of type `int` and $\mathbb{U}$ is an infinite set of arbitrary undefined values representing the unitialized values. We have the following bijection for the representation of `int` values: $\dot{\_} : \mathsf{Int} \simeq [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}] : \_^{\mathsf{int}}$. The use of the set $\mathbb{U}$ and the explicit bijection between Int and $[\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]$ are not necessary for the purpose of our analysis. This details could be omited here, but we keep them to be consistent with the semantics of [3] since we prove here assumptions of this paper. We denote by $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$ the set of truth values and by $\mathbb{Z}$ be the set of mathematical integers.

The semantics of our languages is evaluated in a *concrete environment* Ω, which is a pair of two partial functions $\Omega_{\mathcal{V}} : \mathcal{V} \to \mathsf{Int}$ and $\Omega_{\mathfrak{L}} : \mathfrak{L} \to \mathbb{Z}$. For the sake of simplicity, we sometimes treat Ω as a single partial function, as determining which of the component is referred to is usually non ambiguous from the context. The semantics of a mini-C statement s is expressed by the judgment $\Omega \vDash s \Rightarrow \Omega'$, stating that evaluating s in the environment Ω yields the environment Ω'. Similarly, the semantics of a mini-C expression e is expressed by the judgment $\Omega \vDash e \Rightarrow v$, with $v \in \mathbb{V}$, the semantics of a mini-ACSL predicate

p is expressed by the judgment $\Omega \vDash p \Rightarrow b$ with $b \in \mathbb{B}$ and the semantics of a mini-ACSL term is expressed by the judgment $\Omega \vDash t \Rightarrow z$ with $z \in \mathbb{Z}$. Fig. 3 presents the derivation rules for the semantics of mini-C. The result of a call to function f is transmitted from the callee to the caller through a distinguished variable res_f.The rest of this semantics is fairly standard and straightforward. Fig. 4 presents the semantics of the mini-ACSL specification language. Again this semantics is quite standard, except that terms evaluate in the set of mathematical integers $\mathbb{Z}$ and not in the set of machine integers.

Semantics of declarations

$$\frac{x \notin \mathrm{dom}\,(\Omega_{\mathcal{V}}) \qquad u \in \mathbb{U}}{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash \texttt{int}\ x \Rightarrow \Omega_{\mathcal{V}}\{x \backslash u\}, \Omega_{\mathfrak{L}}}$$

Semantics of statements

$$\frac{}{\Omega \vDash \texttt{skip;} \Rightarrow \Omega} \qquad \frac{\Omega_{\mathcal{V}}(x) \in \mathbb{V} \qquad \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash e \Rightarrow z}{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash x\ =\ e \Rightarrow \Omega_{\mathcal{V}}\{x \backslash z\}, \Omega_{\mathfrak{L}}} \qquad \frac{\Omega \vDash s \Rightarrow \Omega' \qquad \Omega' \vDash s' \Rightarrow \Omega''}{\Omega \vDash s\ s' \Rightarrow \Omega''}$$

$$\frac{\Omega \vDash e \Rightarrow z \qquad z \neq 0^{\mathsf{int}} \qquad \Omega \vDash s \Rightarrow \Omega'}{\Omega \vDash \texttt{if}(e)\ \texttt{then}\ s\ \texttt{else}\ s' \Rightarrow \Omega'} \qquad \frac{\Omega \vDash e \Rightarrow 0^{\mathsf{int}} \qquad \Omega \vDash s' \Rightarrow \Omega'}{\Omega \vDash \texttt{if}(e)\ \texttt{then}\ s\ \texttt{else}\ s' \Rightarrow \Omega'}$$

$$\frac{\Omega \vDash \texttt{if}(e)\ \texttt{then}\ s;\ \texttt{while}(e)\ s\ \texttt{else skip} \Rightarrow \Omega'}{\Omega \vDash \texttt{while}(e)\ s \Rightarrow \Omega'}$$

$$\frac{\Omega \vDash e \Rightarrow z \qquad z \neq 0}{\Omega \vDash \texttt{assert(e)} \Rightarrow \Omega} \qquad \frac{\Omega \vDash p \Rightarrow \mathrm{T}}{\Omega \vDash \texttt{/*@ assert}\ p\ \texttt{*/} \Rightarrow \Omega} \qquad \frac{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash e \Rightarrow z}{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash \texttt{return}(e) \Rightarrow \Omega_{\mathcal{V}}\{\mathrm{res}_f \backslash z\}, \Omega_{\mathfrak{L}}}$$

$$\frac{\begin{array}{c}\mathcal{F}(f) = (x_1, \ldots, x_n; b) \\ \Omega \vDash e_1 \Rightarrow z_1; \ldots; \Omega \vDash e_n \Rightarrow z_n \qquad \{x_1 \backslash z_1, \ldots, x_n \backslash z_n\}, \Omega_{\mathfrak{L}} \vDash b \Rightarrow \Omega'_{\mathcal{V}}, \Omega'_{\mathfrak{L}} \qquad \Omega'_{\mathcal{V}}(\mathrm{res}_f) = z\end{array}}{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash c\ =\ f(e_1, \ldots, e_n) \Rightarrow \Omega_{\mathcal{V}}\{c \backslash z\}, \Omega_{\mathfrak{L}}}$$

Semantics of expressions

$$\frac{}{\Omega \vDash z_m \Rightarrow z_m} \qquad \frac{\Omega_{\mathcal{V}}(x) = z}{\Omega \vDash x \Rightarrow z} \qquad \frac{\Omega \vDash e \Rightarrow z \quad \Omega \vDash e' \Rightarrow z' \quad \dot{z} \lhd \dot{z'}}{\Omega \vDash e\ \lhd_{\mathbf{C}}\ e' \Rightarrow 1^{\mathsf{int}}} \qquad \frac{\Omega \vDash e \Rightarrow z \quad \Omega \vDash e' \Rightarrow z' \quad \dot{z} \not\lhd \dot{z'}}{\Omega \vDash e\ \lhd_{\mathbf{C}}\ e' \Rightarrow 0^{\mathsf{int}}}$$

$$\frac{\Omega \vDash e \Rightarrow z \qquad \Omega \vDash e' \Rightarrow z' \qquad \mathrm{m}_{\mathrm{int}} \leq \dot{z} \diamond \dot{z'} \leq \mathrm{M}_{\mathrm{int}} \qquad \mathrm{not}(\diamond_{\mathbf{C}} = / \text{ and } \dot{z'} = 0)}{\Omega \vDash e\ \diamond_{\mathbf{C}}\ e' \Rightarrow (\dot{z}\ \diamond\ \dot{z'})^{\mathsf{int}}}\ \begin{array}{l}(\lhd \text{ models } \lhd_{\mathbf{C}};\\ \diamond \text{ models } \diamond_{\mathbf{C}})\end{array}$$

Fig. 3: Semantics of the mini-C language.

The semantics presented here is blocking, that is only correct programs with correct annotations can be ascribed a semantics using these rules. In particular, terms and predicates calling logic definitions with ill-formed recursion schemes have no semantics, since as soon as a call is non-terminating, there is no finite derivation tree to ascribe a semantics to it. Constructs that would lead to runtime errors, which are restricted to division by zero in our arithmetic context, have also no semantics. In practice, E-ACSL embeds a mechanism that checks at runtime potential runtime errors such as divisions by zero in terms and predicates before executing them [9]. It allows E-ACSL to not add executable undefined behaviors in the generated code. This mechanism is not presented here.

3.4 Collecting Semantics

Our static analysis is based on abstract interpretation. Proving its correctness in Section 5 requires to show that its result includes the results from all concrete executions. A common way to proceed is to first define the *collecting semantics* that computes all these results at once. Since our analysis focuses on terms, we only define it for such constructs, not for the others. Let us denote $\Xi \in \mathcal{P}(\mathfrak{L} \rightharpoonup \mathbb{Z})$ a collecting environment, i.e. a set of partial functions from binders to integers

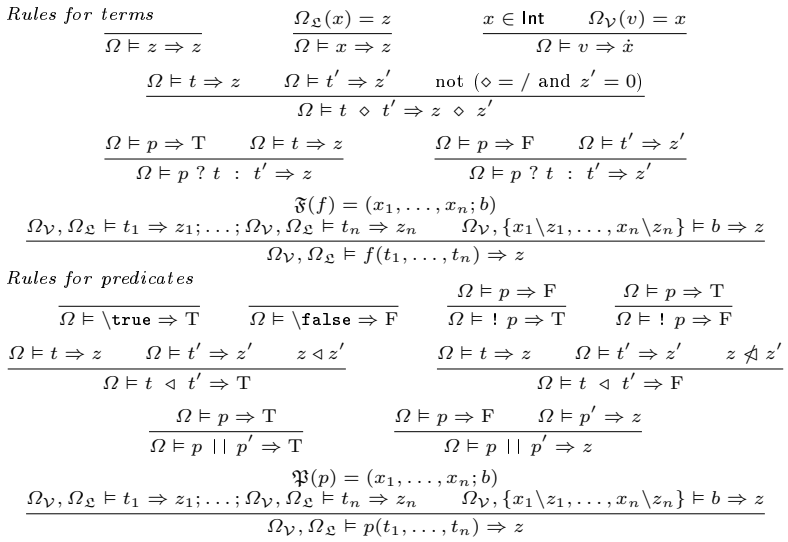

Fig. 4: Semantics of the mini-ACSL language.

(otherwise said, a set of logic environments). The collecting semantics $\mathcal{C}(\Xi, t)$ of a term t in an environment Ξ is then defined as follows:

$$\mathcal{C}(\Xi, t) \equiv \{z \mid \exists \Omega_{\mathfrak{L}} \in \Xi, \exists \Omega_{\mathcal{V}} : \mathcal{V} \rightharpoonup [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}], \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t \Rightarrow z\}.$$

4 Abstract Interpretation without Logic Functions

This section presents our static analysis based on abstract interpretation, assuming there is no logic definition: they will be added in Section 5. We only analyze mini-ACSL annotations: no static analysis is performed on the mini-C code. Our aim is to provide an interval associated to each term, so that a monitor generator can decide whether the term can be safely monitored with machine integers. If the interval contains integers that do not fit into machine integers, the monitor will perform the computation in arbitrary precision arithmetic for soundness. The monitor generator that uses the interval computed here is presented in [3].

4.1 Lattice of Intervals

Our analysis is only based on the integer interval domain, presented here. Indeed, while more evolved domains might provide more precise answers, it would be less efficient and could prevent E-ACSL to be as fast as optimizing compilers. The precision of the interval domain is enough in practice. Would a more precise domain be necessary in the future, our analysis could easily be adapted.

$\mathcal{I}$ denotes the set of (possibly empty) integer intervals with possibly infinite bounds. We denote by $\bot$ the empty interval and $\top$ the interval with infinite lower and upper bounds, which is $\mathbb{Z}$ itself. $\mathcal{I}$ and the set inclusion $\subseteq$ as partial order is a lattice. The join operator $\vee$ (resp. meet operator $\wedge$) is the set union $\cup$ (resp. set intersection $\cap$). We introduce the pair of maps $\mathcal{P}(\mathbb{Z}) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \mathcal{I}$. with the map α being defined by $\alpha(X) = [\min X, \max X]$, assuming that $\max X = +\infty$

(resp. $\min X = -\infty$) if X has no upper (resp. lower) bound. For the empty set, we define $\alpha(\emptyset) = \bot$. α is named the abstraction map. γ, defined by $\gamma(I) = I$, is named the concretization map. This pair of maps is a Galois connection, i.e. for each $X \in \mathcal{P}(\mathbb{Z})$ and $I \in \mathcal{I}$, $X \subseteq \gamma(I)$ if and only if $\alpha(X) \subseteq I$. It allows us to convert data from the concrete world to the abstract world through α and conversely through γ, possibly by introducing approximations. Given an operator $\star$ on $\mathcal{P}(\mathbb{Z})$, we denote $\star^\sharp$ the corresponding operator on intervals, defined by $I \star^\sharp I' = \alpha(\gamma(I) \star \gamma(I'))$. This abstract operator allows us to lift operators $\diamond$ on concrete values to operators $\diamond^\sharp = \diamond^\sharp_{\text{set}}$ in the abstract world, where $\diamond_{\text{set}}$ is defined by $X \diamond_{\text{set}} Y = \{x \diamond y \mid x \in X, y \in Y\}$. We will also use *abstract environments* $\Gamma : \mathfrak{L} \rightharpoonup I$ that abstract concrete environments by mapping logic variables to intervals. In order to ensure that the static analysis always terminates quickly, even in the presence of non-terminating functions, we will use a *widening* operator ∇, introduced in Section 6. For the time being, it is enough to know that it satisfies the two following properties, quite usual in abstract interpretation [8]:

(W1) For every pair of intervals I and I', we have $I \subseteq I \nabla I'$ and $I' \subseteq I \nabla I'$
(W2) For every increasing sequence (J_i), the sequence defined by $I_0 = J_0$ and $I_{n+1} = I_n \nabla J_{n+1}$ stabilizes.

4.2 Inference Rules

This section presents the inference rules for the derivation of interval judgments. We introduce an *environment of logic functions* $\Delta : \mathfrak{F} \to (\mathfrak{L} \rightharpoonup \mathcal{I}) \times \mathcal{I}$. For each function f already encountered, it keeps track of the intervals inferred for each of f's parameters and the interval of the f's return value. This environment is useless right now in the absence of logic definitions: it will only be used in Section 5, but introducing it right now allows for rules of this section to remain unchanged.

In the absence of logic definition, our static analysis is a simple interval inference introduced by the judgment $\Gamma | \Delta \vdash t : I$ defined in Fig. 5. It means that the values of the mini-ACSL term t belong to the interval I.

$$\frac{}{\Gamma|\Delta \vdash z : [z, z]} \qquad \frac{}{\Gamma|\Delta \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma|\Delta \vdash v : [\mathrm{m_{int}}, \mathrm{M_{int}}]}$$

$$\frac{\Gamma|\Delta \vdash t : I \qquad \Gamma|\Delta \vdash t' : I'}{\Gamma|\Delta \vdash t \ \diamond \ t' : I \diamond^\sharp I'} \qquad \frac{\Gamma|\Delta \vdash t : I \qquad \Gamma|\Delta \vdash t' : I'}{\Gamma|\Delta \vdash p \ ? \ t \ : \ t' : I \cup^\sharp I'}$$

Fig. 5: Interval inference for the function-free core of the mini-ACSL language.

The rules are quite straightforward. The first rule associates to a constant the corresponding singleton interval. The second rule associates to a logic binder x, its interval stored in the environment Γ. The third rule associates to a C variable v the interval of integers representable in the type `int`. The fourth rule associates to an operation the result of its corresponding abstract operation. The last rule joins the results of both branches of a conditional. These rules are similar to the ones of [13], even if expressed here in another formalism.

4.3 Improving Precision for Conditionals

The rule for conditionals can be improved by taking into account that the condition is necessarily true in the positive branch and false in the negative one.

When these properties can be encoded in the interval domain (e.g., when comparing a variable to a constant), it is possible to refine this rule to improve the precision of the analysis. Such an optimization is implemented in practice, even if the details are omitted here. For instance, when the condition is `x >= 0`, the rule can be refined to the following one:

$$\frac{\Gamma\{x\backslash\Gamma(x) \wedge [0,+\infty]\}|\Delta \vdash t : I \qquad \Gamma\{x\backslash\Gamma(x) \wedge [-\infty,-1]\}|\Delta \vdash t' : I'}{\Gamma|\Delta \vdash \texttt{x >= 0 ?}\ t : t' : I \cup^\sharp I'}.$$

5 Abstract Interpretation with Logic Functions

We now extend our static analysis to handle recursive functions. We do not formalize the support for recursive predicates: it is very similar to recursive functions and even simpler since their body are Boolean values, which leads to a trivial finite lattice. Yet, they are handled in our evaluation, in Section 6. Throughout this section, we consider a function f such that $\mathfrak{F}(f) = (x_1, \ldots, x_n; b)$, meaning that its parameters are $x_1, \ldots x_n$ and its body is b.

5.1 Inference Rules

When encountering a function call, we need to extend the abstract environment in order to associate the interval of each argument to the corresponding function's parameter. We also need to update the abstract environment when encountering recursive calls up to reaching a fixpoint. Given an environment for logic functions Δ and a function f, we denote by $\Delta_{\text{args}}(f)$ and $\Delta_{\text{res}}(f)$ respectively the first and second component of $\Delta(f)$ in such a way that $\Delta(f) = (\Delta_{\text{args}}(f), \Delta_{\text{res}}(f))$. Given a list of intervals $I_1, \ldots, I_n$, we define $\Delta\langle f \nabla I_1, \ldots, I_n\rangle$ as follows:

$$\Delta\langle f \nabla I_1, \ldots, I_n\rangle \equiv \Delta\{f\backslash(\Gamma\{x_1\backslash\Gamma(x_1)\nabla I_1, \ldots, x_n\backslash\Gamma(x_n)\nabla I_n\}, \Delta(f)_{\text{res}})\}$$
$$\text{where } \Gamma = \Delta_{\text{args}}(f).$$

This definition directly uses $\Delta_{\text{args}}(f)$ in place of the abstract environment Γ, without taking care of any potential existing binding. Said otherwise, this definition does not depend on any abstract environment Γ'. This is possible since the only bounded logic variables in a function body are its formal parameters according to Remark 1, while the interval of any program variable is constant (directly derived from their types, which is necessarily `int`, as made explicit in Fig. 5), so we do not need to store them in the abstract environment.

Fig. 6 presents the inference rules for the interval inference for logic functions. It depends on a second judgment, denoted $\Delta \vdash_f f : I$, which means that the result of the function f fits into the interval I in Δ. By convention, we consider that f not being in the domain of Δ, is equivalent to having $\Delta(f) = (\Gamma, \bot)$ with Γ the constant function equal to $\bot$. As such, this rule system is not deterministic since the premises of the rules (Fun) and (Init) overlap, and so do those of the rules (Base) and (Ind). For determining the inference algorithm, we always apply (Fun) over (Init) and (Base) over (Ind). The rules (Base) and (Ind) only depend on an environment of logic functions Δ and does not depend on any abstract environment Γ for the above-mentioned reason. Altogether, these rules compute two fixpoints: one over the inputs and one over the result of a function

call. The rule (Fun) states that, when the fixpoint for the inputs is reached, the result of a function call is the interval computed for its body and stored in the environment Δ. The rule (Init) initiates a fixpoint computation for the output of the function call, assuming widened intervals associated to each formal parameter before computing the function body. Such a computation also relies on a fixpoint: the rule (Base) returns the interval computed for the body when the fixpoint is reached, while the rule (Ind) is the recursive case that widens the previously computed interval for the body before computing it again.

$$\frac{\Gamma|\Delta \vdash t_1 : I_1 \quad \dots \quad \Gamma|\Delta \vdash t_n : I_n \qquad \forall i, I_i \subseteq \Delta_{\text{args}}(f)(x_i)}{\Gamma|\Delta \vdash f(t_1, \dots, t_n) : \Delta_{\text{res}}(f)}(\text{Fun})$$

$$\frac{\Gamma|\Delta \vdash t_1 : I_1 \quad \dots \quad \Gamma|\Delta \vdash t_n : I_n \qquad \Delta\langle f \nabla I_1, \dots, I_n\rangle \vdash_f f : I}{\Gamma|\Delta \vdash f(t_1, \dots, t_n) : I}(\text{Init})$$

$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I \qquad I \subseteq \Delta_{\text{res}}(f)}{\Delta \vdash_f f : \Delta_{\text{res}}(f)}(\text{Base})$$

$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I' \qquad \Delta\{f \backslash (\Delta_{\text{args}}(f), \Delta_{\text{res}}(f) \nabla I')\} \vdash_f f : J}{\Delta \vdash_f f : J}(\text{Ind})$$

Fig. 6: Interval inference for recursive functions in mini-ACSL.

5.2 Example of Derivation

We illustrate our analysis by computing the derivation tree explicitly on a particular program. Fig. 7 shows the derivation tree for the term `f(50)` at line 6 in the example of Fig. 1, starting from an empty environment, and assuming that our widening operator satisfies the following equations

$$\bot \nabla [50,50] = [50,50] \qquad [50,50] \nabla [49,49] = [0,50] \qquad \bot \nabla [0,0] = \top.$$

These assumptions are not realistic for an actual choice of widening operator, but are taylor made for the example to converge quickly, so that we can construct a reasonably sized derivation tree. The derivation uses the following abstract environments and environments for logic functions:

$$\begin{array}{lll} \Gamma_1 = \{x : [50,50]\} & \Delta_1 = \{f : (\Gamma_1, \bot)\} & \Delta_2 = \{f : (\Gamma_1, [0, +\infty])\} \\ \Gamma_2 = \{x : [0,50]\} & \Delta_3 = \{f : (\Gamma_2, \bot)\} & \Delta_4 = \{f : (\Gamma_2, [0, +\infty])\} \\ \Gamma_3 = \{x : [1,50]\}. & & \end{array}$$

For the sake of simplicity, c denotes the condition `x <= 0`, r denotes the recursive term `f(x - 1)` and b denotes the body of the function, in such a way that $b =$ `c ? 0 : r + 1`. We also omit the environments in the abstract judgments for constants, and sometimes we also omit the whole judgment for constants, typically for most increment and decrement operations.

In this example, we can look at the environments Δ_i that appear in the derivation tree to understand how the fixpoints are computed both for the (unique) argument and the result of the function. The fixpoint for the argument is reached at the interval $[0, 50]$, while the fixpoint for the result is $[0, +\infty]$. This allows us to have an argument of type `int` in the generated code, but is not precise enough to store the result in an `int`: a GMP integer is required. This observation generalizes: In practice, for recursive functions, it is much more common that our analysis gives useful information on the arguments of a function

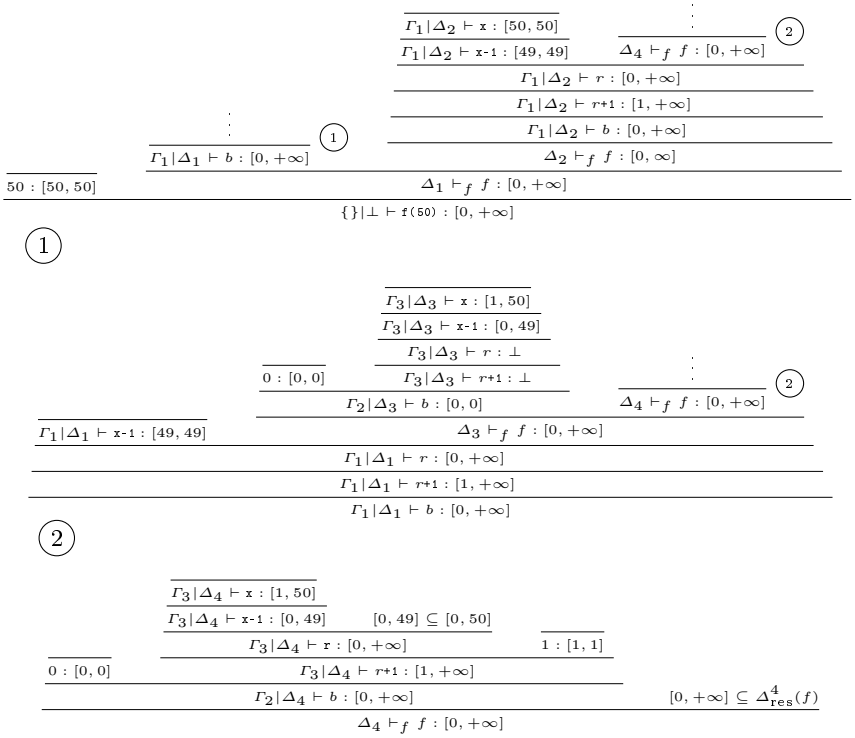

Fig. 7: Example of Interval Inference for a Recursive Function Call.

than on its output, and most of the time saved comes from performing internal computations with the arguments using machine integers. Indeed, in the presence of recursive functions, useful bounds for the results can unlikely be inferred.

5.3 Termination of the Static Analysis

With the strategy of always chosing the rule (Fun) over (Init) and (Base) over (Ind), our rule system is deterministic and defines an inference algorithm. This inference algorithm always terminates, as stated by the theorem below.

Theorem 1. *The rule system for intervals on mini-ACSL terms yields a terminating algorithm of interval inference.*

Proof (sketch). The proof is done by defining a well-founded order on the judgements, and showing that the judgements decrease for this order along any derivation tree. This order is defined as follows: first, we say that an environment Δ widens another one Δ' when, for every $f \in \mathrm{dom}\,(\Delta)$ and x in $\mathrm{dom}\,(\Delta_{\mathrm{args}}(f))$, there is an interval $I_{f,x}$ such that $\Delta_{\mathrm{args}}(f)(x) = \Delta'_{\mathrm{args}}(f)(x) \nabla I_{f,x}$ and there exists an interval I_f such that $\Delta_{\mathrm{res}}(f) = \Delta'_{\mathrm{res}}(f) \nabla I_f$. The chosen order relation on judgments is the lexicographic order induced by this relation and the relation of being a structural subterm:

$$\Gamma|\Delta \vdash t : \_ \prec \Gamma'|\Delta' \vdash u : \_ \Leftrightarrow \begin{cases} \Delta \neq \Delta' \text{ and } \Delta \text{ widens } \Delta' \\ \Delta = \Delta' \text{ and } t \text{ is a structural subterm of } u. \end{cases}.$$

We also establish by convention that

$\Gamma|\Delta \vdash t : \_ \prec \Delta' \vdash_f f : \_ \Leftrightarrow \Delta$ widens Δ'

$\Delta \vdash_f f : \_ \prec \Gamma|\Delta \vdash t : \_ \Leftrightarrow t$ is the body of f or a structural subterm of it.

This partial order $\prec$ is well-founded[2].

5.4 Interval Inference

Our rule system defines a deterministic inference algorithm that always terminates as stated in Theorem 1. Given an abstract environment Γ, we denote $\mathcal{I}(\Gamma, t)$ the result of this inference on the term t in environment $\Gamma|\bot$. However, we need to handle specifically the function's arguments that are widened. For such an argument t of a function f appearing in a term u representing a function call, we infer the result of the function call by building the derivation of $\Gamma|\bot \vdash f(t) : J$. In the corresponding derivation tree, consider the top-most application of the rule (FUN) for term u. It has necessarily an hypothesis of the form $\Gamma|\Delta \vdash t : I'$, where the interval I' is widened to the interval I associated to t in the environment Δ_{args}. We define $\mathcal{I}(\Gamma, t)$ to be this interval I for such function arguments. For instance, considering the term `f(50)` at line 6 of Fig. 1, for which the derivation tree is shown in Fig. 7, we have $\mathcal{I}(\{\}|\bot, 50) = [0, 50]$. Indeed, even though we first derive the interval $[50, 50]$ for its argument, it is later widened to $[0, 50]$ in the derivation tree, as witnessed in Δ_4.

As explained in the introduction, this paper extends the type system of [13] (and changes its theoretical framework for relying on abstract interpretation) in order to formalize the assumed static analysis of [3] and prove its assumptions, namely *type soundness* and *convergence*. The above-mentioned operator $\mathcal{I}$ matches the one of this latter paper. Theorem 1 ensures convergence, while soundness is proved in the next section.

5.5 Soundness of the Static Analysis

We now prove that the static analysis is sound. Since both the inference and the semantics require an environment, we first define a relation between such environments. We say that an interval environment Γ *abstracts* an environment for binders Ω, which is denoted $\Omega \triangleleft \Gamma$, if for every binder $x \in \mathrm{dom}\,(\Omega)$, we have $x \in \mathrm{dom}\,(\Gamma)$ and $\Omega_{\mathfrak{L}}(x) \in \Gamma(x)$. For a semantic environment $\Omega = (\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}})$, we define $\Omega \triangleleft \Gamma$ if and only if $\Omega_{\mathfrak{L}} \triangleleft \Gamma$. For a collecting environment Ξ, we say that Γ *abstracts* Ξ and we write $\Xi \blacktriangleleft \Gamma$ when $\Omega \triangleleft \Gamma$ for every $\Omega \in \Xi$.

Theorem 2. *For every mini-ACSL term t, every collecting environment Ξ, and every abstract environment Γ such that $\Xi \blacktriangleleft \Gamma$, we have $\mathcal{C}(\Xi, t) \subseteq \mathcal{I}(\Gamma, t)$.*

Proof (Sketch). The proof is done by induction. It is trivial without recursive definitions. With them, the main difficulty consists in finding the right invariants. For this, we provides a rule system, denoted $\Xi \vDash_\Delta t \in X$ and defining the set X of possible values for a term t in a collecting environment Ξ and an environment of logic functions Δ. When $\Delta = \bot$, it over-approximates the collecting semantics defined in Section 3.4 (i. e. if the judgment $\Xi \vDash_\Delta t \in X$ is derivable, then X contains the collecting semantics) and allows us to perform a per-case reasoning. The following Lemma gives the right invariants, proved by mutual induction.

[2] The proof details are in Appendix 7.

Lemma 1. *The judgments for the interval inference and fixpoint algorithm satisfy respectively each of the following property:*

1. *If the judgment $\Gamma|\Delta \vdash t : I$ is derivable in the abstract semantics, then for every collecting environment Ξ such that $\Xi \blacktriangleleft \Gamma$ and every derivation of the judgment $\Xi \vDash_{\Delta} t \in X$, we have $X \subseteq I$.*
2. *If the judgment $\Delta \vdash_f f : I$ is derivable in the abstract semantics, then denoting by b the body of the function, for every collecting environment Ξ such that $\Xi \blacktriangleleft \Delta_{\text{args}}$ and every derivation of the judgment $\Xi \vDash_{\Delta_{\text{res}}\{f\setminus I\}} b \in X$ in the collecting semantics augmented by $\Delta_{\text{res}}\{f\setminus I\}$, we have $X \subseteq I$.*

This theorem implies the soundness corollary below.

Corollary 1 (Interval Soundness). *For every mini-ACSL term t in an environment Ω such that there is a derivation of the semantics $\Omega \vDash t \Rightarrow z$, and for every abstract environment Γ such that $\Omega \lhd \Gamma$, we have $z \in \mathcal{I}(\Gamma, t)$.*

6 Experimental Evaluation

This section deals with the practical aspects of implementing our static analysis to analyse user-defined logic definitions and generate efficient monitors.

6.1 Practical Widening Operators

Our formal presentation is agnostic to the chosen widening operator, as long as it satisfies the properties mentioned in Section 4.1. However, in practice, the choice of this operator matters since it results in generating monitors with different efficiency. The choice is always a trade-off between efficiency and precision: depending on the widening operator, the fixpoint algorithm will converge in a small or large number of steps to a precise or unprecise interval. Our experimentation compares three different widening operators, presented below. The first two operators are extreme cases, which are only introduced for being compared against the third one, which is better and used by default in E-ACSL.

- The "naive widening", defined by the following formula

$$I_1 \nabla_{\text{naive}} I_2 = \begin{cases} I_2 & \text{if } I_1 = \bot \\ \top & \text{otherwise} \end{cases}$$

  This widening strategy makes the fixpoint reached in at most two iterations. Yet, it is extremely imprecise. In fact, it often returns $\top$ for recursive functions: only non-recursive functions are handled precisely.
- The "precise widening", defined by the following formula

$$I_1 \nabla_{\text{precise}} I_2 = \begin{cases} I_1 \vee I_2 & \text{if } I_1 \vee I_2 \subseteq [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}] \\ \top & \text{otherwise} \end{cases}$$

  This widening strategy is quite opposite to the naive one: it converges extremely slowly, but is very precise. In practice, the convergence is too slow for any practical application, and the monitor generation even takes too much time on minimal examples.

– The "smart widening", defined by $I_1 \nabla_{\text{smart}} I_2 = [a, b]$ where

$$a = \begin{cases} \min I_1 & \text{if } \min I_2 \geq \min I_1 \\ \mathrm{m}_{\texttt{int}} & \text{if } \mathrm{m}_{\texttt{int}} \leq \min I_2 \leq \min I_1 \\ -\infty & \text{otherwise;} \end{cases} \quad b = \begin{cases} \max I_1 & \text{if } \max I_2 \leq \max I_1 \\ \mathrm{M}_{\texttt{int}} & \text{if } \mathrm{M}_{\texttt{int}} \geq \max I_2 \geq \max I_1 \\ +\infty & \text{otherwise} \end{cases}$$

When the function is not decreasing, this operator leaves the lower bound unchanged. Otherwise, it directly approximates it to $\mathrm{m}_{\texttt{int}}$ if the lower bounds of both operands are bigger and goes to $-\infty$ otherwise. The behavior is similar for the upper bound and an increasing function. In practice, E-ACSL generalises it to a family of C types, and not only one by jumping from the boundary of one type to the other (e.g., from `int` to `long`).

6.2 Evaluation and Comparison of Widening Choices

The static analysis formalized in this paper is implemented within E-ACSL [20], the runtime assertion checker of Frama-C [1]. It supports the three widening operators of Section 6.1. It is used to optimize the code of the generated monitor, as formalized on the same mini-C and mini-ACSL languages in [3]. It is worth noting that the language supported by E-ACSL is much larger than mini-C and mini-ACSL [19], and so is the implementation of our static analysis.

We have run a few different examples to evaluate our static analysis and the widening strategy. The precise strategy is quite unusable even in simple tests since the monitor generation is dramatically slow in that case. Hence we only present the results of the experimental evaluation of the smart widening against the naive one. We ran the test on 4 different annotated C files[3]: `linear.c` contains definitions of typical recursive logic functions, where $f(n)$ is an affine function of $f(n-1)$, `fibonacci.c` contains the definition and various calls to the Fibonacci function, `mergesort.c` contains a C implementation of merge sort as well as a few recursive predicates that assert that the resulting array contains the same elements as the original one and is sorted, and finally `complex.c` contains arbitrary recursive functions with complex recursion schemes. We ran the benchmark on a laptop equipped with a 16-core AMD Ryzen 7 processor and 32GB of RAM. For each file, we measured the time for generating the monitor and for running it, with both the naive and the smart widening strategies. Each measure was performed with the hyperfine[4] software and repeated 10 times. The results are displayed in Fig. 8, where the mean of the 10 runs is written along with the standard deviation (all the durations are given in seconds). They are also compared to runs (named GMP) for which the static analysis was not used, so that only GMP operations are used at runtime. For each test case, the column `gen` is the time for generating the code, while the column `exe` is the time for executing the generated monitor. The last two lines show the gain of the smart widening operator with respect to using GMP only, or using the naive strategy. Fig. 9 graphically presents these results.

[3] source files and scripts of at https://thibautbenjamin.github.io/software/benchmarks-tap23.zip, the version of Frama-C/E-ACSL at https://thibautbenjamin.github.io/software/frama-c-tap23.zip.

[4] https://github.com/sharkdp/hyperfine

	`linear.c`		`fibonacci.c`		`mergesort.c`		`complex.c`	
	gen	exe	gen	exe	gen	exe	gen	exe
GMP	1.217 ± 0.008	96.034± 1.450	1.270 ± 0.007	75.617± 0.900	1.305 ± 0.009	71.196± 1.189	1.214 ± 0.006	`Fails`
naive	1.210 ± 0.007	95.866± 1.177	1.267 ± 0.005	75.454± 0.342	1.305 ± 0.007	62.170± 1.046	1.231 ± 0.010	52.453± 0.827
smart	1.207 ± 0.008	59.141± 0.363	1.294 ± 0.006	35.620± 0.366	1.300 ± 0.006	63.291± 0.552	1.217 ± 0.009	50.644± 0.217
vs. GMP	N/A	38%	N/A	53%	N/A	11%	N/A	`Fails`
vs. naive	N/A	38%	N/A	53%	N/A	N/A	N/A	N/A

Fig. 8: Experimental Evaluation With Different Widening Strategies.

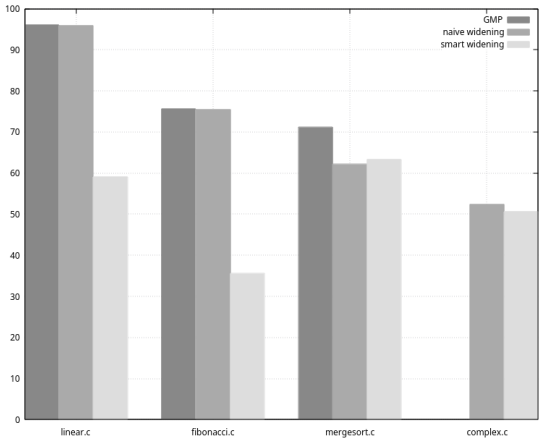

Fig. 9: Evaluation of Monitor Efficiency.

Overall, running the fixpoint algorithm with the smart widening as opposed to the naive one comes with no noticeable cost for the monitor generation. As already mentioned, the cost of generating the monitor using the precise widening is prohibitive on all these examples, and therefore not displayed here. In terms of efficiency when running the generated monitor, the smart widening performs significantly better on every case than with no analysis at all. In particular, on the file `complex.c`, without analysis, the generated program sometimes fails to execute properly because it is too resource intensive and exceeds the memory limit. The widening strategy is also significant: on the files `linear.c` and `fibonacci.c`, the smart widening performs respectively 38% and 53% better than the naive widening, which does not perform better than the systematic use of GMP. On the contrary, for the `mergesort` and `complex` examples, the smart widening and the naive widening leads to similar efficiency, which is better than the systematic use of GMP. Indeed, the file `complex.c` contains complicated recursion schemes, on which the heuristics implemented in the smart widening fail, and the file `mergesort.c` contains mostly calls to functions whose arguments are C variables, whose intervals are already fixpoints of the function.

6.3 Further Improvements to Widening

As illustrated by our evaluation, the widening strategy is important in practice. It is also impactful for the efficiency of monitor generator. The current "smart" strategy is based on the intuition that, in practice, the boundary of C types are likely to be values of importance, where the function may change behavior, and thus are good candidates for looking for fixpoints. Few other heuristics might also be used, even if not yet experimented with nor implemented. First, one could also widen to the boundary of C types plus (or minus for the lower bound) a small offset, in order to take into account typical off-by-one.This case might be frequent enough that adding those values to the candidates might give good results. Another possible improvement for the widening strategy could be to run a small syntactic analysis to look for important constants, and add those to our widening steps. For this idea to be viable, the analysis has to be very lightweight in order to ensure that it does not induce a significant overhead on the monitor generation. In all of these suggestions, we are adding more widening steps, which makes the convergence slower. It is likely that the most satisfying solution is to keep our "smart" widening strategy as a default, and run other more precise ones only on a case by case basis for the particular functions where the default is not good enough. It is possible in practice since E-ACSL allows choosing different widening strategies for different logic definitions. Last, we could also benefit from existing analysis on the C code, such as EVA [5], to gain precision of the C program variables used in the logic definitions.

7 Conclusion and Further Work

This article has presented a static analysis based on abstract interpretation for infering intervals in logic definitions used in formal code annotations. We have proved its termination and soundness properties and evaluated its practical efficiency, which depends on a widening strategy that have been discussed. It extends the work of [13] to logic definitions. This static analysis is used for generating efficient monitor for runtime assertion checking of arithmetic properties by allowing the code generator to soundly and efficiently choose between machine bounded integers and exact mathematical integers. How the monitors are generated based on our analysis is formalized in [3].

Three different widening strategies have been explored in this paper: investigating others strategies is left to future work, as well evaluating other abstract domains. Extending our formalization to rational numbers [13], memory properties [16], multi-state properties [18,12] or how to deal with undefined terms such as division by zero [9] is also left to future work. Our formalization effort would also greatly benefit from using a proof assistant, such as Coq [4]. Last, our static analysis might be complemented by a mechanism that would decide at runtime to use machine or mathematical integers. Such mechanisms already exist on top of exact arithmetic libraries, e.g., ZArith[5] for OCaml.

[5] https://github.com/ocaml/Zarith/

References

1. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. Communications of the ACM (2021)
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Tech. rep., CEA List and Inria, https://frama-c.com/download/acsl.pdf
3. Benjamin, T., Signoles, J.: Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates. In: Symposium on Applied Computing (2023)
4. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
5. Blazy, S., Bühler, D., Yakobowski, B.: Structuring Abstract Interpreters through State and Value Abstractions. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17) (2017)
6. Cheon, Y.: A runtime assertion checker for the Java Modeling Language. Ph.D. thesis, Iowa State University (2003)
7. Clarke, L.A., Rosenblum, D.S.: A Historical Perspective on Runtime Assertion Checking in Software Development. SIGSOFT Software Engineering Notes (2006)
8. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2022)
9. Delahaye, M., Kosmatov, N., Signoles, J.: Common Specification Language for Static and Dynamic Analysis of C Programs. In: Symposium on Applied Computing (SAC) (2013)
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems (2013)
11. Filliâtre, J.C., Pascutto, C.: Ortac: Runtime Assertion Checking for OCaml (tool paper). In: International Conference on Runtime Verification (RV) (2021)
12. Filliâtre, J.C., Pascutto, C.: Optimizing Prestate Copies in Runtime Verification of Function Postconditions. In: International Conference on Runtime Verification (RV) (2022)
13. Kosmatov, N., Maurica, F., Signoles, J.: Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In: International Conference on Runtime Verification (RV) (2020)
14. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design (1999)
15. Lehner, H.: A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking. Ph.D. thesis, ETH Zurich (2011)
16. Ly, D., Kosmatov, N., Loulergue, F., Signoles, J.: Verified Runtime Assertion Checking for Memory Properties. In: International Conference on Tests and Proofs (TAP) (2020)
17. Ly, D., Kosmatov, N., Loulergue, F., Signoles, J.: Soundness of a dataflow analysis for memory monitoring. In: Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT) (2018)
18. Signoles, J.: The E-ACSL Perspective on Runtime Assertion Checking. In: International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX) (2021)
19. Signoles, J.: E-ACSL Version 1.18. Implementation in Frama-C Plug-in E-ACSL 26.1 (2022), http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf

20. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In: International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES) (2017)

Appendix: Proof of the theorems

This appendix contains the proof of the technical results contained in the article as well as the additional lemmas needed to prove them.

Explicit Description of the Collecting Semantics

We recall the definition of the collecting semantics

$$\mathcal{C}(\Xi, t) \equiv \{z \mid \exists \Omega_{\mathfrak{L}} \in \Xi, \exists \Omega_{\mathcal{V}} : \mathcal{V} \to [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}], \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t \Rightarrow z\}.$$

Fig. 10 gives an explicit set of rules that computes the collecting semantics. We denote $\Xi \vDash t \in X$ the judgment for this rule system, and we show it to be equivalent to the collecting semantics. These rules are the ones we use in practice, when we work with the collecting semantics. To introduce these rules, we use some notations: Given an arithmetic operator $\diamond$, we introduce the corresponding operator $\diamond_{\mathrm{set}}$ on $\mathcal{P}(\mathbb{Z})$, defined by the following formula:

$$X \diamond_{\mathrm{set}} Y = \{x \diamond y \mid x \in X \text{ and } y \in Y\}$$

We also define, given a collecting environment Ξ, a new collecting environment $\{x_1 \backslash t_1, \ldots, x_n \backslash t_n\}^{\Xi}$, defined by

$$\{x_1 \backslash t_1, \ldots, x_n \backslash t_n\}^{\Xi} = \{\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \mid z_1 \in \mathcal{C}(\Xi, t_1), \ldots, z_n \in \mathcal{C}(\Xi, t_n)\}$$

$$\frac{}{\Xi \vDash z \in \{z\}} \qquad \frac{}{\Xi \vDash x \in \{\Omega_{\mathfrak{L}}(x) \mid \Omega_{\mathfrak{L}} \in \Xi\}}$$

$$\frac{}{\Xi \vDash v \in [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]}$$

$$\frac{\Xi \vDash t \in X \qquad \Xi \vDash t' \in X'}{\Xi \vDash t \ \diamond \ t' \in X \ \diamond \ X'}$$

$$\frac{\Xi \vDash t \in X \qquad \Xi \vDash t' \in X'}{\Xi \vDash p \ ? \ t \ : \ t' \in X \cup X'}$$

$$\frac{\mathfrak{F}(f) = (x_1, \ldots, x_n; b) \qquad \{x_i \backslash t_i\}^{\Xi} \vDash b \in X}{\Xi \vDash f(t_1, \ldots, t_n) \in X}$$

Where the collecting environment $\{x_i \backslash t_i\}^{\Xi}$ is defined as

$$\{x_i \backslash t_i\}^{\Xi} = \{\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \mid \exists \Omega, \Omega \lhd \Xi \text{ and } \forall i \in \{1, \ldots, n\}, \Omega \vDash t_i \in z_i\}$$

Fig. 10: Rules for the Collecting Semantics of the mini-ACSL Language

Theorem 3. *The rules given in Fig. 10 compute an over-approximation of the collecting semantics. If a judgment $\Xi \vDash t \in X$ then necessarily $\mathcal{C}(\Xi, t) \subseteq X$.*

Proof. We proceed by structural induction on the term.

- If the term is a constant z, its concrete semantics is given by the rule

$$\frac{}{\Omega \vDash z \Rightarrow z}$$

  Hence $\mathcal{C}(\Xi, z) = \{z\}$, which is given exactly by the explicit rule for constants.
- If the term is a binder x, then its semantics is obtained by application of the following rule

$$\frac{\Omega_{\mathfrak{L}}(x) = z}{\Omega \vDash x \Rightarrow z}$$

  Hence the we have the collecting semantics $\mathcal{C}(\Xi, x) = \{\Omega_{\mathfrak{L}}(x) \mid \Omega_{\mathfrak{L}} \in \Xi\}$, which corresponds exactly to the explicit rule for binders.
- If the term is a mini-C variable v, then its semantics is obtained by application of the following rule

$$\frac{x \in \mathsf{Int} \qquad \Omega_{\mathcal{V}}(v) = x}{\Omega \vDash v \Rightarrow \dot{x}}$$

  Since we are free to chose an environment Ω where $\Omega_{\mathcal{V}}(v)$ takes any arbitrary value in $[\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]$, the collecting semantics is given by $\mathcal{C}(\Xi, v) = [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]$, which corresponds exaclty to the rule given for mini-C variables.
- If the term is the application of an arithmetic operator of the form $t \diamond t'$, then its semantics is necessarily obtained by application of the following rule

$$\frac{\Omega \vDash t \Rightarrow z \qquad \Omega \vDash t' \Rightarrow z' \qquad \mathrm{not}(\diamond = / \text{ and } z' = 0)}{\Omega \vDash t \diamond t' \Rightarrow z \diamond z'}$$

  Thus, we have the equality

$$\mathcal{C}(\Xi, t \diamond t') = \{z \diamond z' \mid z \in \mathcal{C}(\Xi, t), z' \in \mathcal{C}(\Xi, t')\} = \mathcal{C}(\Xi, t) \diamond_{\mathrm{set}} \mathcal{C}(\Xi, t')$$

  This is exactly given by the explicit rule for operators.
- If the term is a conditional of the form $p \ \mathtt{?}\ t \ \mathtt{:}\ t'$, then the semantics is obtained by either of the following rules

$$\frac{\Omega \vDash p \Rightarrow 1 \qquad \Omega \vDash t \Rightarrow z}{\Omega \vDash p \ \mathtt{?}\ t \ \mathtt{:}\ t' \Rightarrow z} \qquad\qquad \frac{\Omega \vDash p \Rightarrow 0 \qquad \Omega \vDash t' \Rightarrow z'}{\Omega \vDash p \ \mathtt{?}\ t \ \mathtt{:}\ t' \Rightarrow z'}$$

  Thus the collecting semantics is obtained satisfies the equation

$$\mathcal{C}(\Xi, p \ \mathtt{?}\ t \ \mathtt{:}\ t') = A \cup B$$

$$\text{where } \begin{cases} A = & \{z \mid \exists \Omega_{\mathfrak{L}} \in \Xi, \exists \Omega_{\mathcal{V}}, \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash p \Rightarrow \mathrm{T} \text{ and } \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t \Rightarrow z\} \\ B = & \{z \mid \exists \Omega_{\mathfrak{L}} \in \Xi, \exists \Omega_{\mathcal{V}}, \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash p \Rightarrow \mathrm{F} \text{ and } \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t' \Rightarrow z\} \end{cases}$$

By definition of the collecting semantics, we have $A \subseteq \mathcal{C}(\Xi, t)$ and $B \subseteq \mathcal{C}(\Xi, t')$, thus $\mathcal{C}(\Xi, p \,?\, t : t') \subseteq \mathcal{C}(\Xi, t) \cup \mathcal{C}(\Xi, t')$. Suppose that $\Xi \vDash p \,?\, t : t' \in X \cup X'$ is derivable from derivations of $\Xi \vDash t \in X$ and $\Xi \vDash t' \in X'$, then we have by induction $\mathcal{C}(\Xi, t) \subseteq X$ and $\mathcal{C}(\Xi, t') \subseteq X'$. This gives the following inclusion

$$\mathcal{C}(\Xi, p \,?\, t : t') \subseteq \mathcal{C}(\Xi, t') \cup \mathcal{C}(\Xi, t') \subseteq X \cup X' = \mathcal{C}(\Xi, p \,?\, t : t')$$

– If the term is a function call of the form $f(t_1, \ldots, t_n)$, with $\mathfrak{F}(f) = (x_1, \ldots, x_n; b)$, then the semantics is obtained by application of the following rule

$$\frac{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t_1 \Rightarrow z_1 \qquad \ldots \qquad \Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash t_n \Rightarrow z_n \qquad \Omega_{\mathcal{V}}, \{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \vDash b \Rightarrow z}{\Omega_{\mathcal{V}}, \Omega_{\mathfrak{L}} \vDash f(t_1, \ldots, t_n) \Rightarrow z}$$

This gives exactly the following inductive relation satisfied by the collecting semantics $\mathcal{C}(\Xi, f(t_1, \ldots, t_n)) = \mathcal{C}(\{x_i \backslash t_i\}^{\Xi}, f(t_1, \ldots, t_n))$. The explicit rule for function gives exactly this relation.

We thus have proved that the explicit rules define satisfy the exact same inductive relation as the collecting semantics, except for the conditional terms, where they over-approximate the relation. Hence these rules over-approximate the collecting semantics.

For the rest of the proofs, when we mention the collecting semantics, we refer to this explicit set of rules, which is slightly less precise but sufficient for our purpose.

Properties of the Abstract Semantics

Theorem 1. *The rule system for intervals on mini-ACSL terms yields a terminating algorithm of interval inference.*

Proof. We define an order relation on the interval judgments that does not allow for infinite decreasing chains, and show that throughout the construction of a derivation tree by the inference algorithm, the judgments are always decreasing for this order. This shows that the inference algorithm only tries to construct finite derivation trees, and thus it terminates. Indeed, either it manages to construct the valid tree, or it fails to do so, which can be observed in finite time since the tree is finite. First, we say that an environment Δ widens another one Δ' when for every $f \in \mathrm{dom}\,(\Delta)$, for every x in $\mathrm{dom}\,(\Delta_{\mathrm{args}}(f))$, there exists an interval $I_{f,x}$ such that $\Delta_{\mathrm{args}}(f)(x) = \Delta'_{\mathrm{args}}(f)(x) \nabla I_{f,x}$ and there exists an interval I_f such that $\Delta_{\mathrm{res}}(f) = \Delta'_{\mathrm{res}}(f) \nabla I_f$. The order relation that we consider on judgments is the lexicographic order induced by this relation and the relation of being a structural subterm:

$$\Gamma | \Delta \vdash t : \_ \prec \Gamma' | \Delta' \vdash u : \_ \Leftrightarrow \begin{cases} \Delta \neq \Delta' \text{ and } \Delta \text{ widens } \Delta' \\ \Delta = \Delta' \text{ and } t \text{ is a structural subterm of } u \end{cases}.$$

We also establish by convention that

$$\Gamma|\Delta \vdash t : \_ \prec \Delta' \vdash_f f : \_ \Leftrightarrow \Delta \text{ widens } \Delta'$$
$$\Delta \vdash_f f : \_ \prec \Gamma|\Delta \vdash t : \_ \Leftrightarrow t \text{ is the body of } f \text{ or a structural subterm of it}$$

We first show that this order relation does not allow for infinite decreasing sequences, since it is constructed as the lexicographic product of two orders that do not allow for infinite decreasing sequences. First it is clear that the relation of being a structural subterm does not allow for such sequences: by construction a term is constructed from a finite amount of data. So it suffices to check that the relation of widening on environments does not allow for infinitely decreasing sequences. This is a direct consequence of the property (W2) about the widening operator. Hence this relation on judgments prevents the existence of infinite decreasing sequences.

We now check that during the run of the interval inference algorithm, the derivation tree that is being constructed is such that all the judgments are decreasing as we go from the conclusion to the premices. There is nothing to prove for the three rules without premices, and the result is immediate for the inference rules for arithmetic operation and conditionals since the premices are all in the same environment and in a structural subterm of the conclusion. So we just have to check that this holds for every application of the rules for functions (Fun), (Init), (Base) and (Ind). In the rules (Fun) and (Base), all the judgments that appear in the premices are smaller than the conclusion, because they concern structural subterms of it. There is one judgment which does not concern a structural subterm in both the rule (Init) and (Ind). They are respectively the judgments $\Delta\langle f \nabla I_1, \ldots, I_n\rangle \vdash_f f : I$ and $\Delta_{\text{args}}\{f \backslash I \nabla I'\} \vdash_f f : J$. We show that in both of these judgment, the environment widens Δ. This is a consequence of our strategy of always trying the rules (Fun) and (Base) over (Init) and (Ind). Indeed, in every application of either of these rules, the rules (Fun) and (Base) cannot apply, thus the inclusion of the intervals is not satisfied, and hence the environment constructed for these judgments is different from the one in the conclusion, by (W1). By construction, this environment thus widens the one from the conclusion, and the judgment is thus smaller.

Lemma 2. *Consider a binary operation $\star$ on $\mathcal{P}(\mathbb{Z})$ preserving the inclusion relation, along with two subsets X, X' of $\mathbb{Z}$ and two intervals I, I' such that $X \subseteq I$ and $X' \subseteq I'$, then we have $X \star X' \subseteq I \star^\sharp I'$.*

Proof. For the sake of this proof, we write the concretization map γ explicitly. Our assumption is thus $X \subseteq \gamma(I)$ and $X' \subseteq \gamma(I')$. Since the operation $\star$ preserves the inclusion realtion, we have $X \star X' \subseteq \gamma(I) \star (I')$. Since α is a morphism of lattice, it preserves the inclusion, thus $\alpha(X \star X') \subseteq \alpha(\gamma(I) \star \gamma(I')) = I \star^\sharp I'$. Since (α, γ) is a Galois connection, this is equivalent to $X \star X' \subseteq \gamma(I \star^\sharp I')$.

Lemma 3. *For every derivation of a judgment of the form $\Delta \vdash_f f : I$, we have $\Delta_{\text{res}}(f) \subseteq I$*

Proof. We prove this result by induction of the derivation

- For a derivation obtained by application of the rule (BASE) of the following form
$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I \qquad I \subseteq \Delta_{\text{res}}(f)}{\Delta \vdash_f f : \Delta_{\text{res}}(f)}$$
we have $\Delta_{\text{res}}(f) \subseteq \Delta_{\text{res}}(f)$.
- For a derivation obtained by application of the rule (IND) of the following form
$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I' \qquad \Delta_{\text{res}}(f) = I \qquad \Delta_{\text{args}}\{f \backslash I \nabla I'\} \vdash_f f : J}{\Delta \vdash_f f : J}$$
we have by induction hypothesis that $I \nabla I' \subseteq J$. By the property (W1) of the widening $I \subseteq I \nabla I'$, and thus $I \subseteq J$.

Given an environment for functions Δ, we define the collecting semantics augmented by Δ to be the semantics expressed with the judgment $\Xi \vDash t \in X$ given with the explicit rules, to which we add, for every function f, with arguments $x_1, \ldots, x_n$, the following rule
$$\frac{\Xi \vDash_\Delta t_1 \in \Delta_{\text{args}}(f)(x_1) \qquad \ldots \qquad \Xi \vDash_\Delta t_n \in \Delta_{\text{args}}(f)(x_n)}{\Xi \vDash_\Delta f(t_1, \ldots, t_n) \in \Delta_{\text{res}}(f)}$$
The addition of these rules may result in an undeterministic rule system, since these newly added rules may overlap with the rule for functions that is already present. We use the convention that the rules coming from Δ always take precedence over the rule for functions.

Lemma 4. *The judgments for the interval inference and fixpoint algorithm satisfy respectively each of the following property:*

1. *If the judgment $\Gamma|\Delta \vdash t : I$ is derivable in the abstract semantics, then for every collecting environment Ξ such that $\Xi \blacktriangleleft \Gamma$ and every derivation of the judgment $\Xi \vDash_\Delta t \in X$, we have $X \subseteq I$.*
2. *If the judgment $\Delta \vdash_f f : I$ is derivable in the abstract semantics, then denoting by b the body of the function, for every collecting environment Ξ such that $\Xi \blacktriangleleft \Delta_{\text{args}}$ and every derivation of the judgment $\Xi \vDash_{\Delta_{\text{res}}\{f \backslash I\}} b \in X$ in the collecting semantics augmented by $\Delta_{\text{res}}\{f \backslash I\}$, we have $X \subseteq I$.*

Proof. We prove those two propoerties together by mutual induction on the derivation, following the induction scheme given by the ordering in the judgments that we gave in the proof of Theorem 1.

1. Consider a derivation of the judgment $\Gamma|\Delta \vdash t : I$, together with a collecting environment Ξ such that $\Xi \blacktriangleleft \Gamma$ and a derivation of $\Xi \vDash t \Rightarrow X$ in the concrete semantics augmented by Δ. We show that $X \subseteq I$ by induction on the derivation of the interval judgment.

– If the abstract semantics is obtained by application of the rule for constant, then the term t is the constant z. We have an application of the two following rules for the collecting and the bastract semantics

$$\frac{}{\Xi \vDash z \in \{z\}} \qquad \frac{}{\Gamma|\Delta \vdash z : [z, z]}$$

– If the abstract semantics is obtained by application of the rule for binders, then the term is a binder x, then its collecting semantics and its abstract semantics are respectively obtained by the following rules

$$\frac{}{\Xi \vDash x \in \{\Psi(x) \mid \Psi \in \Xi\}} \qquad \frac{}{\Gamma|\Delta \vdash x : \Gamma(x)}$$

By hypothesis, we have $\Xi \blacktriangleleft \Gamma$, thus for every $\Psi \in \Xi$, we have $\Psi(x) \in \Xi(x)$. This proves that $\{\Psi(x) \mid \Psi \in \Xi\} \subseteq \Xi(x)$.

– If the abstract semantics is obtained by application of the rule for mini-C variables, then the term t is a mini-C variable v, and its collecting semantics and its abstract semantics are respectively obtained by application of the following rules

$$\frac{}{\Xi \vDash v \in [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]} \qquad \frac{}{\Gamma|\Delta \vdash x : [\mathrm{m}_{\mathtt{int}}, \mathrm{M}_{\mathtt{int}}]}$$

– If the abstract semantics is obtained by application of the rule for operations, then the term is the application of an arithmetic operator of the form $u \diamond u'$, then its collecting semantics and its abstract semantics are respectively obtained by application of the following rules

$$\frac{\Omega \vDash u \in Y \qquad \Omega \vDash u' \in Y'}{\Omega \vDash u \diamond u' \in Y \diamond Y'} \qquad \frac{\Gamma|\Delta \vdash u : J \qquad \Gamma|\Delta \vdash u' : J'}{\Gamma|\Delta \vdash u \diamond u' : J \diamond^\sharp J'}$$

By induction hypothesis, we have $Y \subseteq J$ and $Y' \subseteq J'$. Since the operation $\diamond$ on $\mathcal{P}(\mathbb{Z})$ preserves the inclusion, Lemma 2 implies $Y \diamond Y' \subseteq J \diamond^\sharp J'$.

– If the abstract semantics is obtained by application of the rule for conditionals, then the term is a conditional of the form $p \ \mathtt{?}\ u \ \mathtt{:}\ u'$, and the collecting semantics and the abstract semantics are respeictively obtained by application of the following rules

$$\frac{\Xi \vDash t \in Y \qquad \Xi \vDash t' \in Y'}{\Xi \vDash p \ \mathtt{?}\ t \ \mathtt{:}\ t' \in Y \cup Y'} \qquad \frac{\Gamma|\Delta \vdash t : J \qquad \Gamma|\Delta \vdash t' : J'}{\Gamma|\Delta \vdash p \ \mathtt{?}\ t \ \mathtt{:}\ t' : J \cup^\sharp J'}$$

By induction hypothesis, we have $Y \subseteq J$ and $Y' \subseteq J'$. Since the $\cup$ operation on $\mathcal{P}(Z)$ preserves the inclusion, Lemma 2 implies that $Y \cup Y' \subseteq J \cup^\sharp J'$.

- If the abstract semantics is obtained by application of the rule (FUN) of the following form

$$\frac{\Gamma|\Delta \vdash t_1 : I_1 \qquad \ldots \qquad \Gamma|\Delta \vdash t_n : I_n \qquad \forall i, I_i \subseteq \Delta_{\text{args}}(x_i)}{\Gamma|\Delta \vdash f(t_1, \ldots, t_n) : \Delta_{\text{res}}(f)}$$

  From the rules of the collecting semantics augmented by Δ, such a derivation always comes from premises of the form $\Omega \vDash t_1 \Rightarrow Y_1, \ldots, \Omega \vDash t_n \Rightarrow Y_n$. By induction hypothesis, we necessarily have $Y_1 \subseteq I_1, \ldots, Y_n \subseteq I_n$. By using the last premise of our application of the rule (FUN), this implies that we have $Y_1 \subseteq \Delta_{\text{args}}(x_1), \ldots, Y_n \subseteq \Delta_{\text{args}}(x_n)$. The precedence of the rules from Δ in the augmented collecting semantics implies that the derivation of $\Omega \vDash_\Delta f(t_1, \ldots, t_n) \in X$ necessarily comes from a rule given by Δ, which implies that $X = \Delta_{\text{res}}(f)$.
- If the abstract semantics is obtained by application of the rule (INIT) of the following form

$$\frac{\Gamma|\Delta \vdash t_1 : I_1 \qquad \ldots \qquad \Gamma|\Delta \vdash t_n : I_n \qquad \Delta\langle f \nabla I_1, \ldots, I_n\rangle \vdash_f f : I}{\Gamma|\Delta \vdash f(t_1, \ldots, t_n) : I}$$

  Then the judgment in the collecting semantics augmented by Δ is obtained either by application of a rule of the collecting semantics, or of a rule coming from Δ. If the applied rule comes from Δ, then this implies that $X = \Delta_{\text{res}}(f)$ and thus by Lemma 3, we have $X \subseteq I$. If the derivation is obtained by application of the function rule of the collecting semantics, then we have a derivation for all the following premices in the collecting semantics augmented by Δ: $\Xi \vDash_\Delta t_1 \in Y_1, \ldots, \Xi \vDash_\Delta t_n \in Y_n$ and $\{x_i \backslash z_i\}^\Xi \vDash_\Delta b \in X$. By induction hypothesis, we have $Y_1 \subseteq I_1, \ldots, Y_n \subseteq I_n$. This implies in particular that $\{x_i \backslash I_i\}^\Xi \blacktriangleleft \Delta\langle f \nabla I_1, \ldots, I_n\rangle$. Note that the inlcusion of the intervals given by the widening relation $\Delta_{\text{args}}(f)(x_i) \subseteq \Delta_{\text{args}}(f)(x_i) \nabla I_i$ and Lemma 3 imply that any rule in the collecting semantics augmented by Δ is also valid in the collecting semantics augmented by $\Delta\langle f \nabla I_1, \ldots, I_n\rangle\{f \backslash I\}$. This implies in particular that we have a derivation of $\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \vDash_\Delta b \in X$ in the collecting semantics augmented by $\Delta\langle f \nabla I_1, \ldots, I_n\rangle\{f \backslash I\}$ and thus by the mutual induction hypothesis, $X \subseteq I$.

2. Consider a function f with body b, such that we have a derivation of the judgment $\Delta \vdash_f f : I$, together with a collecting environment Ξ such that $\Xi \blacktriangleleft \Delta_{\text{args}}(f)$, and a derivation of $\Xi \vDash_{\Delta_{\text{args}}\{f \backslash I\}} b \in X$ in the collecting semantics augmented by $\Delta_{\text{args}}\{f \backslash I\}$. We show by induction on the derivation tree of the fixpoint judgment that $X \subseteq I$.
   - For a derivation obtained by application of the rule (BASE) of the following form

$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I \qquad I \subseteq \Delta_{\text{res}}(f)}{\Delta \vdash_f f : \Delta_{\text{res}}(f)}$$

     By mutual induction hypothesis, we have $X \subseteq I$. Since $I \subseteq \Delta_{\text{res}}(f)$, this implies $X \subseteq \Delta_{\text{res}}(f)$.

– For a derivation obtained by application of the rule (IND) of the following form

$$\frac{\Delta_{\text{args}}(f)|\Delta \vdash b : I' \qquad \Delta_{\text{res}}(f) = I \qquad \Delta_{\text{args}}\{f \backslash I \nabla I'\} \vdash_f f : J}{\Delta \vdash_f f : J}$$

We have by induction hypothesis on the last premice that $X \subseteq J$.

Theorem 2. *For every mini-ACSL term t, every collecting environment Ξ, and every abstract environment Γ such that $\Xi \blacktriangleleft \Gamma$, we have $\mathcal{C}(\Xi, t) \subseteq \mathcal{I}(\Gamma, t)$.*

Proof. Since the oracle $\mathcal{I}$ always return a superset of or the interval derivation, it suffices to show that this property holds for the interval derivation. This is a special case of Lemma 4, taking Δ to be the empty environment $\bot$, in which case the collecting semantics augmented by Δ is simply the collecting semantics.

Corollary 1 (Interval Soundness). *For every mini-ACSL term t in an environment Ω such that there is a derivation of the semantics $\Omega \vDash t \Rightarrow z$, and for every abstract environment Γ such that $\Omega \triangleleft \Gamma$, we have $z \in \mathcal{I}(\Gamma, t)$.*

Proof. Consider the collecting interval interval $\{\Omega_{\mathfrak{L}}\}$, which satisfies both $\Omega \triangleleft \{\Omega_{\mathfrak{L}}\}$ and $\{\Omega_{\mathfrak{L}}\} \blacktriangleleft \Gamma$. By application of Theorem 2, we thus have $z \in \mathcal{C}(\{\Omega_{\mathfrak{L}}\}, t) \subseteq \mathcal{I}(\Gamma, t)$.