Optimized Runtime Verification of ACSL Arithmetic Primitives

FÉLIX RIDOUX[*], École Normale Supérieure de Rennes, France and Université Paris-Saclay, CEA, LIST, France

ACSL is an annotation language used for the runtime assertion checking of **C** programs. Its semantics is defined over mathematical numbers, as opposed to bounded machine numbers. This leads to costly implementations using multi-precision packages. We formalize, prove and implement arithmetic primitives of **ACSL**. This implementation performs a static analysis to discover an over-approximation of the actual range of integer variables in order to safely implement them as machine numbers as often as it is possible. It is integrated inside the plug-in **E-ACSL** of **Frama-C**.

Additional Key Words and Phrases: Runtime Assertion Checking, Dynamic Analysis, Typing, Optimized Code Generation

1 INTRODUCTION

Runtime verification consists in checking during the program execution if a given property is verified [Leucker and Schallhart 2009]. It is opposed to *static verification* which consists in checking if a property holds for all executions without ever executing the program [Rival and Yi 2020]. For instance, runtime verification can check if a loop invariant is verified during an execution. It does not prove that the program satisfies the invariant, only that this execution does. If runtime verification finds that the invariant is violated, it proves that the program does not satisfy the invariant; either the program or the invariant is false. If the invariant is expressed in a less artificial language than the program (say logic vs. Javascript), it is highly possible that it is the program that is false.

In this work, we are interested in the technique called *Runtime Assertion Checking* (**RAC**) [Clarke and Rosenblum 2006]. It consists in inserting annotations in the source code, like, for instance, assertions. Usually, these assertions are expressed in propositional logic. This allows to use *quantifiers* for modeling loops. They also distinguishes program variables from *logical variables*. The latter are introduced by binding constructs like quantifiers or let expressions, and the former are used as objects in assertions. If during an execution, all the assertions are verified, then the annotated program has the same semantics as the original one, so checking the assertions is *transparent* for the program. However, if at least one assertion fails, **RAC** makes the program signal an error and stop. We are interested in implementing **RAC** assertion checking for **C** program.

E-ACSL [Signoles et al. 2017] is a **RAC** tool that allows to express properties over *mathematical numbers*. It is part of the **Frama-C** program analysis toolbox for **C** programs [Kirchner et al. 2015]. **E-ACSL** is an implementation of a specification language called **ACSL** [Baudin et al. 2020]. Its operating principle is to translate a **C** program annotated with **ACSL** assertions into another **C** program which has the same semantics if the assertions are verified or signals an error otherwise. The new **C** program essentially consists in the original one where **ACSL** annotations have been replaced by their **C** translation.

Why does **E-ACSL** work over mathematical numbers, instead of program numbers? There are two approaches for dealing with numbers in assertions. The first one uses the programming language's semantics (**C** in our case); i.e., the integers are represented using fixed-length bit encodings, and

[*]Work performed at the Software Safety and Security Laboratory (LSL) of CEA LIST under the supervision of Julien Signoles.

Author's address: Félix Ridoux, Department of Computer Science, École Normale Supérieure de Rennes, Bruz, France , Université Paris-Saclay, CEA, LIST, Palaiseau, France, felix.ridoux@ens-rennes.fr.

```
long __gen_e_acsl_k;
long __gen_e_acsl_one;
int __gen_e_acsl_cond;
__e_acsl_mpz_t __gen_e_acsl_lambda;
__e_acsl_mpz_t __gen_e_acsl_sum;
__e_acsl_mpz_t __gen_e_acsl_;
int __gen_e_acsl_eq;
__gen_e_acsl_one = 1;
__gen_e_acsl_cond = 0;
__gmpz_init_set_si(__gen_e_acsl_lambda,0L);
__gmpz_init_set_si(__gen_e_acsl_sum,0L);
__gen_e_acsl_k = (long)a;
while (1) {
  __gen_e_acsl_cond = __gen_e_acsl_k > (long)b;
  if (__gen_e_acsl_cond) break;
  else { {
    __e_acsl_mpz_t __gen_e_acsl_1;
    __gmpz_init_set_si(__gen_e_acsl_1,c *
        __gen_e_acsl_k);
    __gmpz_set(__gen_e_acsl_lambda, __gen_e_acsl_1);
    __gmpz_clear(__gen_e_acsl_1); }
    __gmpz_add(__gen_e_acsl_sum,__gen_e_acsl_sum,
        __gen_e_acsl_lambda);
    __gen_e_acsl_k += __gen_e_acsl_one;
} }
__gmpz_init_set_si(__gen_e_acsl_,nL);
__gen_e_acsl_eq = __gmpz_cmp(__gen_e_acsl_sum,
    __gen_e_acsl_);
assert(__gen_e_acsl_eq == 0);
__gmpz_clear(__gen_e_acsl_lambda);
__gmpz_clear(__gen_e_acsl_sum);
__gmpz_clear(__gen_e_acsl_);
```

(a)

```
int __gen_e_acsl_k;
int __gen_e_acsl_one;
int __gen_e_acsl_cond;
int __gen_e_acsl_lambda;
int __gen_e_acsl_sum;
__gen_e_acsl_one = 1;
__gen_e_acsl_cond = 0;
__gen_e_acsl_lambda = 0;
__gen_e_acsl_sum = 0;
__gen_e_acsl_k = a;
while (1) {
  __gen_e_acsl_cond = __gen_e_acsl_k > b;
  if (__gen_e_acsl_cond) break;
  else {
    __gen_e_acsl_lambda = c * __gen_e_acsl_k;
    __gen_e_acsl_sum += __gen_e_acsl_lambda;
    __gen_e_acsl_k += __gen_e_acsl_one;
} }
assert(__gen_e_acsl_sum == n);
```

(b)

Fig. 1. `logical_assert(\sum(a, b, \lambda k; c*k)==n)`: **C** code for (a) large, and (b) small integers.

therefore are bounded. The second one uses the mathematical semantics of integers; i.e., they are unbounded. The advantage of the first approach is its time efficiency, since the operations on bounded integers are native in **C**, whereas unbounded integers are implemented in software. For instance, the **GMP** library, which is used in this work, is one order of magnitude less efficient as native **C** operations. Furthermore, it is closer to the underlying **C** program semantics. However, an issue is that the developers may have in mind plain unbounded arithmetic when they write annotations (e.g., compute the average of a set of numbers). Furthermore, even when they deal with bounded integers they may prefer to express annotations in a way that improves formal readability even if it incurs computing with unbounded integers. For instance, a program may use a contrived way to compute the average of a large set of numbers, only to avoid overflowing; but an annotation may simply express that the goal is to compute the sum of all numbers, and divide it by the size of the set, ignoring all possibilities of overflow, in a pure mathematical way. This issue is addressed by the second approach, paying the price of a less efficient implementation of integer operations.

The goal of our work is to implement a part of **ACSL** that is still missing in **E-ACSL**, the *extended quantifiers*, and to reconcile the two approaches for dealing with integers in this implementation; that is to offer the expressivity of unbounded integers at the price of bounded integers (as often as possible). Consider for instance assertion `logical_assert(\sum(a, b, \lambda k;c*k)==0)` which expresses that the sum of all integers from `a` to `b` multiplied by `c` must be equal to the integer n. The naïve code generated for the unbounded semantics is shown in Fig. 1.(a). However, if one recognizes that `a`, `b` and `c` are small enough, the code of Fig. 1.(b) could be generated instead. It contains less operations, and they are simpler (simple assignments instead of function calls) than for the naïve code.

To help implementing that behavior, **E-ACSL** offers a type system that aims at deciding the most favourable **C** type (`int`, `long`, `unsigned int`, ...) that can soundly (i.e., without any risk of overflow) implement an integer operation [Jakobsson et al. 2015; Kosmatov et al. 2020]. This is undecidable in the general case, and what the system computes is only a safe approximation of the optimal answer. Several previous works deal with the analysis and code generation used in **E-ACSL**. First of all, [Jakobsson et al. 2015] and [Kosmatov et al. 2020] proposed a type system to soundly infer a type over which basic **ACSL** assertions can be computed. [Kosmatov et al. 2020] benchmarked this technique on practical cases and have shown that this method significantly improves the performances of the generated code compared with an implementation where all the operations are done using unbounded arithmetic.

The outline of the paper is as follow. Section 2 introduces basic notions of Programming Language syntax and semantics, presents the extended quantifiers, and formalizes the semantics of subsets of **C**, **ACSL** and of operations of the **GMP** library. Then, Section 3 presents our contribution. Firstly, a type system which infers for each extended quantifier a type which can contains its result, the properties that this type system must respect and a schematic proof of why the type system respects these properties. Secondly, the code generation process and the properties that it must respect and a schematic proof of why it respects them. Finally, this section presents the implementation of the type system and of the code generation process inside **Frama-C**. Section 4 presents conclusions and ideas for further works.

2 SYNTAX AND SEMANTICS OF THE UNDERLYING LANGUAGES

Our work strongly relies on syntactic and semantic tools for representing programming languages. In programming language theory, semantics is the art of studiyng the meaning of syntactically valid programs [Hennessy 1990] using mathematical tools and methods for modeling the behavior of a program. In the following, we detail the formalization of the syntax and semantics of a fragment of **C**, which we call **mini-C**, a **mini-ACSL** language which consists in the core of the **ACSL** language, and a **mini-GMP** language which contains the elements of the **GMP** library that we use.

2.1 Syntax of mini-C, mini-ACSL, and mini-GMP

Fig. 2 presents the syntax of **mini-C**, **mini-ACSL** and **mini-GMP**. For **mini-C** and **mini-GMP**, we have defined two layers of constructions: the expressions and the statements. Informally, expressions compute a value and statements affect the memory. Keeping this distinction is useful for the simplicity of our work, even if in plain **C** the distinction between expressions and statements is not so clear cut. For instance, `i++` and `x=a+b` are expressions that behave as statements. Nevertheless, it does not involve any loss of generality.

The **mini-ACSL** language mainly consists in first order logic assertions, while being expressive enough to define the syntactic and semantic context in which `\sum`, `\product`, and `\numof` occur.

In the grammars of **mini-C** and **mini-ACSL** we have factored rules by using a generic $\diamond$ to represent all the arithmetic operators, and a generic $\triangleright$ to represent all the comparison operators.

It is worth noting the difference between *int variable*, *mpz variable* and *logical variable*. *int variables* are variables declared as `int` in a **C** program, so they contain bounded integers. *mpz variables* can be seen as variables of an abstract C-type which represents unbounded integers. *int variable* and *mpz variable* are stored in the program memory space and have a scope limited to the block where they are declared. Finally, *logical variable* are only used inside extended quantifiers. They are essentially lambda-variables that can be bound to unbounded integers. They are stored in disjoint memory space and have a scope restrained to the lambda-term where they are defined.

⟨*int variable*⟩ ::= ‘`x`’ (‘`x`’$\in \mathcal{L}$(`[a-z]`$^+$`[0-9]`*))

⟨*expression*⟩ ::= ‘`n`’ (‘`n`’$\in \mathcal{L}$(`0`|(`−`$^?$)`[1-9][0-9]`*))
| ⟨*int variable*⟩
| ⟨*expression*⟩ $\diamond$ ⟨*expression*⟩ ($\diamond \in$ {‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’})
| ⟨*expression*⟩ $\triangleright$ ⟨*expression*⟩ ($\triangleright \in$ {‘`<`’, ‘`<=`’, ‘`>`’, ‘`>=`’; ‘`==`’; ‘`!=`’})

⟨*declaration*⟩ ::= ‘`int`’ ⟨*int variable*⟩ ‘`;`’ (variable declaration)

⟨*statement*⟩ ::= ⟨*variable*⟩ ‘`=`’ ⟨*expression*⟩ (assignment)
| ⟨*statement*⟩ ‘`;`’ ⟨*statement*⟩ (sequence)
| ‘`{`’ ⟨*declaration*⟩* ⟨*statement*⟩ ‘`}`’ (code block)
| ‘`while(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement*⟩ (loop)
| ‘`if(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement*⟩ ⟨*statement*⟩ (condition)
| ‘`assert(`’ ⟨*expression*⟩ ‘`)`’

(mini-C)

⟨*predicate*⟩ ::= ‘`\true`’ (predicates for assertions)
| ‘`\false`’
| ‘`!`’ ⟨*predicate*⟩
| ⟨*predicate*⟩ ‘`||`’ ⟨*predicate*⟩ (logical or)
| ⟨*predicate*⟩ ‘`&&`’ ⟨*predicate*⟩ (logical and)
| ⟨*predicate*⟩ ‘`==>`’ ⟨*predicate*⟩ (logical implication)
| ⟨*predicate*⟩ ‘`<==>`’ ⟨*predicate*⟩ (logical equivalence)
| ⟨*term*⟩ $\triangleright$ ⟨*term*⟩ ($\triangleright \in$ {‘`<`’, ‘`<=`’, ‘`>`’, ‘`>=`’, ‘`==`’, ‘`!=`’})

⟨*statement*⟩ ::= ‘`logical_assert(`’ ⟨*predicate*⟩ ‘`)`’

⟨*term*⟩ ::= ⟨*expression*⟩
| ⟨*logical variable*⟩ (‘`x`’$\in \mathcal{L}$(`[a-z]`$^+$`[0-9]`*))
| ⟨*term*⟩ $\diamond$ ⟨*term*⟩ ($\diamond \in$ {‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’})
| ⟨*predicate*⟩ ‘`?`’ ⟨*term*⟩ ‘`:`’ ⟨*term*⟩ (conditional term)
| ‘`\sum(`’ ⟨*term*⟩ ‘`,`’ ⟨*term*⟩ ‘`, \lambda`’ ⟨*logical variable*⟩ ‘`;`’ ⟨*term*⟩ ‘`)`’
| ‘`\product(`’ ⟨*term*⟩ ‘`,`’ ⟨*term*⟩ ‘`, \lambda`’ ⟨*logical variable*⟩ ‘`;`’ ⟨*term*⟩ ‘`)`’
| ‘`\numof(`’ ⟨*term*⟩ ‘`,`’ ⟨*term*⟩ ‘`, \lambda`’ ⟨*logical variable*⟩ ‘`;`’ ⟨*predicate*⟩ ‘`)`’

(mini-ACSL)

⟨*mpz variable*⟩ ::= ‘`x`’ (‘`x`’$\in \mathcal{L}$(`[a-z]`$^+$`.[0-9]`*))

⟨*expression*⟩ ::= ‘`mpz_cmp(`’⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz comparison)

⟨*declaration*⟩ ::= ‘`mpz`’ ⟨*mpz variable*⟩ ‘`;`’ (variable declaration)

⟨*statement*⟩ ::= ‘`mpz_init(`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz initialization)
| ‘`mpz_set_int(`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*expression*⟩ ‘`)`’ (mpz assignment)
| ‘`mpz_set_mpz(`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz affectation)
| ‘`mpz_clear(`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz dealloc)
| ‘`mpz_add(`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz addition)
| ‘`mpz_mul(`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`,`’ ⟨*mpz variable*⟩ ‘`)`’ (mpz multiplication)

(mini-GMP)

Fig. 2. Syntax

2.2 Extended quantifiers

Extended quantifiers behave like definite integrals; they contain a function with a bound variable that is evaluated from a first value to a second value. All intermediate results are aggregated according to the semantics of the extended quantifier: sum for `\sum`, product for `\product`, and counting for `\numof`. Two other extended quantifiers in the **ACSL** language are `\min` and `\max`, and all we have done in this work may be applies directly to these other extended quantifiers.

In their concrete syntax, extended quantifiers are noted as functions with three parameters. The first parameter, the **Min** term, is an expression that denotes the first value (`a` in `\sum(a, b, \lambda k;c*k)`). The second parameter, the **Max** term, is an expression that denotes the second value (`b`). The third parameter is a lambda-expression that denotes the function which is applied from **Min** to **Max** and whose results are summed ($k \mapsto c \times k$ in the example).

2.3 Semantics of mini-C, mini-ACSL, and mini-GMP

The semantics of the three languages are defined using an inference system that formalizes a *big-step operational semantics.*

An *inference system* is made of several inference rules that are noted using an horizontal bar that separates *hypotheses* (possibly none) from one *conclusion.* The hypotheses lay above the bar, and

$$\frac{}{\Gamma, x : \tau \vdash_{stl} x : \tau} \qquad \frac{\Gamma, x : \tau \vdash_{stl} e : \tau'}{\Gamma \vdash_{stl} \lambda x.e : \tau \longrightarrow \tau'} \qquad \frac{\Gamma, A \vdash_{seq} B}{\Gamma \vdash_{seq} A \Rightarrow B} \qquad \frac{A \qquad B}{A \wedge B}$$

Fig. 3. Four examples of inference rules from simply typed lambda-calculus (*stl*), intuitionistic sequent calculus (*seq*) and natural deduction.

the conclusion lays underneath. Fig. 3 shows four examples of inference rules. Beware that they do not form an inference system by themselves, but are only parts of well-known inference systems.

Conclusion and hypotheses are *judgements* that express properties, so that the meaning of an inference rule is "if all the hypotheses are proved, then the conclusion is proved". Very often, conditions that are not strictly formalized as judgements are added to the hypotheses. They act as guards or triggers. Inference rules that have no hypothesis (other than guards or triggers) are called *axioms*; they express unconditionally true judgements (e.g., the leftmost inference rule in Fig. 3).

An inference system composes inference rules to form *proofs*. Inference rules can be considered as tiles that can be assembled if their faces match. In the case of inference systems, one says that two rules match if the conclusion of one rule matches one hypothesis of the other rule. There is a finite set of inference rules, but a proof can use an unbounded number of copies (also called *instances*) of them. When a set of instances of inference rule is properly connected, it forms a *proof tree* where the root is the conclusion, and the leaves are hypotheses that have not yet been proved. When a proof tree has no leaves (one says it is *closed*) it is called a *proof*. Note that without axioms a proof tree could never be closed.

The structure and syntax of judgements vary greatly from an inference system to another. However, whichever the structure and syntax, judgement contains *meta-variables*. Meta-variables are local to an inference rule, and can be replaced by terms according to the structure of the judgement. For instance, the rightmost rule in Fig. 3 contains meta-variables A and B, and its conclusion only matches formulas that are conjunctions. All occurrences of the same meta-variable in the same instance of a rule must be replaced by the same term; they act as co-occurrence constraints.

Very often, parts of the judgements are terms that may contain *object variables* (e.g., program chunks containing program variables). In the two leftmost rules of Fig. 3, x is a meta-variable that captures an object variable. Even if object variables are also captured by meta-variables, they must absolutely be distinguished from them. Object variables belong to the language an inference system is formalizing, but meta-variables belong to the language of the formalization. Object variables must be considered as symbols, the scope and meaning of which are formalized by the inference system. Very often, a goal of an inference system is to assign something (memory location, value, type, etc.) to object variables, and more generally to symbols. This is often done using *contexts* that associate something to symbols. Tradition is to use capital greek letters (e.g., Γ or Λ) as meta-variable identifiers for contexts. Contexts may be structured as sets, lists, trees, etc., according to the scoping rules that are formalized. In Fig. 3, the context of the two leftmost rules is a list (or a stack), in order to cope for the scoping of lambda-variables, and the context of the third rule can be simply a set since neither order nor multiplicity matters. Note that for the two leftmost rule, notation $\Gamma, x : \tau$ must be interpreted slightly differently in hypotheses and in conclusion. In hypotheses, the $x : \tau$ is inserted as on the top of a stack, whereas in the conclusion it is sought for in the stack from top to bottom. This is part of the rule of the game that every inference system must precisely define.

Such inference systems are of standard use for formalizing logics, typing, semantics of programming languages, etc. See for instance [Jones 1997; Lalement 1990; Reynolds 2009], they are very versatile, and we have used them thoroughly in the formal part of this work. In many cases,

$$\frac{\mathit{precondition}}{\mathit{Context} \models \texttt{code} \Rightarrow \mathit{postcondition}} \qquad \frac{\mathit{Memory} \models \texttt{condition} \Rightarrow \mathit{false}}{\mathit{Memory} \models \texttt{while(condition)\{code\}} \Rightarrow \mathit{Memory}}$$

Fig. 4. Principle of big-step operational semantics.

inference rules can be easily encoded in a programming language provided it is high-level enough (recursivity, pattern-matching, etc., like in OCaml or Scala).

yutgfcgrthyjuygtrf

So, we have formalized the semantics of **mini-C**, **mini-ACSL**, and **mini-GMP** as inference systems. This does not tell much of the semantic style, because inference systems are very flexible. According to previous works like [Blazy and Leroy 2009] or [Ly et al. 2020], we have chosen the *big-step operational semantics* style. This is a class of inference rule systems which allow to deduce a post-condition from pre-conditions, a context and a chunk of code. Fig. 4 shows two schematic rules. The first one illustrates the principle of this class of inference rules. It expresses that if *precondition* is true then the execution of `code` in some *Context* implies *postcondition*. The second rule expresses that if in context *Memory* (that associates variables to values) expression *condition* returns false then executing `while(condition){code}` in the same context does not affect memory.

In order to express code generation and prove it, one needs to formalize the semantics of the source and target languages of the translation. **Source** is **mini-C** extended with **mini-ACSL** while **Target** is **mini-C** extended with **mini-GMP**. It is worth noting that **Source** and **Target** share many constructs, which have the same semantics in both languages. For the sake of simplicity, we express the semantics of shared constructs only once. These languages are mainly composed of statements and expressions, plus predicates and terms for the **mini-ACSL** part of **Source**.

It is important to note that it is a *blocking semantics*, that is to say, if no rules can be applied, then nothing can be deduced. It is useful for proving the code generation correctness. Also, for the sake of simplicity, we suppose that all the programs are well-formed. For example, no program uses an undeclared variable. This is checked by the **Frama-C** kernel before running **E-ACSL**.

2.3.1 Memory and Environment semantics. An environment is a function which maps a variable to a memory block. It is formalized as a set of tuples (v, b) (variable, memory block). One notes $E(v) = b$ if $(v, b) \in E$. A memory is a function which maps a memory block to a value. It is formalized as a set of tuples (b, val) (memory block, value). One notes $M(b) = val$ if $(b, val) \in M$. If b is a memory block and val a value then $M[b \longrightarrow val] = M \backslash \{(b, M(b))\} \cup \{(b, val)\}$. These two functions can be composed to form a function $(M \circ E)$ which maps variables to values. We note $C = (E, M)$.

Since a program using a variable before its declaration is not well-formed and according to the semantics that we define hereafter, $E(v)$ and $M(b)$ are always well defined.

Values are noted $int(v)$ or $mpz(v)$. It denotes the format under which v is read or written in memory. Since we assume that programs are well-typed, $\mathcal{T}$ is an oracle that returns the type of a declared variable. The two notations are often combined. E.g., $\mathcal{T}(x)(v)$ denotes a value v represented under the format of the type of variable `x`.

2.3.2 Semantics of predicates and terms. Predicates and terms belong to the **mini-ACSL** part of **Source**. Semantic rules for predicates (Fig. 5) and terms (Fig. 6) are defined in a semantic context (E, M), extended with a logical context Λ which maps logical variables to mathematical numbers. To distinguish between syntactic operators (e.g., + in a **C** program) and semantic operator (e.g., + on integers), which have almost the same typography, we decorate the semantic operators with a dot; e.g., syntactic + denotes the $\dot{+}$ semantic operation. For semantic operators with no risk of confusion (ie. $\dot{<}$, $\dot{<=}$, $\dot{>}$, $\dot{>=}$, $\dot{==}$ and $\dot{!=}$), we use their usual typography ($<$, $\leq$, $>$, $\geq$, $=$, and $\neq$). Note that because variables in Λ are the variables that are bound in surrounding extended

$$\frac{}{E,M,\Lambda \models_p \texttt{\textbackslash true} \Rightarrow True} \qquad \frac{}{E,M,\Lambda \models_p \texttt{\textbackslash false} \Rightarrow False} \qquad \frac{E,M,\Lambda \models_p p \Rightarrow False}{E,M,\Lambda \models_p\ !p \Rightarrow True} \qquad \frac{E,M,\Lambda \models_p p \Rightarrow True}{E,M,\Lambda \models_p\ !p \Rightarrow False}$$

$$\frac{E,M,\Lambda \models_p p1 \Rightarrow True \qquad E,M,\Lambda \models_p p2 \Rightarrow True}{E,M,\Lambda \models_p p1\ \texttt{||}\ p2 \Rightarrow True} \qquad \frac{E,M,\Lambda \models_p p1 \Rightarrow False \qquad E,M,\Lambda \models_p p2 \Rightarrow True}{E,M,\Lambda \models_p p1\ \texttt{||}\ p2 \Rightarrow True}$$

$$\frac{E,M,\Lambda \models_p p1 \Rightarrow True \qquad E,M,\Lambda \models_p p2 \Rightarrow False}{E,M,\Lambda \models_p p1\ \texttt{||}\ p2 \Rightarrow True} \qquad \frac{E,M,\Lambda \models_p p1 \Rightarrow False \qquad E,M,\Lambda \models_p p2 \Rightarrow False}{E,M,\Lambda \models_p p1\ \texttt{||}\ p2 \Rightarrow False}$$

$$\frac{E,M,\Lambda \models_t t1 \Rightarrow v \qquad E,M,\Lambda \models_t \texttt{t2} \Rightarrow w \qquad v \mathrel{\dot{\triangleright}} w}{E,M,\Lambda \models_p t1 \triangleright \texttt{t2} \Rightarrow 1} \qquad \frac{E,M,\Lambda \models_t t1 \Rightarrow v \qquad E,M,\Lambda \models_t \texttt{t2} \Rightarrow w \qquad \neg(v \mathrel{\dot{\triangleright}} w)}{E,M,\Lambda \models_p t1 \triangleright \texttt{t2} \Rightarrow False}$$

Fig. 5. Semantics of predicates.

$$\frac{E,M,\Lambda \models_e e \Rightarrow int(v)}{E,M,\Lambda \models_t e \Rightarrow v} \qquad \frac{}{E,M,\Lambda \models_t \texttt{x} \Rightarrow \Lambda(\texttt{x})} \qquad \frac{E,M,\Lambda \models_t \texttt{t1} \Rightarrow v \qquad E,M,\Lambda \models_t \texttt{t2} \Rightarrow w}{E,M,\Lambda \models_t \texttt{t1} \diamond \texttt{t2} \Rightarrow v \mathrel{\dot{\diamond}} w}$$

$$\frac{E,M,\Lambda \models_p \texttt{condition} \Rightarrow True \qquad E,M,\Lambda \models_t \texttt{t1} \Rightarrow v1}{E,M,\Lambda \models_t \texttt{condition ? t1: t2} \Rightarrow v1} \qquad \frac{E,M,\Lambda \models_p \texttt{condition} \Rightarrow False \qquad E,M,\Lambda \models_t \texttt{t2} \Rightarrow v2}{E,M,\Lambda \models_t \texttt{condition ? t1: t2} \Rightarrow v2}$$

$$\frac{E,M,\Lambda \models_t \texttt{t1} \Rightarrow min \qquad E,M,\Lambda \models_t \texttt{t2} \Rightarrow max \qquad \forall k \in [min,max](E,M,\Lambda[\texttt{x} \to k] \models_t \texttt{t3} \Rightarrow lambda_k)}{E,M,\Lambda \models_t \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow \sum_{k=min}^{max} lambda_k}$$

Fig. 6. (Fragments of) Semantics of terms (totality in Appendix A).

$$\frac{min\_int \le valof(\texttt{n}) \le max\_int}{E,M \models_e \texttt{n} \Rightarrow int(valof(\texttt{n}))} \qquad \frac{\mathcal{T}(\texttt{x}) = \texttt{int} \qquad (M \circ E)(\texttt{x}) = int(v)}{E,M \models_e \texttt{x} \Rightarrow int(v)}$$

$$\frac{E,M \models_e \texttt{e1} \Rightarrow int(v_1) \qquad E,M \models_e \texttt{e2} \Rightarrow int(v_2) \qquad min\_int \le v_1 \mathrel{\dot{\diamond}} v_2 \le max\_int}{E,M \models_e \texttt{e1} \diamond \texttt{e2} \Rightarrow int(v_1 \mathrel{\dot{\diamond}} v_2)}$$

$$\frac{E,M \models_e \texttt{e1} \Rightarrow int(v_1) \qquad E,M \models_e \texttt{e2} \Rightarrow int(v_2) \qquad v_1 \mathrel{\dot{\triangleright}} v_2 \qquad v \ne 0 \qquad min\_int \le v \le max\_int}{E,M \models_e \texttt{e1} \triangleright \texttt{e2} \Rightarrow int(v)}$$

$$\frac{E,M \models_e \texttt{e1} \Rightarrow int(v_1) \qquad E,M \models_e \texttt{e2} \Rightarrow int(v_2) \qquad \neg(v_1 \mathrel{\dot{\triangleright}} v_2)}{E,M \models_e \texttt{e1} \triangleright \texttt{e2} \Rightarrow int(0)}$$

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad (M \circ E)(\texttt{x}) = mpz(v_1) \qquad (M \circ E)(\texttt{y}) = mpz(v_2) \qquad v_1 \mathrel{\dot{=}} v_2}{E,M \models_s \texttt{mpz\_cmp(x,y)} \Rightarrow int(0)}$$

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad (M \circ E)(\texttt{x}) = mpz(v_1) \qquad (M \circ E)(\texttt{y}) = mpz(v_2) \qquad v_1 \mathrel{\dot{>}} v_2 \qquad 0 < v \le max\_int}{E,M \models_s \texttt{mpz\_cmp(x,y)} \Rightarrow int(v)}$$

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad (M \circ E)(\texttt{x}) = mpz(v_1) \qquad (M \circ E)(\texttt{y}) = mpz(v_2) \qquad v_1 \mathrel{\dot{<}} v_2 \qquad min\_int \le v < 0}{E,M \models_s \texttt{mpz\_cmp(x,y)} \Rightarrow int(v)}$$

Fig. 7. Semantics of expressions.

quantifiers, Λ only changes in `\sum`, `\product` and `\numof` (see Fig. 6). Note also that if $max < min$, then $\sum_{k=min}^{max} lambda_k = 0$ and $\prod_{k=min}^{max} lambda_k = 1$. One says that the neutral element for `\sum` and `\product` is respectively 0 and 1.

2.3.3 Semantics of expressions. Fig. 7 presents the semantics rules for expressions. We define the semantics of expressions using a judgement noted $\models_e$. It allows to deduce the value of an expression in a semantic context $C = (E, M)$. These rules are usual, except the rules that model integer overflow. For doing so, `min_int` and `max_int` respectively denotes the smallest and the largest integer which fits in an `int`. As for predicates and terms, to prevent confusion between operators on expressions and on integers, we note the operator on integers with a dot. We also introduce a function $valof$ which maps the characters sequence representing an integer to the integer itself. As an example $valof(42)$ is equal to 42.

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{int} \qquad E_1, M_1 \models_e \texttt{e} \Rightarrow \mathit{int}(v) \qquad M_2 = M_1[E_1(\texttt{x}) \rightarrow \mathit{int}(v)]}{E_1, M_1 \models_s \texttt{x=e} \Rightarrow M_2}$$

$$\frac{E_1, M_1, \emptyset \models_p \texttt{p} \Rightarrow \mathit{True}}{E_1, M_1 \models_s \texttt{logical\_assert(p)} \Rightarrow M_1} \qquad \frac{E_1, M_1 \models_e \texttt{condition} \Rightarrow v \qquad v \neq 0}{E_1, M_1 \models_s \texttt{assert(condition)} \Rightarrow M_1} \qquad \frac{E_1, M_1 \models_s \texttt{s1} \Rightarrow M_2 \qquad E_1, M_2 \models_s \texttt{s2} \Rightarrow M_3}{E_1, M_1 \models_s \texttt{s1; s2} \Rightarrow M_3}$$

$$\frac{E_2, M_2 = \mathit{alloc\_vars}(\texttt{d}^*, E_1, M_1) \qquad E_2, M_2 \models_s \texttt{s} \Rightarrow M_3 \qquad M_4 = \mathit{dealloc\_vars}(\texttt{d}^*, E_2, M_3)}{E_1, M_1 \models_s \texttt{\{d}^*\texttt{ s\}} \Rightarrow M_4}$$

$$\frac{E_1, M_1 \models_e \texttt{e} \Rightarrow \mathit{int}(0)}{E_1, M_1 \models_s \texttt{while(e)s} \Rightarrow M_1} \qquad \frac{E_1, M_1 \models_e \texttt{e} \Rightarrow \mathit{int}(v) \qquad n \neq 0 \qquad E_1, M_1 \models_s \texttt{s} \Rightarrow M_2 \qquad E_1, M_2 \models_s \texttt{while(e)s} \Rightarrow M_3}{E_1, M_1 \models_s \texttt{while(e)s} \Rightarrow M_3}$$

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{mpz} \qquad E_1, M_1 \models_e \texttt{e} \Rightarrow \mathit{int}(v) \qquad M_2 = M_1[E_1(\texttt{x}) \rightarrow \mathit{mpz}(v)]}{E_1, M_1 \models_s \texttt{mpz\_set\_int(x,e)} \Rightarrow M_2}$$

$$\frac{\mathcal{T}(\texttt{x}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad (M \circ E)(\texttt{y}) = \mathit{mpz}(v) \qquad M_2 = M_1[E_1(\texttt{x}) \rightarrow \mathit{mpz}(v)]}{E_1, M_1 \models_s \texttt{mpz\_set\_mpz(x,y)} \Rightarrow M_2}$$

$$\frac{\begin{array}{c}\mathcal{T}(\texttt{x}) = \texttt{mpz} \\ \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{z}) = \texttt{mpz} \qquad (M_1 \circ E_1)(\texttt{y}) = \mathit{mpz}(v_1) \qquad (M_1 \circ E_1)(\texttt{z}) = \mathit{mpz}(v_2) \qquad M_2 = M_1[E_1(\texttt{x}) \rightarrow \mathit{mpz}(v_1 \dot{+} v_2)]\end{array}}{E_1, M_1 \models_s \texttt{mpz\_add(x,y,z)} \Rightarrow M_2}$$

$$\frac{\begin{array}{c}\mathcal{T}(\texttt{x}) = \texttt{mpz} \\ \mathcal{T}(\texttt{y}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{z}) = \texttt{mpz} \qquad (M_1 \circ E_1)(\texttt{y}) = \mathit{mpz}(v_2) \qquad (M_1 \circ E_1)(\texttt{z}) = \mathit{mpz}(v_2) \qquad M_2 = M_1[E_1(\texttt{x}) \rightarrow \mathit{mpz}(v_1 \dot{*} v_2)]\end{array}}{E_1, M_1 \models_s \texttt{mpz\_mul(x,y,z)} \Rightarrow M_2}$$

Fig. 8. Semantics of statements.

2.3.4 *Tool functions.* Defining the semantics of statements requires to introduce the following tool functions for accessing and modifying environments. Let D be a set of pairs $(\mathit{type}, \mathit{variable})$ to add or subtract to the environment:

- Function $\mathit{new\_block} : t, M \mapsto b, M'$ such that b is a block that is not already used in M and $M' = M \cup \{(b, t(0))\}$, where t is some type. We assume an unbounded number of blocks.
- Function $\mathit{alloc\_vars} : D, E, M \mapsto E', M'$ such that
  - $\mathit{alloc\_vars}(\emptyset, E, M) = E, M$ and
  - $\mathit{alloc\_vars}((t, dv) \uplus D', E, M) = \mathit{alloc\_vars}(D', E', M')$
    such that $E' = \{(v, b) \mid (v, b) \in E \wedge v \neq dv\} \cup \{(dv, db)\}$, and $(db, M') = \mathit{new\_block}(t, M)$.

  This function returns the semantic context (E', M') that results from the declarations of all the pairs $(\mathit{type}, \mathit{variable})$ in D in the semantic context (E, M).
- Function $\mathit{dealloc\_vars} : D, E, M \mapsto M - \{(b, M(b)) \mid v \in \mathit{codom}(D) \wedge E(v) = b\}$
  returns a copy of M where all the blocks corresponding to variables in D have been removed.
- Function unwrap returns the integer contained in a value, so $\mathit{unwrap}(\mathit{mpz}(n)) = n$ and $\mathit{unwrap}(\mathit{int}(m)) = m$.

2.3.5 *Semantics of statements.* Fig. 8 shows the semantics of statements using a judgement noted $\models_s$. It allows us to deduce the memory state M_2 after the execution of a statement in an environment E_1 and a memory M_1.

For the sake of simplicity, we do not define the semantic rules for `mpz_init` and `mpz_clear`. We just ask that a well formed program never uses `mpz` variables before having initialized them and all the `mpz` variables of a block are cleared before the block ends. So the execution of a program that does not respect one of these two rules is blocked.

2.3.6 *A functional reading of the semantics.* An expected property of these four inference systems is that from a context and a construction of one of the languages, it can only be inferred one thing. A semantics which satisfies this property is said to be *deterministic*; it is a sought-after property of a semantics since it allows to see the semantics as a function.

However, because the integers that represent truth values are under-specified, the semantics is only nearly deterministic. Indeed, in the semantics of expressions given Fig. 7, the result of comparisons is a value $\mathit{int}(v)$ where v is not strictly specified; $v = 0$ encodes False and $v \neq 0$

encodes $True$. Otherwise, the semantics would be deterministic because all rule preconditions are mutually exclusive, which ensures that only one rules can be applied at each node of a proof tree.

We can enforce determinism by either specifying the v that encodes truth values (say, $v = 1$), or by considering that programs whose semantics depends on the choice of v are unsafe.

3 CONTRIBUTIONS

As we have said previously, our goal is to generate from a program in the **Source** language, a program with the same operational semantics in the **Target** language. Our contribution is divided in four parts:

- an inference system which allows to deduce from a context and an extended quantifier a type in which its result soundly fits. We express the soundness property of this system, and prove it for the extended quantifiers.
- the formalization of the code generation for the extended quantifiers.
- the formalization and proof of the soudness property of the code generation.
- the implementation of the analysis-generation process in the **E-ACSL** plug-in of **Frama-C**.

3.1 Term typing judgment

For preparing code generation, we use a judgment noted $\models_{interv}$. It allows to deduce, before the execution and according to the program abstract syntax tree (AST), on which type should be computed a term in an **ACSL** assertion in the generated program. Several rules of this judgment have been specified in [Jakobsson et al. 2015] and [Kosmatov et al. 2020]. Our contribution is to propose new rules for extended quantifiers. For the sake of simplicity, we suppose that there are only two types, `int` and `mpz`, even if we know that in reality there exists other types. What is important is to decide between using a non bounded type like **mini-GMP**'s `mpz` and a bounded type like `int`.

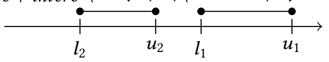
The main goal is to infer the type `int` (of bounded integers) as frequently as possible, since operations on `int` are significantly faster than operations on `mpz`. Another one is to propose rules which are efficient enough to keep the inference system efficient on large projects. To infer a type for each term, we first present a judgment which deduces intervals from terms and then a judgment which deduces types from intervals.

Firstly, we propose an interval inference system which deduce from every extended quantifier a *"not too big"* interval such as its operational semantics is inside. (The notion of *"not too big"* may seem imprecise, and so it is, because in fact, it is an optimization problem solved by heuristics and we cannot pretend or even less prove that the inferred intervals are as small as possible. In practice, it is a best-effort process to infer intervals that are as small as possible.)

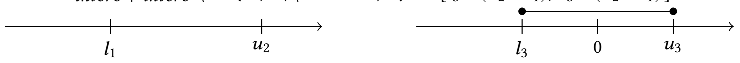
Note that there are intervals of two kinds. First, the **Min**-**Max** intervals bound the extended quantifiers. They are entirely specified by the programmer using terms. Second, the intervals we compute approximate the range of variables and terms. The second kind is an abstraction of the values manipulated by the program [Rival and Yi 2020]. The two kinds meet when **Min** and **Max** are abstracted. Recall that nothing forces **Min** to be lower than **Max** (see remark at the end of 2.3.2), but note also that even if that would be true, it would not prevent their abstractions to overlap.

A fragment of the inference rules of this system for `\sum` is shown Fig. 9. The complete system is given in Appendix B. All the complexity consists in dealing with the different overlapping situations that may happen between the upper and lower bounds of the intervals that contains the **Min** and **Max** terms. These situation are illustrated with diagrams that are inserted after each deduction rule. The first rule in Fig. 9 deals with the case where the **Min** interval is strictly to the right of the **Max** interval in the integer line: the result is $[0, 0]$. The second rule deals with the case where the

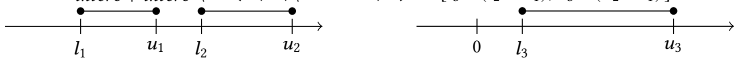
$$\frac{\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \qquad \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad l_1 > u_2}{\Gamma_{interv} \models_{interv} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow [0, 0]}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \leq u_2 \qquad l_3 \leq 0 \leq u_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow [l_3 * (u_2 - l_1), u_3 * (u_2 - l_1)]}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \leq u_2 \qquad u_1 \leq l_2 \qquad 0 < l_3 \leq u_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow [l_3 * (l_2 - u_1), u_3 * (u_2 - l_1)]}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \leq u_2 \qquad u_1 > l_2 \qquad 0 < l_3 \leq u_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow [0, u_3 * (u_2 - l_1)]}$$

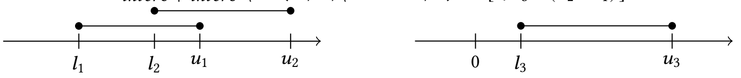


Fig. 9. (Fragment of) Interval inference rules for `\sum(t1, t2, \lambda x;t3)` (totality in Appendix B).

Min and **Max** intervals overlap and the codomain of the lambda-function contains 0; the result must be between the least possible value times the max number of steps and the greatest possible value times the max number of steps. The third rule deals with the case where all possible values are positive; the result must be between the least possible value times the min number of steps and the greatest possible value times the max number of steps. Finally, the fourth rule is a variant of the third one when the **Min** and **Max** intervals overlap; the min value of the result can then be 0. All other rules deal with symmetrical cases. A more formal proof is shown in Appendix C.

The inference rules maintain as an invariant that the lower bound of an interval is always inferior to the upper bound. This invariant is obviously maintained by the rules presented here.

The rules presented in this report are only concerned with extended quantifiers. We assume that there are correct rules for the other terms of the language, some of them being described in [Jakobsson et al. 2015] and [Kosmatov et al. 2020]. We also assume that Γ_{interv} is a function which maps a logical variable of **Source** to the interval of values it can takes. It is called the *interval context.* In summary, we have formalized the interval abstraction of extended quantifiers assuming that the contexts in which they occur are properly instrumented.

It remains to formalize the fact that Γ_{interv} is an abstraction of Λ, and to prove it.

Definition 1 (Consistency of Γ_{interv} with Λ). *An interval context Γ_{interv} is consistent with a logical context Λ, if $dom(\Lambda) \subseteq dom(\Gamma_{interv})$ and for all $x \in dom(\Lambda)$, $\Lambda(x) \in \Gamma_{interv}(x)$. If so, Γ_{interv} is an abstraction of Λ.*

Theorem 1 (Interval inference system is correct). *The inference system denoted by $\models_{interv}$ is correct for all term t. That is to say, in an interval context Γ_{interv} consistent with a logical context Λ, and a semantic context $\widehat{C} = (\widehat{E}, \widehat{M})$ where all the variables in t have been declared, if $\widehat{E}, \widehat{M}, \Lambda \models_t t \Rightarrow val$ and $\Gamma_{interv} \models_{interv} t \Rightarrow interv$ then $val \in interv$.*

In the same way as for $\models_{interv}$, a judgement noted $\models_{type}$ infers the type of a term from an interval context Γ_{interv}. Fig. 10 shows the rules for `\sum`. In this judgment we use a function Θ which maps an interval i of $\mathbb{N}$ to the smallest `C` type which contains all the elements of i considering that

$$\frac{\Gamma_{interv} \models_{interv} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow [l_{sum}, u_{sum}]}{\Gamma_{interv} \models_{type} \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow \Theta([l_{sum}, u_{sum}])}$$

Fig. 10. Type inference rules for `\sum(t1,t2,\lambda x;t3)`.

`mpz` contains $[-\infty, +\infty]$. The proof of the correctness of this judgment is straightforward using Theorem 1.

Theorem 2 (Type inference is correct). *The inference system denoted by $\models_{type}$ is correct for all term t. That is to say, in a logical context Λ consistent with an interval context Γ_{interv} and a semantic context $\widehat{C} = (\widehat{E}, \widehat{M})$ where all the variables in t have been declared, the system always infers a type which can contains all the possible values of t in $\widehat{C}$ and Λ.*

3.2 Code generation

Code generation for terms can be seen as a function $[\![\cdot,\cdot]\!]_{tr}$, which maps a **Source** term t and a code generation context Ctx to a **Target** program of same semantics. Ctx is a function which maps a logical variable in t to the variable in the generated program which contains its runtime value.

$[\![\cdot,\cdot]\!]_{tr}$ returns three elements: *decl* which contains a list of program variables declarations, *code* which contains the list of statements of the generated program, and *res* which is the variable that contains the runtime value of the translated term at the end of the program execution. Furthermore, all the variables used in *code* are declared in *decl*. A function *type*, which relies on the type system presented before, maps a term to a type under which the term can be stored without any risk of overflow.

We define two functions, *init_var* which maps *decl* to the set of *mpz_init* required to have a well-formed code, and *clear_var* which maps *decl* to the set of *mpz_clear* required to have a well-formed code (see final remark in section 2.3.5).

3.2.1 Macros. When it comes to formalize and prove the code generation, an important issue is that we face too many cases. For example, translating `\product(`t_1`, `t_2`, \lambda x;`t_3`)` in the code generation context Ctx can result in an `int` or an `mpz` and so it is for all terms t_1, t_2 and t_3. This forms $2 * 2 * 2 * 2 = 16$ type combinations. A brute force approach to formalizing code generation and its proof would be to propose 16 formalizations and 16 proofs (one per type combination). This would be an inelegant waste of time.

To tackle that issue, we define *macros* which generate code depending of the type of their arguments. The *Macros* can be seen as functions that map several ASTs to another. It is worth noting that we do not need to specify a returned AST for each theoretical type combination. This is because in practice, not all cases are used to express the code generation process and not all cases can be expressed in one statement using **mini-C** or **mini-GMP**. For instance, no statement allows to assign an `mpz` to an `int`. Furthermore, this statement would not be useful in practice since it could lead to an overflow. So we have ignored the useless or undefinable statements.

For the sake of readability, we have chosen to represent ASTs by a possible concrete representation. Fig. 11 shows the specifications of the *macros.*

So doing, we can express the semantic rules of the *macros* in a unified type dependent way. That is to say we unify the semantic rules of each case of one macro in a single rule dependent of the type of the variable given as parameter (parameter `var`). We give these semantics as lemmas and prove their correctness (Appendix D).

Lemma 1 (Semantics of *int_assignment*). *int_assignment assigns to* `var` *an integer which fits in an* `int`.

```
int_assignment(var, n) :
  match T(var) with :
    case int :
      var=n
    case mpz :
      mpz_set_int(var ,n)
condition(var1, var2) :
  match T(var1) with :
    case int :
      var1<=var2
    case mpz :
      mpz_cmp(var1 , var2)<=0
```

```
addition_assignment(var1, var2) :
  match T(var1) with :
    case int :
      var1=var1+var2
    case mpz :
      mpz_add(var1 , var1 , var2)

multiplication_assignment(var1, var2) :
  match T(var1) with :
    case int :
      var1=var1*var2
    case mpz :
      mpz_mul(var1 , var1 , var2)
```

```
var_assignment(var1, var2) :
  match T(var1),T(var2) with :
    case int,int :
      var1=var2
    case mpz , int :
      mpz_set_int(var1 , var2)
    case mpz , mpz :
      mpz_set_mpz(var1 , var2)
```

Fig. 11. *Macros* specification.

$$\frac{min\_int \mathrel{\dot{\le}} valof(\mathsf{n}) \mathrel{\dot{\le}} max\_int \qquad M_2 = M_1[E(\mathsf{var}) \to \mathcal{T}(\mathsf{var})(valof(\mathsf{n}))]}{E, M_1 \models_s int\_assignment(\mathsf{var}, \mathsf{n}) \Rightarrow M_2}$$

Lemma 2 (Semantics of *var_assignment*). *var_assignment assigns to* var1 *the value of* var2*, the type of* var1 *must be able to contain all the possible values of* var2*. For instance, if* var1 *is an* int *then* var2 *cannot be an* mpz*.*

$$\frac{(M_1 \circ E)(\mathsf{var2})) = \mathcal{T}(\mathsf{var2})(v) \qquad M_2 = M_1[E(\mathsf{var1}) \to \mathcal{T}(\mathsf{var1})(v)]}{E, M_1 \models_m var\_assignment(\mathsf{var1}, \mathsf{var2}) \Rightarrow M_2}$$

Lemma 3 (Semantics of *addition_assignment*). *addition_assignment assigns to* var1 *its value plus the value of* var2*, the types of* var1 *and* var2 *must be equal and the result must fits into this type.*

$$\frac{\begin{array}{c}\mathcal{T}(\mathsf{var2}) = \mathcal{T}(\mathsf{var1}) \qquad (M_1 \circ E)(\mathsf{var1}) = \mathcal{T}(\mathsf{var1})(v_1) \qquad (M_1 \circ E)(\mathsf{var2}) = \mathcal{T}(\mathsf{var1})(v_2) \\ min\_type(\mathcal{T}(\mathsf{var1})) \mathrel{\dot{\le}} v_1 \mathbin{\dot{+}} v_2 \mathrel{\dot{\le}} max\_type(\mathcal{T}(\mathsf{var1})) \qquad M_2 = M_1[E(\mathsf{var1}) \to \mathcal{T}(\mathsf{var1})(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_m addition\_assignment(\mathsf{var1}, \mathsf{var2}) \Rightarrow M_2}$$

*where min_type and max_type are functions which return respectively the minimal and the maximal integer which fits in a type (eg. min_type(*int*) = min_int and max_type(*mpz*) = max_mpz, furthermore min_mpz =* $-\infty$ *and max_mpz =* $+\infty$*).*

Lemma 4 (Semantics of *multiplication_assignment*). *multiplication_assignment assigns to* var1 *its value times the value of* var2*, the types of* var1 *and* var2 *must be equal.*

$$\frac{\begin{array}{c}\mathcal{T}(\mathsf{var2}) = \mathcal{T}(\mathsf{var1}) \qquad (M_1 \circ E)(\mathsf{var1}) = \mathcal{T}(\mathsf{var1})(v_1) \qquad (M_1 \circ E)(\mathsf{var2}) = \mathcal{T}(\mathsf{var1})(v_2) \\ min\_type(\mathcal{T}(\mathsf{var1})) \mathrel{\dot{\le}} v_1 \mathbin{\dot{+}} v_2 \mathrel{\dot{\le}} max\_type(\mathcal{T}(\mathsf{var1})) \qquad M_2 = M_1[E(\mathsf{var1}) \to \mathcal{T}(\mathsf{var1})(v_1 \mathbin{\dot{*}} v_2)]\end{array}}{E, M_1 \models_m multiplication\_assignment(\mathsf{var1}, \mathsf{var2}) \Rightarrow M_2}$$

As before, min_type and max_type are functions which return respectively the minimal and the maximal integer which fits in a type.

Lemma 5 (Semantics of *condition*). *condition computes whether* var1 *is lesser or equal to* var2*, the types of* var1 *and* var2 *must be equal.*

$$\frac{\begin{array}{c}\mathcal{T}(\mathsf{var2}) = \mathcal{T}(\mathsf{var1}) \\ (M_1 \circ E)(\mathsf{var1}) = \mathcal{T}(\mathsf{var1})(v_1) \qquad (M_1 \circ E)(\mathsf{var2})) = \mathcal{T}(\mathsf{var1})(v_2) \qquad v_1 \mathrel{\dot{\le}} v_2 \qquad v \neq 0 \qquad min\_int \mathrel{\dot{\le}} v \mathrel{\dot{\le}} max\_int\end{array}}{E, M_1 \models_m condition(\mathsf{var1}, \mathsf{var2}) \Rightarrow int(v)}$$

$$\frac{\mathcal{T}(\mathsf{var2}) = \mathcal{T}(\mathsf{var1}) \qquad (M_1 \circ E)(\mathsf{var1}) = \mathcal{T}(\mathsf{var1})(v_1) \qquad (M_1 \circ E)(\mathsf{var2})) = \mathcal{T}(\mathsf{var1})(v_2) \qquad v_1 \mathrel{\dot{>}} v_2}{E, M_1 \models_m condition(\mathsf{var1}, \mathsf{var2}) \Rightarrow int(0)}$$

$\llbracket$`\sum(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.decl$ =
  t_k one ; t_k k ; t_k max ;
  t_sum sum ; t_sum lbda ;
  $\llbracket$t1, $Ctx\rrbracket_{tr}.decl$; $\llbracket$t2, $Ctx\rrbracket_{tr}.decl$

$\llbracket$`\sum(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.code$ =
  $\llbracket$t1, $Ctx\rrbracket_{tr}.code$; $\llbracket$t2, $Ctx\rrbracket_{tr}.code$;
  int_assignment(one, 1) ;
  var_assignment(k, $\llbracket$t1, $Ctx\rrbracket_{tr}.res$) ;
  var_assignment(max, $\llbracket$t2, $Ctx\rrbracket_{tr}.res$) ;
  int_assignment(sum, 0) ;
  while (*condition*(k,max)){
    $\llbracket$t3, Ctx[x $\rightarrow$k]$\rrbracket_{tr}.code$;
    var_assignment(lbda, $\llbracket$t3, Ctx[x $\rightarrow$k]$\rrbracket_{tr}.res$) ;
    addition_assignment(sum,lbda) ;
    addition_assignment(k,one) ;
  }

$\llbracket$`\sum(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.res$ = sum

$\llbracket$`\product(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.decl$ =
  t_k one ; t_k k ; t_k max ;
  t_product product ; t_product lbda ;
  $\llbracket$t1, $Ctx\rrbracket_{tr}.decl$; $\llbracket$t2, $Ctx\rrbracket_{tr}.decl$

$\llbracket$`\product(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.code$ =
  $\llbracket$t1, $Ctx\rrbracket_{tr}.code$; $\llbracket$t2, $Ctx\rrbracket_{tr}.code$;
  int_assignment(one, 1) ;
  var_assignment(k, $\llbracket$t1, $Ctx\rrbracket_{tr}.res$) ;
  var_assignment(max, $\llbracket$t2, $Ctx\rrbracket_{tr}.res$) ;
  int_assignment(product, 1) ;
  while (*condition*(k,max)){
    $\llbracket$t3, Ctx[x $\rightarrow$k]$\rrbracket_{tr}.code$;
    var_assignment(lbda, $\llbracket$t3, Ctx[x $\rightarrow$k]$\rrbracket_{tr}.res$) ;
    multiplication_assignment(product,lbda) ;
    addition_assignment(k,one) ;
  }

$\llbracket$`\product(t1,t2,\lambda x;t3)`, $Ctx\rrbracket_{tr}.res$ = product

Fig. 12. Formalization of code generation.

Translation from **Source** to **Target** is defined by pattern matching. The left side of Fig. 12 shows the translation for the construct `\sum(t1,t2,\lambda x;t3)`. It starts with the declaration of variables for controlling the loop; `one`, `k` and `max` are of type `t_k` which is the largest of the types inferred for `t1` and `t2+1`. Indeed, variable `k` must be large enough to contain all the possible values between those of `t1` and `t2+1`, since `k` will be at least equal to `t2` + 1 in the last execution of the `while` loop condition. Then variables `sum` and `lbda` for summing the values from `t1` to `t2` are declared. They have type `t_sum` inferred for this occurrence of the `\sum` pattern. To avoid inserting spurious casts in the generated code, the type of `one` and `max` is also `t_k`, while the type of `lbda` is `t_sum`, even if it is not strictly necessary. Because the $\llbracket\cdot,\cdot\rrbracket_{tr}.decl$ part can be inserted in a unique block, all variable names are prototype names; we assume that all variables are given new names when generated (e.g., `k` is renamed in `k_1`, `k_2`, etc.). The $\llbracket\cdot,\cdot\rrbracket_{tr}.code$ part starts with the code generated for `t1` and `t2`. Then it prepares the variables that control the loop, and it initializes the accumulator for the sum. Then, a while loop is generated. It contains the code for evaluating `t3` which depends on the current value of `k`. Last line declares that the final result $\llbracket\cdot,\cdot\rrbracket_{tr}.res$ is in `sum`. The right side of Fig. 12 shows the translation for pattern `\product(t1,t2,\lambda x;t3)`, which obey similar principles.

3.3 Proof of code generation

In order to prove that the code generation presented above is correct, we are interested in two properties: *soundness* and *transparency*. We first define these two properties, and then we prove that the code generation presented above is correct. When there is a possible confusion between the semantic context of a construct in **Source** and a context of the same construct in **Target** is possible, we note the semantic context of **Source** $\widehat{C} = (\widehat{E}, \widehat{M})$, despite the fact that $\widehat{C}$ and C are the same kind of objects.

Definition 2. *One says that a semantic context* $C_1 = (E_1, M_1)$ *is included in a semantic context* $C_2 = (E_2, M_2)$ *if* $(M_1 \circ E_1) \subseteq (M_2 \circ E_2)$. *We note that* $C_1 \subseteq C_2$.

$C_1 \subseteq C_2$ means that each variable of the domain of E_1 is associated to the same value in C_1 and in C_2.

Soundness is the property that ensures that the execution of the **Target** version of a term really computes the semantics of this term in the **Source** language.

Definition 3 (Code generation soundness). *Let t be a term, the code generation is sound if and only if, for all code generation context Ctx, **Source** semantic context $\widehat{C} = (\widehat{E}, \widehat{M_1})$, logical context Λ and **Target** semantic context $C = (E, M_1)$ where the declaration in $[\![t, Ctx]\!]_{tr}.decl$ and $init\_var([\![t, Ctx]\!]_{tr}.decl)$ have been done in C, $\widehat{C} \subseteq C$, and $\Lambda = unwrap \circ (M_1 \circ E) \circ Ctx$, we have:*
if $\widehat{E}, \widehat{M_1}, \Lambda \models_t t \Rightarrow value$,
then $E, M_1 \models_s [\![t, Ctx]\!]_{tr}.code \Rightarrow M_2$ and $(M_2 \circ E)([\![t, Ctx]\!]_{tr}.res) = \mathcal{T}([\![t, Ctx]\!]_{tr}.res)(value)$.

Informally, $\widehat{C} \subseteq C$ ensures that all the variables not generated during the generation process have the same value in $\widehat{C}$ and in C. Furthermore, $\Lambda = unwrap \circ (M_1 \circ E) \circ Ctx$ means that the logical variables in the domain of Λ are mapped to a program variable with the same value in the target program. Obviously, these two conditions are necessary for the correctness of the transformation.

Transparency is the property that allows to replace a program chunk by its translation without changing the program behaviour. Side-effects usually prevent transparency (i.e., they cause opacity).

Definition 4 (Code generation transparency). *Let t be a term, the code generation is transparent if and only if, for all code generation context Ctx, semantic contexts $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$such as the declarations in $[\![t, Ctx]\!]_{tr}.decl$ and $init\_var([\![t, Ctx]\!]_{tr}.decl)$ have been done in C_2 and $C_1 \subseteq C_2$, and V is the union between the set of variables in $[\![t, Ctx]\!]_{tr}.decl$ and $codomain(Ctx)$, we have:*
If $E, M_2 \models_s [\![t, Ctx]\!]_{tr}.code \Rightarrow M_t$, then
$(E_1 - \{(v, E_1(v)) \mid v \in V\}, M_1 - \{(E_1(v), M_1(E_1(v)) \mid v \in V\}) \subseteq (E_2, M_t)$.

Informally, transparency insures that executing the code generated for term t does not affect its context. Because terms can be nested, one cannot assume that the declarations generated by a term are close to its code (see for instance Fig. 12). This is why the definition above demands that declarations are represented in the context.

Theorem 3. *The code generation for all term t is sound and transparent.*

The main difficulty for proving Theorem 3 for the extended quantifier `\sum` is to prove that $addition\_assignment$(`sum`, `lbda`) in the generated code never leads to an overflow. We formalize this property in Lemma 6 and prove it in the Appendix E.

Lemma 6 (addition_assignment(`sum`, `lbda`) can not overflow). *Let $C = (E, M)$ be the current context when computing the semantics of $addition\_assignment$(`sum`, `lbda`) in the code generated for `\sum(t1,t2,\lambda x;t3)`. If $E, M \models_e$ `sum` $\Rightarrow \mathcal{T}$(`sum`)(m) and $E, M \models_e$ `lbda` $\Rightarrow \mathcal{T}$(`sum`)(n) then $E, M \models_e addition\_assignment$(`sum`, `lbda`) $\Rightarrow \mathcal{T}$(`sum`)$(m \dotplus n)$.*

Lemma 6 is not trivial because it relates the details of the computation of `\sum(t1,t2,t3)` with the interval and type inferred for it. In particular, if the inferred interval is $I_{sum} = [l_{sum}, u_{sum}]$ with $l_{sum} > 0$, it is not true that variable `sum` takes all its values in I_{sum} (it starts with value 0), but it is true that is takes all its values in $\Theta(I_{sum})$ (= `int` or `mpz`).

3.4 Implementation

Our practical contribution is to have implemented the process detailed in the previous sections inside the **E-ACSL** plug-in of **Frama-C**.

```
mpz __gen_e_acsl__7; mpz __gen_e_acsl__8; mpz __gen_e_acsl_k_6; mpz __gen_e_acsl_one_6;
int __gen_e_acsl_cond_6; int __gen_e_acsl_lambda_6; int __gen_e_acsl_sum_6;
__gmpz_init_set_si(__gen_e_acsl__7,0L);
__gmpz_init_set_ui(__gen_e_acsl__8,18446744073709551615UL);
__gmpz_init_set_si(__gen_e_acsl_one_6,1L);
__gen_e_acsl_cond_6 = 0; __gen_e_acsl_lambda_6 = 0; __gen_e_acsl_sum_6 = 0;
__gmpz_init_set(__gen_e_acsl_k_6,__gen_e_acsl__7);
while (1) {
  __gen_e_acsl_cond_6 = __gmpz_cmp(__gen_e_acsl_k_6,__gen_e_acsl__8));
  if (__gen_e_acsl_cond_6 > 0) break;
  else {
    __gen_e_acsl_lambda_6 = 0;
    __gen_e_acsl_sum_6 += __gen_e_acsl_lambda_6;
    __gmpz_add(__gen_e_acsl_k_6,__gen_e_acsl_k_6,__gen_e_acsl_one_6);
} }
assert(__gen_e_acsl_sum_6 == 0,1);
__gmpz_clear(__gen_e_acsl__7); __gmpz_clear(__gen_e_acsl__8);
__gmpz_clear(__gen_e_acsl_k_6); __gmpz_clear(__gen_e_acsl_one_6);
```

Fig. 13. Example of an **E-ACSL** assertion which computes 0 at a high cost

The main difficulty of this task is to enter in the code base of **Frama-C**, and more particulary, **E-ACSL**. Indeed, the **Frama-C** project contains about 300 000 lines of code, while **E-ACSL** contains 24 000. So, it is not possible at all to understand the overall structure of the code base by reading it. Fortunately, [Correnson et al. 2021] provides an overview of **Frama–C** from the user's point of view, and [Signoles et al. 2021] explains how to develop a plug-in inside **Frama-C**. These two documents are very helpful to understand the code base.

Another difficulty is that the translation of some other terms and predicates of **E-ACSL** have already been implemented. So, our additions must respect the conventions of what has been already developed. In the same spirit, the way **Target** code is generated by an AST with internal invariants to be preserved. This makes it difficult to generate exactly the code that one wants to. Furthermore, some valid constructions of **C** are not allowed by the API. For example, `for` loops and most of the `while` loops are forbidden. The only authorized loops are `while(1){...}` loop escaped by a `break`.

After implementing our translation, we carefully designed integration tests which aimed to catch all the potential source of errors in this process. In particular, we have tested nested extended quantifiers. Finally, our implementation of `\sum`, `\product`, and `\numof` could enter the code review phase before being pushed on the master branch of the **Frama-C** project.

Fig. 1 used as a motivation at the beginning of this report is also a first example of the translations that are obtained. Indeed, it has actually been generated automatically. It is worth remarking that the grammar of the generated code slightly differs from the grammar presented above for **Target**. This is because our formalization only deals with `int` and `mpz`, in order to keep it manageable. In the concrete **Target** code `__gmpz_set_si` is the variant of `mpz_set_int` for signed `long` integers, while `__gmpz_set_ui` is for the unsigned `long`. `__gmpz_init_set_si` and `__gmpz_init_set_ui` are the combination of `__gmpz_init` and `__gmpz_set_...` `__gmpz_set` is similar to `mpz_set_mpz`. One can also see the use of `while (1) {c = not cond; if (c) break; else {body}}` instead of `while (cond) {body}`. It is also worth noting that, unfortunately, temporary variable `__gen_e_acsl_1` is initialized and cleared at each execution of the loop, whereas this could have been done once and for all before and after the `while` loop. It certainly negatively affects the performance of the generated code, but it is a mandatory pattern that comes from the building blocks provided in the code base. Tackling this issue could be the subject of another work.

Fig. 13 shows the translation of `logical_assert(\sum(0,` I`, \lambda k;0)==0)`, where I = 18446744073709551615 (its layout has been edited to keep it small). It illustrates a limitation of our interval analysis for extended quantifiers. Indeed, since I is the largest integer which fits in a bounded **C** integer variable in a 64−bit architecture, a loop from **Min** = 0 to **Max** = I must be computed on unbounded integers otherwise its last incrementation will overflow. However the value of `\sum(0,` I`, \lambda k; 0)` is obviously zero. So the generated code uses an inefficient way for computing an obvious result. In the same way, `\sum(0,` I`, \lambda k; (-1)`^{k}`)` also has a trivial solution. Maybe a more deeper analysis of the terms could tackle with this issue.

4 CONCLUSIONS AND PERSPECTIVES

During this work, we have designed, proved and implemented an *Optimized runtime verification of **ACSL** arithmetic primitives* inside **Frama-C**.

An aspect of this project that we have found interesting is its wide-spectrum coverage of a subject. Indeed, we started with the formalization and proof of the code generation of a toy language and finished with an implementation inside an industrial-strength platform. It allowed us to learn a lot from programming language engineering with the use of inference systems, to software engineering with a contribution inside **Frama-C** including its testing and review mechanism. Thanks to our internship advisor, we have also learned a lot about research methodology.

Another interesting aspect is the verification context of this work, which forces to check every step, whether through proofs for the most upstream developments, or through tests for the most downstream developments. We have been particularly impressed by the array of precautions taken at the end of the life-cycle for publishing a new contribution.

This work inspired us several subjects for further works. We list them from the most immediate to the most ambitious.

As we have seen in section 2.2, it remains two extended quantifiers, `\min` and `\max`, that could be readily accommodated as we have done for `\sum`, `\product`, and `\numof`. The logic of interval inference will be the same as above, and only the translation slightly differs.

The first motivation for this work was to improve the efficiency of **RAC**. However, we have not measured to what extent performances have been improved. Certainly, replacing unbounded integer operations with bounded operations is a good deal, but we do not really know how much it is a good deal. Previous experiments have shown that an actual improvement can be observed for operators of **E-ACSL** which are not extended quantifiers [Kosmatov et al. 2020], but it remains to check if it is still true, on representative use cases, for our implementation of the extended quantifiers. The difficulty is probably to identify what are representative use cases for extended quantifiers.

We have also seen that our interval inference is sometimes sub-optimal. We see at least two paths to tackle this question. The first one is to check if it really harms by benchmarking as above. The second path, only for those who are not afraid of "premature optimization" (according to Donald Knuth, premature optimization is the root of all evil [Knuth 1974]), will consist in refining the analysis in order to avoid the most easy traps. However, it will certainly be an endless quest. Combining these two steps may shed some light on which traps are the most important to avoid.

We have observed that some code generation patterns are sub-optimal (see section 3.4). It is then natural to imagine developing more efficient patterns. However, these patterns are shared by the whole community of **Frama-C** developers. So, the main difficulty could be to reach a consensus on which new generation patterns to introduce, and check that it causes no harm to existing code.

In retrospect, we must admit that the proof of the code generation is done on a code which is very far from the implemented one. When safety or security matters, the whole chain must be consistent. A perspective could be to do the proof with a proof assistant and then extract the code as a **Frama-C** plug-in, as done for instance in [Herms et al. 2012] for a deductive verification tool.

Finally, this work on the implementation of the specification language **ACSL** leads us to wonder why in 2021 we are still incited by most programming languages to use bounded size integers

(Python being a notable exception). We believe a more modern approach would be to have the `int` as true natural numbers, and use specialized libraries for using bounded integers only when efficiency matters. It is already the choice of lesser used programming languages like Scheme, but what we look for is that it becomes the standard offer in mainstream programming languages. Analyses like what is done in the **E-ACSL** compilation of **ACSL** would help to recover performance (only when it is required), but only behind the scene. So doing, the compilation of **ACSL** could be both naïve and efficient, simply because the underlying programming language will be smart.

REFERENCES

Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2020. ACSL: ANSI/ISO C Specification. https://frama-c.com/download/acsl.pdf

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

Lori Clarke and David Rosenblum. 2006. A Historical Perspective on Runtime Assertion Checking in Software Development. *ACM SIGSOFT Software Engineering Notes* 31 (05 2006), 25–37. https://doi.org/10.1145/1127878.1127900

Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. 2021. *Frama-C User Manual.*

Matthew Hennessy. 1990. *The semantics of programming languages: an elementary introduction using structural operational semantics.* John Wiley & Sons, Inc.

Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A certified multi-prover verification condition generator. In *International Conference on Verified Software: Tools, Theories, Experiments.* Springer, 2–17.

Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. 2015. Rester statique pour devenir plus rapide, plus précis et plus mince. In *Journées Francophones des Langages Applicatifs (JFLA)*, David Baelde and Jade Alglave (Eds.). Le Val d'Ajol, France. https://hal.inria.fr/hal-01096352/document

Neil D. Jones. 1997. *Computability and complexity - from a programming perspective.* MIT Press.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609.

Donald E Knuth. 1974. Structured programming with go to statements. *ACM Computing Surveys (CSUR)* 6, 4 (1974), 261–301.

Nikolai Kosmatov, Fonenantsoa Maurica, and Julien Signoles. 2020. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *International Conference on Runtime Verification*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). Springer International Publishing, Cham, 310–322.

René Lalement. 1990. *Logique, réduction, résolution.* Masson.

Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.

Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. 2020. Verified runtime assertion checking for memory properties. In *International Conference on Tests and Proofs.* Springer, 100–121.

John C Reynolds. 2009. *Theories of programming languages.* Cambridge University Press.

Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis.* MIT Press. https://hal.archives-ouvertes.fr/hal-02402597

Julien Signoles, Thibaud Antignac, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto. 2021. *Frama-C Plug-in Development Guide.* http://frama-c.com/download/frama-c-plugin-development-guide.pdf

Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. 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.* 164–173.

A SEMANTICS OF TERMS

Fig. 14 shows the operational semantics of **mini-ACSL** terms, complete for extended quantifiers `\sum`, `\product`, and `\numof`. Note that `\numof` is defined in terms of `\sum`.

B INTERVAL INFERENCE RULES FOR \SUM, \PRODUCT AND \NUMOF

Fig. 15, 16 and 17 show respectively the interval inference rules for `\sum(t1,t2,\lambda x;t3)`, `\product(t1,t2,\lambda x;t3)` and `\numof(t1,t2,\lambda x;t3)`. They all repeat the interval overlapping situations that are explained in Section 3.1. Only the way of using of the resulting intervals

$$\frac{E,M,\Lambda \models_e e \Rightarrow int(v)}{E,M,\Lambda \models_t e \Rightarrow v} \qquad \frac{}{E,M,\Lambda \models_t \mathtt{x} \Rightarrow \Lambda(\mathtt{x})} \qquad \frac{E,M,\Lambda \models_t \mathtt{t1} \Rightarrow v \qquad E,M,\Lambda \models_t \mathtt{t2} \Rightarrow w}{E,M,\Lambda \models_t \mathtt{t1} \diamond \mathtt{t2} \Rightarrow v \,\dot{\diamond}\, w}$$

$$\frac{E,M,\Lambda \models_p \mathtt{condition} \Rightarrow True \qquad E,M,\Lambda \models_t \mathtt{t1} \Rightarrow v1}{E,M,\Lambda \models_t \mathtt{condition\ ?\ t1:\ t2} \Rightarrow v1} \qquad \frac{E,M,\Lambda \models_p \mathtt{condition} \Rightarrow False \qquad E,M,\Lambda \models_t \mathtt{t2} \Rightarrow v2}{E,M,\Lambda \models_t \mathtt{condition\ ?\ t1:\ t2} \Rightarrow v2}$$

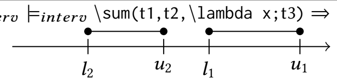
$$\frac{E,M,\Lambda \models_t \mathtt{t1} \Rightarrow min \qquad E,M,\Lambda \models_t \mathtt{t2} \Rightarrow max \qquad \forall k \in [min,max](E,M,\Lambda[\mathtt{x} \to k] \models_t \mathtt{t3} \Rightarrow lambda_k)}{E,M,\Lambda \models_t \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow \sum_{k=min}^{max} lambda_k}$$

$$\frac{E,M,\Lambda \models_t \mathtt{t1} \Rightarrow min \qquad E,M,\Lambda \models_t \mathtt{t2} \Rightarrow max \qquad \forall k \in [min,max](E,M,\Lambda[\mathtt{x} \to k] \models_t \mathtt{t3} \Rightarrow lambda_k)}{E,M,\Lambda \models_t \mathtt{\backslash product(t1,t2,\backslash lambda\ x;t3)} \Rightarrow \prod_{k=min}^{max} lambda_k}$$

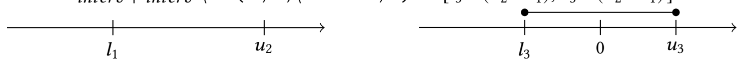
$$\frac{E,M,\Lambda \models_t \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;p3\ ?\ 1:\ 0)} \Rightarrow v}{E,M,\Lambda \models_t \mathtt{\backslash numof(t1,t2,\backslash lambda\ x;p3)} \Rightarrow v}$$

Fig. 14. Semantics of terms.

$$\frac{\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \qquad \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad l_1 > u_2}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [0,0]}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \\ \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad \Gamma_{interv}, x := [l_1,u_2] \models_{interv} \mathtt{t3} \Rightarrow [l_3,u_3] \qquad l_1 \le u_2 \qquad l_3 \le 0 \le u_3\end{array}}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [l_3 * (u_2 - l_1), u_3 * (u_2 - l_1)]}$$



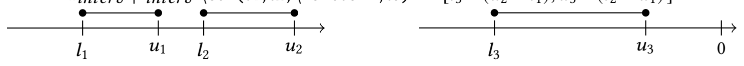
$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \\ \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad \Gamma_{interv}, x := [l_1,u_2] \models_{interv} \mathtt{t3} \Rightarrow [l_3,u_3] \qquad l_1 \le u_2 \qquad u_1 \le l_2 \qquad 0 < l_3 \le u_3\end{array}}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [l_3 * (l_2 - u_1), u_3 * (u_2 - l_1)]}$$

l_1
u_1
l_2
u_2
0
l_3
u_3

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \\ \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad \Gamma_{interv}, x := [l_1,u_2] \models_{interv} \mathtt{t3} \Rightarrow [l_3,u_3] \qquad l_1 \le u_2 \qquad u_1 > l_2 \qquad 0 < l_3 \le u_3\end{array}}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [0, u_3 * (u_2 - l_1)]}$$

l_1
l_2
u_1
u_2
0
l_3
u_3

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \\ \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad \Gamma_{interv}, x := [l_1,u_2] \models_{interv} \mathtt{t3} \Rightarrow [l_3,u_3] \qquad l_1 \le u_2 \qquad u_1 \le l_2 \qquad l_3 \le u_3 < 0\end{array}}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [l_3 * (u_2 - l_1), u_3 * (l_2 - u_1)]}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \mathtt{t1} \Rightarrow [l_1,u_1] \\ \Gamma_{interv} \models_{interv} \mathtt{t2} \Rightarrow [l_2,u_2] \qquad \Gamma_{interv}, x := [l_1,u_2] \models_{interv} \mathtt{t3} \Rightarrow [l_3,u_3] \qquad l_1 \le u_2 \qquad u_1 > l_2 \qquad l_3 \le u_3 < 0\end{array}}{\Gamma_{interv} \models_{interv} \mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)} \Rightarrow [l_3 * (u_2 - l_1), 0]}$$

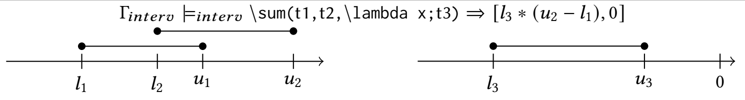


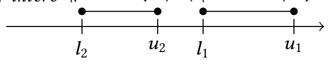
Fig. 15. Interval inference rules for `\sum(t1, t2, \lambda x;t3)`.

differs from an extended quantifier to another. Note also that product having 1 as a neutral element some overlapping situations lead to bound 1 instead of bound 0.

C PROOF OF THEOREM 1

Theorem 1 (Interval inference system is correct). *The inference system denoted by $\models_{interv}$ is correct for all term t. That is to say, in an interval context Γ_{interv} consistent with a logical context Λ,*

$$\frac{\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \qquad \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad l_1 > u_2}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [1, 1])}$$



$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2 \qquad l_3 \le 0 \le u_3 \qquad u_3 \le -l_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [-(-l_3)^{u_2-l_1}, (-l_3)^{u_2-l_1}]}$$

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2 \qquad l_3 \le 0 \le u_3 \qquad u_3 > -l_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [-u_3^{u_2-l_1}, u_3^{u_2-l_1}]}$$

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2 \qquad u_1 \le l_2 \qquad 0 < l_3 \le u_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [l_3^{l_2-u_1}, u_3^{u_2-l_1}]}$$

l_1
u_1
l_2
u_2
0
l_3
u_3

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2 \qquad u_1 > l_2 \qquad 0 < l_3 \le u_3\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [1, u_3^{u_2-l_1}])}$$

l_1
l_2
u_1
u_2
0
l_3
u_3

$$\frac{\begin{array}{c}\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \\ \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2 \qquad l_3 \le u_3 < 0\end{array}}{\Gamma_{interv} \models_{interv} \texttt{\product(t1,t2,\lambda x;t3)} \Rightarrow [-(-l_3)^{u_2-l_1)}, (-l_3)^{u_2-l_1}])}$$

Fig. 16. Interval inference rules for `\product(t1, t2, \lambda x;t3)`.

$$\frac{\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \qquad \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad \Gamma_{interv}, x := [l_1, u_2] \models_{interv} \texttt{t3} \Rightarrow [l_3, u_3] \qquad l_1 \le u_2}{\Gamma_{interv} \models_{interv} \texttt{\numof(t1,t2,\lambda x;t3)} \Rightarrow [0, u_2 - l_1]}$$

$$\frac{\Gamma_{interv} \models_{interv} \texttt{t1} \Rightarrow [l_1, u_1] \qquad \Gamma_{interv} \models_{interv} \texttt{t2} \Rightarrow [l_2, u_2] \qquad l_1 > u_2}{\Gamma_{interv} \models_{interv} \texttt{\numof(t1,t2,\lambda x;t3)} \Rightarrow [0, 0])}$$

Fig. 17. Interval inference rules for `\numof(t1, t2, \lambda x;t3)`.

and a semantic context $\widehat{C} = (\widehat{E}, \widehat{M})$ where all the variables in t have been declared, if $\widehat{E}, \widehat{M}, \Lambda \models_t t \Rightarrow val$ and $\Gamma_{interv} \models_{interv} t \Rightarrow interv$ then $val \in interv$.

It can be proved by structural induction on terms. We detail the proof for the terms of the form `\sum(t1, t2, \lambda x;t3)` and `\product(t1, t2, \lambda x;t3)`.

Recall that

$$\frac{\widehat{E}, \widehat{M}, \Lambda \models_t \texttt{t1} \Rightarrow min \qquad \widehat{E}, \widehat{M}, \Lambda \models_t \texttt{t2} \Rightarrow max \qquad \forall k \in [min, max](\widehat{E}, \widehat{M}, \Lambda[\texttt{x} \to k] \models_t \texttt{t3} \Rightarrow lambda_k)}{\widehat{E}, \widehat{M}, \Lambda \models_t \texttt{\sum(t1,t2,\lambda x;t3)} \Rightarrow \sum_{k=min}^{max} lambda_k}$$

Prove that if Γ_{interv} is consistent with Λ, $\Gamma_{interv} \models_{interv}$ `\sum(t1,t2,\lambda x;t3)` $\Rightarrow interv$ and $\widehat{E}, \widehat{M}, \Lambda \models_t$ `\sum(t1,t2,\lambda x;t3)` $\Rightarrow \sum_{k=min}^{max} lambda_k$ then $\sum_{k=min}^{max} lambda_k \in interv$.

From the induction hypothesis that the inference system is correct for all the subterms of `\sum(t1,t2,\lambda x;t3)`, and the hypothesis that Γ_{interv} is consistent with Λ, we have:

(1) If $\Gamma_{interv} \models_{interv}$ `t1` $\Rightarrow [l_1, u_1]$ and $\Gamma_{interv} \models_{interv}$ `t2` $\Rightarrow [l_2, u_2]$, then $min \in [l_1, u_1]$, $max \in [l_2, u_2]$,

(2) and if $k \in [l_1, u_2]$, then $\Gamma_{interv}, x := [l_1, u_2]$ is consistent with $\Lambda, x := k$, so if $\Gamma_{interv}, x := [l_1, u_2] \models_{interv}$ `t3` $\Rightarrow [l_3, u_3]$, then $lambda_k \in [l_3, u_3]$.

We make these assumptions for the rest of the proof which proceeds rule after rule.

First rule:

i.e., $l_1 > u_2$, then $min > max$, so the range of the quantifier is empty, and its value is its neutral element: $\sum_{k=min}^{max} lambda_k = 0$ and $\sum_{k=min}^{max} lambda_k \in [0, 0]$.

All other rules:

i.e., $l_1 \leq u_2$, the maximum range $[l_1, u_2]$ of the quantifier is not empty,
however, the minimum range $[u_1, l_2]$ of the quantifier can be empty (if min and max overlap),
then, let $k \in [l_1, u_2]$ and $lambda_k \in [l_3, u_3]$ (property (2))
then, $l_3 * \sum_{k=min}^{max} 1 = \sum_{k=min}^{max} l_3 \leq \sum_{k=min}^{max} lambda_k \leq \sum_{k=min}^{max} u_3 = u_3 * \sum_{k=min}^{max} 1$
then, the analysis of overlapping situations determines the bounds of $\sum_{k=min}^{max} 1$.

Function values span 0:

i.e., $l_3 \leq 0 \leq u_3$, maximize the factors of l_3 and u_3 to find a sound interval,
then, $l_3 * (u_2 - l_1) \leq l_3 * \sum_{k=min}^{max} 1$ and $u_3 * \sum_{k=min}^{max} 1 \leq u_3 * (u_2 - l_1)$.
So, $\sum_{k=min}^{max} lambda_k \in [l_3 * (u_2 - l_1), u_3 * (u_2 - l_1)]$.

Function values are positive and *min* and *max* do not overlap:

i.e., $u_1 \leq l_2$ and $0 < l_3 \leq u_3$, minimize the factor of l_3 and maximize the factor of u_3 to find a sound interval,
then, $l_3 * (l_2 - u_1) \leq l_3 * \sum_{k=min}^{max} 1$ and $u_3 * \sum_{k=min}^{max} 1 \leq u_3 * (u_2 - l_1)$.
So, $\sum_{k=min}^{max} lambda_k \in [l_3 * (l_2 - u_1), u_3 * (u_2 - l_1)]$.

Function values are positive and *min* and *max* overlap:

i.e., $u_1 > l_2$ and $0 < l_3 \leq u_3$, as previous case, except the minimum range of the quantifier can be empty, 0 is a minimum,
then, $0 \leq l_3 * \sum_{k=min}^{max} 1$ and $u_3 * \sum_{k=min}^{max} 1 \leq u_3 * (u_2 - l_1)/$
So, $\sum_{k=min}^{max} lambda_k \in [0, u_3 * (u_2 - l_1)]$.

Function values are negative and *min* and *max* do not overlap:

i.e., $u_1 \leq l_2$ and $l_3 \leq u_3 < 0$, maximize the factor of l_3 and minimize the factor of u_3 to find a sound interval,
then, $l_3 * (u_2 - l_1) \leq l_3 * \sum_{k=min}^{max} 1$ and $u_3 * \sum_{k=min}^{max} 1 \leq u_3 * (l_2 - u_1)$.
So, $\sum_{k=min}^{max} lambda_k \in [l_3 * (u_2 - l_1), u_3 * (l_2 - u_1)]$.

Function values are negative and *min* and *max* overlap:

i.e., $u_1 > l_2$ and $l_3 \leq u_3 < 0$, as previous case, except 0 is a maximum,
then, $l_3 * (u_2 - l_1) \leq l_3 * \sum_{k=min}^{max} 1$ and $u_3 * \sum_{k=min}^{max} 1 = 0$.
So, $\sum_{k=min}^{max} lambda_k \in [l_3 * (u_2 - l_1), 0]$.

To, conclude, in all cases, if $\Gamma_{interv} \models_{interv}$ `\sum(t1,t2,\lambda x;t3)` $\Rightarrow interv$ then $\sum_{k=min}^{max} lambda_k \in interv$, so the interval inference system is correct for all terms of the form `\sum(t1,t2,\lambda x;t3)`.

For `\product` recall that

$$\frac{\widehat{E}, \widehat{M}, \Lambda \models_t \texttt{t1} \Rightarrow min \qquad \widehat{E}, \widehat{M}, \Lambda \models_t \texttt{t2} \Rightarrow max \qquad \forall k \in [min, max](\widehat{E}, \widehat{M}, \Lambda[\texttt{x} \to k] \models_t \texttt{t3} \Rightarrow lambda_k)}{\widehat{E}, \widehat{M}, \Lambda \models_t \texttt{\textbackslash product(t1,t2,\textbackslash lambda x;t3)} \Rightarrow \prod_{k=min}^{max} lambda_k}$$

Prove that if Γ_{interv} is consistent with Λ, $\Gamma_{interv} \models_{interv}$ `\product(t1,t2,\lambda x;t3)` $\Rightarrow interv$ and $\widehat{E}, \widehat{M}, \Lambda \models_t$ `\product(t1,t2,\lambda x;t3)` $\Rightarrow \prod_{k=min}^{max} lambda_k$ then $\prod_{k=min}^{max} lambda_k \in interv$. The proof follows similar principles as the previous one, but the overlapping situations are different because products of intervals behave differently from sums of intervals.

From the induction hypothesis that the inference system is correct for all the subterms of `\product(t1,t2,\lambda x;t3)`, because Γ_{interv} is consistent with Λ, we have:

(1) If $\Gamma_{interv} \models_{interv}$ `t1` $\Rightarrow [l_1, u_1]$ and $\Gamma_{interv} \models_{interv}$ `t2` $\Rightarrow [l_2, u_2]$, then $min \in [l_1, u_1]$, $max \in [l_2, u_2]$,
(2) If $k \in [l_1, u_2]$, then $\Gamma_{interv}, x := [l_1, u_2]$ is consistent with $\Lambda, x := k$, so if $\Gamma_{interv}, x := [l_1, u_2] \models_{interv}$ `t3` $\Rightarrow [l_3, u_3]$, then $lambda_k \in [l_3, u_3]$.

We make these assumptions for the rest of the proof which proceeds rule after rule.

First rule: If $l_1 > u_2$, then $min > max$ so the range of the quantifier is empty, and its value is its neutral element: $\prod_{k=min}^{max} lambda_k = 1$ and $\prod_{k=min}^{max} lambda_k \in [1, 1]$.

All other rules: If $l_1 \leq u_2$ then the maximum range $[l_1, u_2]$ of the quantifier is not empty, however, the minimum range $[u_1, l_2]$ of the quantifier can be empty (if min and max overlap), then, let $k \in [l_1, u_2]$ and $lambda_k \in [l_3, u_3]$ (property (2))
then, the analysis of signs and of overlapping situations determines the bounds of $\prod_{k=min}^{max} lambda_k$. Note that we cannot infer the parity of the number of values in the product, so we can deduce nothing on this basis.

Function values span 0 and l_3 has greatest absolute value:
i.e., $l_3 \leq 0 \leq u_3$ and $u_3 \leq -l_3$, positive values are possible as well as negative values,
then, $-(-l_3)^{max(\sum_{k=min}^{max} 1)} \leq -\prod_{k=min}^{max} -l_3 \leq \prod_{k=min}^{max} lambda_k$
and $\prod_{k=min}^{max} lambda_k \leq \prod_{k=min}^{max} -l_3 \leq (-l_3)^{max(\sum_{k=min}^{max} 1)}$.
So, $\prod_{k=min}^{max} lambda_k \in [-(-l_3)^{(u_2-l_1)}, (-l_3)^{(u_2-l_1)}]$.

Function values span 0 and u_3 has greatest absolute value:
i.e., $l_3 \leq 0 \leq u_3$ and $u_3 > -l_3$, positive values are possible as well as negative values,
then, $-(u_3)^{max(\sum_{k=min}^{max} 1)} \leq -\prod_{k=min}^{max} u_3 \leq \prod_{k=min}^{max} lambda_k \leq \prod_{k=min}^{max} u_3 \leq (u_3)^{max(\sum_{k=min}^{max} 1)}$.
So, $\prod_{k=min}^{max} lambda_k \in [-(u_3)^{(u_2-l_1)}, (u_3)^{(u_2-l_1)}]$.

Function values are positive and *min* and *max* do not overlap:
i.e., $0 < l_3 \leq u_3$ and $u_1 \leq l_2$, only strictly positive values are possible, the range of the quantifier cannot be empty,
then, $l_3^{min(\sum_{k=min}^{max} 1)} \leq \prod_{k=min}^{max} l_3 \leq \prod_{k=min}^{max} lambda_k \leq \prod_{k=min}^{max} u_3 \leq u_3^{max(\sum_{k=min}^{max} 1)}$.
So, $\prod_{k=min}^{max} lambda_k \in [l_3^{(l_2-u_1)}, u_3^{(u_2-l_1)}]$.

Function values are positive and *min* and *max* overlap:
i.e., $0 < l_3 \leq u_3$ and $u_1 > l_2$, only strictly positive values are possible, the range of the quantifier can be empty,
then, $l_3^{\sum_{k=min}^{max} 1} \leq \prod_{k=min}^{max} l_3 \leq \prod_{k=min}^{max} lambda_k \leq \prod_{k=min}^{max} u_3 \leq u_3^{\sum_{k=min}^{max} 1}$.
So, $\prod_{k=min}^{max} lambda_k \in [1, u_3^{(u_2-l_1)}]$.

Function values are negative:
i.e., $l_3 \leq u_3 < 0$, only negative values are possible, but since we do not know the parity of the number of values, we can deduce nothing,
then, $-(-l_3)^{max(\sum_{k=min}^{max} 1)} \leq -\prod_{k=min}^{max} -l_3 \leq \prod_{k=min}^{max} lambda_k$
and $\prod_{k=min}^{max} lambda_k \leq -\prod_{k=min}^{max} -u_3 \leq -(-u_3)^{max(\sum_{k=min}^{max} 1)}$.
So, $\prod_{k=min}^{max} lambda_k \in [-(-l_3)^{(u_2-l_1)}, -(-u_3)^{(u_2-l_1)}]$.

To, conclude, in all cases, if $\Gamma_{interv} \models_{interv}$ `\product(t1,t2,\lambda x;t3)` $\Rightarrow$ *interv* then $\prod_{k=min}^{max} lambda_k \in interv$, so the interval inference system is correct for the terms of the form `\product(t1,t2,\lambda x;t3)`.

D SEMANTICS OF MACROS

Macros have been defined for factoring different cases of code generation (see Fig. 11). Their semantics is defined as deduction rules that are derived from the semantics of **Target** (see Fig. 5, 6, 7, and 8). These deduction rules are used in the proof of the code generation from **Source** to **Target**. We give here the proofs of these deduction rules. All proofs are proofs by cases organized according to the different type cases that structure the definition of the *macros*.

LEMMA 1 (SEMANTICS OF *int_assignment*).

$$\frac{min\_int \dot{\leq} valof(\mathsf{n}) \dot{\leq} max\_int \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow \mathcal{T}(\mathsf{var})(valof(\mathsf{n}))]}{E, M_1 \models_s int\_assignment(\mathsf{var}, \mathsf{n}) \Rightarrow M_2}$$

PROOF OF LEMMA 1. **Case `int`:** `var=n` is generated, so according to the semantics of statements:

$$\frac{\mathcal{T}(\mathsf{var}) = \mathsf{int} \qquad E, M_1 \models_e \mathsf{n} \Rightarrow int(v) \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow int(v)]}{E, M_1 \models_s \mathsf{var=n} \Rightarrow M_2}$$

so according to the semantics of expressions

$$\frac{\mathcal{T}(\mathsf{var}) = \mathsf{int} \qquad min\_int \dot{\leq} valof(\mathsf{n}) \dot{\leq} max\_int \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow int(valof(\mathsf{n}))]}{E, M_1 \models_s \mathsf{var=n} \Rightarrow M_2}$$

Case `mpz`: `mpz_set_int(var,n)` is generated, so according to the semantics of statements:

$$\frac{\mathcal{T}(\mathsf{var}) = \mathsf{mpz} \qquad E, M_1 \models_e \mathsf{n} \Rightarrow int(v) \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow mpz(v)]}{E, M_1 \models_s \mathsf{mpz\_set\_int(var,n)} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\mathcal{T}(\mathsf{var}) = \mathsf{mpz} \qquad min\_int \dot{\leq} valof(\mathsf{n}) \dot{\leq} max\_int \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow mpz(valof(\mathsf{n}))]}{E, M_1 \models_s \mathsf{mpz\_set\_int(var,n)} \Rightarrow M_2}$$

These two rules can be unified in rule

$$\frac{\mathcal{T}(\mathsf{var}) = \mathcal{T}(\mathsf{var}) \qquad min\_int \dot{\leq} valof(\mathsf{n}) \dot{\leq} max\_int \qquad M_2 = M_1[E(\mathsf{var}) \rightarrow \mathcal{T}(\mathsf{var})(valof(\mathsf{n}))]}{E, M_1 \models_s int\_assignment(\mathsf{var}, \mathsf{n}) \Rightarrow M_2}$$

Since $\mathcal{T}(\mathsf{var}) = \mathcal{T}(\mathsf{var})$ is trivially true, this rule can be simplified in the rule of Lemma 1. □

LEMMA 2 (SEMANTICS OF *var_assignment*).

$$\frac{(M_1 \circ E)(\mathsf{var2})) = \mathcal{T}(\mathsf{var2})(v) \qquad M_2 = M_1[E(\mathsf{var1}) \rightarrow \mathcal{T}(\mathsf{var1})(v)]}{E, M_1 \models_m var\_assignment(\mathsf{var1}, \mathsf{var2}) \Rightarrow M_2}$$

PROOF OF LEMMA 2. **Case `int,int`:** `var1=var2` is generated, so according to the semantics of statements:

$$\frac{\mathcal{T}(\mathsf{var1}) = \mathsf{int} \qquad E, M \models_e \mathsf{var2} \Rightarrow int(v) \qquad M_2 = M_1[E(\mathsf{var1}) \rightarrow int(v)]}{E, M_1 \models_s \mathsf{var1=var2} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\mathcal{T}(\mathsf{var1}) = \mathsf{int} \qquad \mathcal{T}(\mathsf{var2}) = \mathsf{int} \qquad (M \circ E)(\mathsf{var2}) = int(v) \qquad M_2 = M_1[E(\mathsf{var1}) \rightarrow int(v)]}{E, M_1 \models_s \mathsf{var1=var2} \Rightarrow M_2}$$

Case mpz,int: `mpz_set_int(var1,var2)` is generated, so according to the semantics of statements:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \qquad E, M \models_e \mathtt{var2} \Rightarrow \mathit{int}(v) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{mpz}(v)]}{E, M_1 \models_s \mathtt{mpz\_set\_int(var1,var2)} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \qquad \mathcal{T}(\mathtt{var2}) = \mathtt{int} \qquad (M \circ E)(\mathtt{var2}) = \mathit{int}(v) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{mpz}(v)]}{E, M_1 \models_s \mathtt{mpz\_set\_int(var1,var2)} \Rightarrow M_2}$$

Case mpz,mpz: `mpz_set_mpz(var1,var2)` is generated, so according to the semantics of statements:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \\ \mathcal{T}(\mathtt{var2}) = \mathtt{mpz} \qquad (M \circ E)(\mathtt{var2}) = \mathit{mpz}(v) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{mpz}(v)]\end{array}}{E, M_1 \models_s \mathtt{mpz\_set\_mpz(var1,var2)} \Rightarrow M_2}$$

These three rules can be unified in the rule:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathcal{T}(\mathtt{var1}) \\ \mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var2}) \qquad (M_1 \circ E)(\mathtt{var2})) = \mathcal{T}(\mathtt{var2})(v) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathcal{T}(\mathtt{var1})(v)]\end{array}}{E, M_1 \models_m \mathit{var\_assignment}(\mathtt{var1}, \mathtt{var2}) \Rightarrow M_2}$$

Since $\mathcal{T}(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})$ and $\mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var2})$ are trivially true, this rule can be simplified in the rule of Lemma 2. □

Lemma 3 (Semantics of *addition_assignment*).

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var1}) \qquad (M_1 \circ E)(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})(v_1) \qquad (M_1 \circ E)(\mathtt{var2}) = \mathcal{T}(\mathtt{var1})(v_2) \\ \mathit{min\_type}(\mathcal{T}(\mathtt{var1})) \mathrel{\dot{\leq}} v_1 \mathbin{\dot{+}} v_2 \mathrel{\dot{\leq}} \mathit{max\_type}(\mathcal{T}(\mathtt{var1})) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathcal{T}(\mathtt{var1})(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_m \mathit{addition\_assignment}(\mathtt{var1}, \mathtt{var2}) \Rightarrow M_2}$$

Proof of Lemma 3. **Case int:** `var1=var1+var2` is generated, so according to the semantics of statements:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{int} \qquad E, M_1 \models_e \mathtt{var1+var2} \Rightarrow \mathit{int}(v) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{int}(v)]}{E, M_1 \models_s \mathtt{var1=var1+var2} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathtt{int} \qquad E, M_1 \models_e \mathtt{var1} \Rightarrow \mathit{int}(v_1) \\ E, M_1 \models_e \mathtt{var2} \Rightarrow \mathit{int}(v_2) \qquad \mathit{min\_int} \mathrel{\dot{\leq}} v_1 \mathbin{\dot{+}} v_2 \mathrel{\dot{\leq}} \mathit{max\_int} \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{int}(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_s \mathtt{var1=var1+var2} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathtt{int} \qquad (M_1 \circ E)(\mathtt{var1}) = \mathit{int}(v_1) \qquad \mathcal{T}(\mathtt{var2}) = \mathtt{int} \qquad (M_1 \circ E)(\mathtt{var2})) = \mathit{int}(v_2) \\ \mathit{min\_int} \mathrel{\dot{\leq}} v_1 \mathbin{\dot{+}} v_2 \mathrel{\dot{\leq}} \mathit{max\_int} \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{int}(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_s \mathtt{var1=var1+var2} \Rightarrow M_2}$$

Case mpz: `mpz_add(var1,var1,var2)` is generated, so according to the semantics of statements:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \qquad \mathcal{T}(\mathtt{var2}) = \mathtt{mpz} \qquad (M_1 \circ E)(\mathtt{var1})) = \mathit{mpz}(v_1) \\ (M_1 \circ E)(\mathtt{var2}) = \mathit{mpz}(v_2) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{mpz}(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_s \mathtt{mpz\_add(var1,var1,var2)} \Rightarrow M_2}$$

so according to the semantics of expressions:

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \qquad \mathcal{T}(\mathtt{var2}) = \mathtt{mpz} \\ (M_1 \circ E)(\mathtt{var1}) = \mathit{mpz}(v_1) \qquad (M_1 \circ E)(\mathtt{var2}) = \mathit{mpz}(v_2) \qquad M_2 = M_1[E(\mathtt{var1}) \rightarrow \mathit{mpz}(v_1 \mathbin{\dot{+}} v_2)]\end{array}}{E, M_1 \models_s \mathtt{mpz\_add(var1,var1,var2)} \Rightarrow M_2}$$

Since $min\_mpz = -\infty$ and $max\_mpz = +\infty$, $min\_mpz \dot{\leq} v_1 \dot{+} v_2 \dot{\leq} max\_mpz$ is trivially true so it can be added to the hypothesis without changing the semantics:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \quad \mathcal{T}(\mathtt{var2}) = \mathtt{mpz} \quad (M_1 \circ E)(\mathtt{var1}) = mpz(v_1) \quad (M_1 \circ E)(\mathtt{var2}) = mpz(v_2) \\ min\_mpz \dot{\leq} v_1 \dot{+} v_2 \dot{\leq} max\_mpz \quad M_2 = M_1[E(\mathtt{var1}) \to mpz(v_1 \dot{+} v_2)]}{E, M_1 \models_s \mathtt{mpz\_add(var1,var1,var2)} \Rightarrow M_2}$$

These two rules can be unified in the rule:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathcal{T}(\mathtt{var1}) \\ \mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var2}) \quad \mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var1}) \quad (M_1 \circ E)(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})(v_1) \\ (M_1 \circ E)(\mathtt{var2}) = \mathcal{T}(\mathtt{var1})(v_2) \quad M_2 = M_1[E(\mathtt{var1}) \to \mathcal{T}(\mathtt{var1})(\mathcal{O}(\mathcal{T}(\mathtt{var1}), v_1 \dot{+} v_2))]}{E, M_1 \models_m addition\_assignment(\mathtt{var1}, \mathtt{var2}) \Rightarrow M_2}$$

Since $\mathcal{T}(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})$ and $\mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var2})$ are trivially true, this rule can be simplified in the rule of Lemma 3. □

Lemma 5 (Semantics of *condition*).

$$\frac{\mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var1}) \quad (M_1 \circ E)(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})(v_1) \\ (M_1 \circ E)(\mathtt{var2})) = \mathcal{T}(\mathtt{var1})(v_2) \quad v_1 \dot{\leq} v_2 \quad v \neq 0 \quad min\_int \dot{\leq} v \dot{\leq} max\_int}{E, M_1 \models_m condition(\mathtt{var1}, \mathtt{var2}) \Rightarrow int(v)}$$

$$\frac{\mathcal{T}(\mathtt{var2}) = \mathcal{T}(\mathtt{var1}) \quad (M_1 \circ E)(\mathtt{var1}) = \mathcal{T}(\mathtt{var1})(v_1) \quad (M_1 \circ E)(\mathtt{var2})) = \mathcal{T}(\mathtt{var1})(v_2) \quad v_1 \dot{>} v_2}{E, M_1 \models_m condition(\mathtt{var1}, \mathtt{var2}) \Rightarrow int(0)}$$

Proof of Lemma 5. **Case `int`:** `var1<=var2` is generated, so according to the semantics of expressions:

$$\frac{E, M_1 \models_e \mathtt{var1} \Rightarrow int(v_1) \\ E, M_1 \models_e \mathtt{var2} \Rightarrow int(v_2) \quad v_1 \dot{\leq} v_2 \quad v \neq 0 \quad min\_int \dot{\leq} v \dot{\leq} max\_int}{E, M \models_e \mathtt{var1<=var2} \Rightarrow int(v)}$$

$$\frac{E, M_1 \models_e \mathtt{var1} \Rightarrow int(v_1) \quad E, M_1 \models_e \mathtt{var2} \Rightarrow int(v_2) \quad v_1 \dot{>} v_2}{E, M \models_e \mathtt{var1<=var2} \Rightarrow int(0)}$$

so according to the semantics of expressions:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{int} \quad \mathcal{T}(\mathtt{var2}) = \mathtt{int} \quad (M_1 \circ E)(\mathtt{var1}) = int(v_1) \\ (M_1 \circ E)(\mathtt{var2}) = int(v_2) \quad v_1 \dot{\leq} v_2 \quad v \neq 0 \quad min\_int \leq v \leq max\_int}{E, M \models_e \mathtt{var1<=var2} \Rightarrow int(v)}$$

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{int} \\ \mathcal{T}(\mathtt{var2}) = \mathtt{int} \quad (M_1 \circ E)(\mathtt{var1}) = int(v_1) \quad (M_1 \circ E)(\mathtt{var2}) = int(v_2) \quad v_1 \dot{>} v_2}{E, M \models_e \mathtt{var1<=var2} \Rightarrow int(0)}$$

Case `mpz`: `mpz_cmp(var1,var2)<=0` is generated, so according to the semantics of expressions

$$\frac{E, M_1 \models_e \mathtt{mpz\_cmp(var1,var2)} \Rightarrow int(u) \\ E, M \models_e 0 \Rightarrow int(0) \quad u \dot{\leq} 0 \quad v \neq 0 \quad min\_int \dot{\leq} v \dot{\leq} max\_int}{E, M \models_e \mathtt{mpz\_cmp(var1,var2)<=0} \Rightarrow int(v)}$$

$$\frac{E, M_1 \models_e \mathtt{mpz\_cmp(var1,var2)} \Rightarrow int(u) \quad E, M \models_e 0 \Rightarrow int(0) \quad u \dot{>} 0}{E, M \models_e \mathtt{mpz\_cmp(var1,var2)<=0} \Rightarrow int(0)}$$

so according to the semantics of `mpz_cmp` expressions:

$$\frac{\mathcal{T}(\mathtt{var1}) = \mathtt{mpz} \quad \mathcal{T}(\mathtt{var2}) = \mathtt{mpz} \quad (M_1 \circ E)(\mathtt{var1}) = mpz(v_1) \quad (M_1 \circ E)(\mathtt{var2}) = mpz(v_2) \\ v_1 \dot{\leq} v_2 \quad min\_int \dot{\leq} u \dot{\leq} 0 \quad u \dot{\leq} 0 \quad v \neq 0 \quad min\_int \dot{\leq} v \dot{\leq} max\_int}{E, M \models_e \mathtt{mpz\_cmp(var1,var2)<=0} \Rightarrow int(1)}$$

$$\frac{\mathcal{T}(\texttt{var1}) = \texttt{mpz} \qquad \mathcal{T}(\texttt{var2}) = \texttt{mpz} \qquad (M_1 \circ E)(\texttt{var1}) = mpz(v_1) \\ (M_1 \circ E)(\texttt{var2}) = mpz(v_2) \qquad v_1 \dot{>} v_2 \qquad 0 \dot{<} u \dot{\leq} max\_int \qquad u \dot{>} 0}{E, M \models_e \texttt{mpz\_cmp(var1,var2)<=0} \Rightarrow int(0)}$$

These rules can be unified in the rules:

$$\frac{\mathcal{T}(\texttt{var1}) = \mathcal{T}(\texttt{var1}) \qquad \mathcal{T}(\texttt{var2}) = \mathcal{T}(\texttt{var2}) \qquad (M_1 \circ E)(\texttt{var1}) = \mathcal{T}(\texttt{var1})(v_1) \\ (M_1 \circ E)(\texttt{var2})) = \mathcal{T}(\texttt{var1})(v_2) \qquad v_1 \dot{\leq} v_2 \qquad v \neq 0 \qquad min\_int \dot{\leq} v \dot{\leq} max\_int}{E, M_1 \models_m condition(\texttt{var1}, \texttt{var2}) \Rightarrow int(v)}$$

$$\frac{\mathcal{T}(\texttt{var1}) = \mathcal{T}(\texttt{var1}) \\ \mathcal{T}(\texttt{var2}) = \mathcal{T}(\texttt{var2}) \qquad (M_1 \circ E)(\texttt{var1}) = \mathcal{T}(\texttt{var1})(v_1) \qquad (M_1 \circ E)(\texttt{var2})) = \mathcal{T}(\texttt{var1})(v_2) \qquad v_1 \dot{>} v_2}{E, M_1 \models_m condition(\texttt{var1}, \texttt{var2}) \Rightarrow int(0)}$$

Since $\mathcal{T}(\texttt{var1}) = \mathcal{T}(\texttt{var1})$ and $\mathcal{T}(\texttt{var2}) = \mathcal{T}(\texttt{var2})$ are trivially true, this rule can be simplified in the rule of Lemma 5. □

E CODE GENERATION PROOF

Lemma 7. *Let $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ be two semantic contexts such as $C_1 \subseteq C_2$, and a variable* v *declared in C_2 but not in C_1.*

An assignment (i.e., $int\_assignment$, $var\_assignment$, $addition\_assignment$ or $multiplication\_assignment$) to v *in C_2 yields a memory M_3 such as $C_1 \subseteq (E_2, M_3)$, and does not modify the block assigned to another variable.*

Proof of Lemma 7. Firstly, because the allocator is correct, it always affects an unused block to a variable. This means that it does not exists $\texttt{v} \neq \texttt{w}$ such as $E_2(\texttt{v}) = E_2(\texttt{w})$. Then, assigning a new value to $E_2(\texttt{v})$ when computing M_3 changes only the value associated to v in $(M_3 \circ E_2)$. So $(M_3 \circ E_2)$ differs from $(M_2 \circ E_2)$ only by the value associated to v.

Secondly, variable v is not declared in C_1, then the modification of the value associate to v in $(M_2 \circ E_2)$ when computing M_3 keeps $C_1 \subseteq (E_2, M_3)$. □

Lemma 8 (Code generation is transparent). *The code generation for all term t is transparent.*

Proof of Lemma 8. Theorem 8 can be proved by induction over term structures. We only detail the proof for terms of the form `\sum(t1,t2,\lambda x;t3)`.

Let us suppose that Lemma 8 is true for all the subterms of `\sum(t1,t2,\lambda x;t3)` and let us prove that under this hypothesis the code generated for `\sum(t1,t2,\lambda x;t3)` is transparent.

Let Ctx be a code generation context, $C_1 = (E_1, M_1)$ and $C_2 = (E_2, M_2)$ be semantic contexts such that the declarations in $[\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.decl$
and $init\_var([\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.decl)$ have been done in C_2 and $C_1 \subseteq C_2$, and V be the union between the set of variables in $[\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.decl$ and $codomain(Ctx)$.

Prove that if $E, M_2 \models_s [\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.code \Rightarrow M_t$,
then $(E_1 - \{(v, E_1(v)) \mid v \in V\}, M_1 - \{(E_1(v), M_1(E_1(v)) \mid v \in V\}) \subseteq (E_2, M_t)$.

Since $C_1 \subseteq C_2$ obviously $(E_1 - \{(v, E_1(v)) \mid v \in V\}, M_1 - \{(E_1(v), M_1(E_1(v)) \mid v \in V\}) \subseteq (E_2, M_2)$. We will now prove that each statement of $[\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)}, Ctx]\!]_{tr}.code$ maintains this property (see Fig. 18 for the translation schema).

$[\![$t1, $Ctx]\!]_{tr}.code$, $[\![$t2, $Ctx]\!]_{tr}.code$ and $[\![$t3, $Ctx[\texttt{x} \rightarrow \texttt{k}]]\!]_{tr}.code$:

Since the variables in $[\![t1]\!]_{tr}.decl$ or in $[\![t1]\!]_{tr}.decl$ are included in V and Lemma 8 is true for t1 and for t2,
if $E, M_2 \models_s [\![\text{tx},\ Ctx]\!]_{tr}.code \Rightarrow M_t$,
then $(E_1 - \{(v, E_1(v)) \mid v \in V\}, M_1 - \{(E_1(v), M_1(E_1(v)) \mid v \in V\}) \subseteq (E_2, M_t)$ for tx equal to t1 or to t2.

***int_assignment*(one, 1) and *int_assignment*(sum, 0):**
***var_assignment*(k, $[\![$t1, *Ctx*$]\!]_{tr}$*.res*) , *var_assignment*(max, $[\![$t2, *Ctx*$]\!]_{tr}$*.res*)**
and *var_assignment*(lbda, $[\![$t3, *Ctx*[x$\rightarrow$ k]$]\!]_{tr}$*.res*):]
***addition_assignment*(sum, lbda) and *addition_assignment*(k, one):**
By Lemma 7, each macro ...*_assignment* only modifies the memory at the block associated to its left-value (i.e., its first parameter, like one, k, max, etc.). Furthermore, this left-value (say, one) is included in V so that if $E, M_2 \models_s int\_assignment(\texttt{one}, 1) \Rightarrow M_t$,
then $(E_1 - \{(v, E_1(v)) \mid v \in V\}, M_1 - \{(E_1(v), M_1(E_1(v)) \mid v \in V\}) \subseteq (E_2, M_t)$.

So, each statement of the code generated for \sum(t1,t2,\lambda x;t3) maintain the transparent, so the code generation for \sum(t1,t2,\lambda x;t3) is transparent. □

Lemma 6 (addition_assignment(sum, lbda) cannot overflow). *Assume terms* t1*,* t2*,* t3 *are correctly translated (recurrence hypothesis in Lemma 9); and $C = (E, M)$ is the current context when computing the semantics of addition_assignment(*sum*,* lbda*) in the code generated for* \sum(t1,t2,\lambda x;t3) *(see Fig. 18)*
i.e.: $(M \circ E)(\texttt{sum}) = \mathcal{T}(\texttt{sum})(\sum_{q=min}^{i-1} lambda_q)$ *and* $(M \circ E)(\texttt{lbda}) = \mathcal{T}(\texttt{sum})(lambda_i)$
with $i \leq max$ (see step **addition_assignment(sum, lbda)** *in the proof of Lemma 9 in Appendix E) then,* $min\_type(\mathcal{T}(\texttt{sum})) \dot{\leq} \sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \dot{\leq} max\_type(\mathcal{T}(\texttt{sum}))$.

This means that not only the result of \sum(t1,t2,\lambda x;t3) fits in the inferred type (Theorem 2), but every step of the computation fits in that type.

Proof of Lemma 6. Proof by cases:

Case $\mathcal{T}$(sum) = mpz: Since $min\_type(\texttt{mpz}) = -\infty$ and $max\_type(\texttt{mpz}) = +\infty$, Lemma 6 is obviously true.

Case $\mathcal{T}$(sum) = int: The interval inference system which is proved to be correct have inferred for the term \sum(t1,t2,\lambda x;t3) an interval $[l_{sum}, u_{sum}]$ such as $\Theta([l_{sum}, u_{sum}]) = \texttt{int}$ (because $\mathcal{T}(\texttt{sum}) = \texttt{int}$).
If $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \in [l_{sum}, u_{sum}]$
then, since Θ is correct $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \in [min\_int, max\_int]$,
otherwise there are two possibilities (let $[l_3, u_3]$ be the interval inferred by the interval inference system]):

- $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i > u_{sum}$,
  - If $u_3 > 0$ (i.e., $0 \leq l_{sum} \leq u_{sum}$,) then possibly $\sum_{q=min}^{max} lambda_q > u_{sum}$ because it remains to add all the possibly positive $lambda_q$ for $q \in [i+1, max]$. However, $\sum_{q=min}^{max} lambda_q > u_{sum}$ is not possible, because of Theorem 1 on interval inference.
  - Otherwise, $u_3 \leq 0$,
    then according to the interval inference system $u_{sum} \leq 0$
    then $u_{sum} \dot{\leq} \sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \dot{\leq} 0$ so $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i$ fits in an int (in short, the computation of \sum(t1,t2,\lambda x;t3) starts out of $[l_{sum}, u_{sum}]$, because it starts at 0, and proceeds towards a final result in this interval, but in any case it remains in $[min\_int, max\_int]$).
- $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i < l_{sum}$ is symmetric to the previous case.

So, even if $\sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \notin [l_{sum}, u_{sum}]$ then $min\_int \leq \sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \leq max\_int$.

So $min\_type(\mathcal{T}(\texttt{sum})) \dot{\leq} \sum_{q=min}^{i-1} lambda_q \dot{+} lambda_i \dot{\leq} max\_type(\mathcal{T}(\texttt{sum}))$. □

Lemma 9 (Code generation is sound). *Code generation for all term t is sound.*

$[\![$`\sum(t1,t2,\lambda x;t3)`$,\ Ctx]\!]_{tr}.decl =$
```
    t_k one; t_k k; t_k max;
    t_sum sum; t_sum lbda;
    [[t1, Ctx]]_tr.decl; [[t2, Ctx]]_tr.decl
```

$[\![$`\sum(t1,t2,\lambda x;t3)`$,\ Ctx]\!]_{tr}.code =$
```
    [[t1, Ctx]]_tr.code; [[t2, Ctx]]_tr.code;
    int_assignment(one, 1);
    var_assignment(k, [[t1, Ctx]]_tr.res);
    var_assignment(max, [[t2, Ctx]]_tr.res);
    int_assignment(sum, 0);
    while (condition(k,max)) {
        [[t3, Ctx[x →k]]]_tr.code;
        var_assignment(lbda, [[t3, Ctx[x →k]]]_tr.res);
        addition_assignment(sum, lbda);
        addition_assignment(k, one);
    }
```

$[\![$`\sum(t1,t2,\lambda x;t3)`$,\ Ctx]\!]_{tr}.res =$ sum

Fig. 18. Translation of `\sum(t1,t2,\lambda x;t3)`

Proof of Lemma 9. By induction over term structure and Lemma 8. We only detail the proof for terms of the form `\sum(t1,t2,\lambda x;t3)`. We assume that Lemmas 9 and 8 are true for all the subterms of `\sum(t1,t2,\lambda x;t3)` (i.e., t1, t2 and t3) and prove that under this hypothesis code generation for `\sum(t1,t2,\lambda x;t3)` is sound.

Let Ctx be a code generation context, Λ be a logical context, $\widehat{C} = (\widehat{E}, \widehat{M_1})$ and $C = (E, M_1)$ be two semantic contexts, where the declaration in $[\![t,\ Ctx]\!]_{tr}.decl$ and $init\_var([\![t,\ Ctx]\!]_{tr}.decl)$ have been done in C, and such as $\widehat{C} \sqsubseteq C$ and $\Lambda = unwrap \circ (M_1 \circ E) \circ Ctx$.

Recall that

$$\frac{E, M_1, \Lambda \models_t \texttt{t1} \Rightarrow inf \qquad E, M_1, \Lambda \models_t \texttt{t2} \Rightarrow sup \qquad \forall k \in [min, max](E, M_1, \Lambda[\texttt{x} \rightarrow k] \models_t \texttt{t3} \Rightarrow lambda_k)}{E, M_1, \Lambda \models_t \texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)} \Rightarrow \sum_{k=min}^{max} lambda_k}$$

Prove that
$E, M_1 \models_s [\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.code \Rightarrow M_{sum}$,
where $M_{sum}(E([\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.res)) =$
$\mathcal{T}([\![\texttt{\textbackslash sum(t1,t2,\textbackslash lambda x;t3)},\ Ctx]\!]_{tr}.res)(\sum_{k=min}^{max} lambda_k)$

To prove these properties, we apply the semantics rules, statement after statement of the generated code, starting in the semantic context C (see Fig. 18 for the translation schema).

At each step, we highlight why the rules can be applied, and properties of the semantic contexts that are useful for the proof.

$[\![\texttt{t1},\ Ctx]\!]_{tr}.code$:

assume $E, M_1 \models [\![\texttt{t1},\ Ctx]\!]_{tr}.code \Rightarrow M_2$

then $(M_2 \circ E)([\![\texttt{t1},\ Ctx]\!]_{tr}.res) = \mathcal{T}([\![\texttt{t1},\ Ctx]\!]_{tr}.res)(min)$ because code generated for t1 is transparent and sound.

$[\![\texttt{t2}, \mathit{Ctx}]\!]_{tr}.\mathit{code}$:
assume $E, M_2 \models [\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{code} \Rightarrow M_3$
then $(M_3 \circ E)([\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) = \mathcal{T}([\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res})(\mathit{max})$ and $(M_3 \circ E)([\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) = \mathcal{T}([\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res})(\mathit{min})$ because code generated for t2 is transparent and sound.
***int_assignment*(one, 1):**
because

$$\frac{\mathit{min\_int} \leq \mathit{valof}(1) \leq \mathit{max\_int} \qquad M_4 = M_3[E(\texttt{one}) \rightarrow \mathcal{T}(\texttt{one})(\mathit{valof}(1))]}{E, M_3 \models_m \mathit{int\_assignment}(\texttt{one}, 1) \Rightarrow M_4}$$

and obviously, $\mathit{min\_int} \leq \mathit{valof}(1) \leq \mathit{max\_int}$
we have $E, M_3 \models_m \mathit{int\_assignment}(\texttt{one}, 1) \Rightarrow M_4$ and $M_4 = M_3[E(\texttt{one}) \rightarrow \mathcal{T}(\texttt{one})(\mathit{valof}(1))$
and because of Lemma 7, the values of t1 and t2 are still available in M_4.
***var_assignment*(k, $[\![\texttt{t1}, \mathit{Ctx}]\!]_{tr}.\mathit{res}$):**
because

$$\frac{(M_4 \circ E)([\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) = \mathcal{T}([\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res})(\mathit{min}) \qquad M_5 = M_4[E(\texttt{k}) \rightarrow \mathcal{T}(\texttt{k})(\mathit{min})]}{E, M_4 \models_m \mathit{var\_assignment}(\texttt{k}, [\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) \Rightarrow M_5}$$

we have $E, M_4 \models_m \mathit{var\_assignment}(\texttt{k}, [\![\texttt{t1}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) \Rightarrow M_5$ and $M_5 = M_4[E(\texttt{k}) \rightarrow \mathcal{T}(\texttt{k})(\mathit{min})]$,
and because of Lemma 7, the values of t1, t2 and one are still available in M_5.
Finally, *var_assignment* is well typed because $[\![\texttt{t1}, \mathit{Ctx}]\!]_{tr}.\mathit{decl}$ tells the type of k (t_k) is the largest of the type of t1 or of t2+1, so all possible values of $[\![\texttt{t}, \ \mathit{Ctx}]\!]_{t1}.\mathit{res}$ can fit in k.
In the following, we note *min* for the value of k in the current memory.
***var_assignment*(max, $[\![\texttt{t2}, \mathit{Ctx}]\!]_{tr}.\mathit{res}$):**
because

$$\frac{(M_5 \circ E)([\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) = \mathcal{T}([\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res})(\mathit{max}) \qquad M_6 = M_5[E(\texttt{max}) \rightarrow \mathcal{T}(\texttt{max})(\mathit{max})]}{E, M_5 \models_m \mathit{var\_assignment}(\texttt{max}, [\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) \Rightarrow M_6}$$

we have $M_5 \models_m \mathit{var\_assignment}(\texttt{max}, [\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{res}) \Rightarrow M_6$ and $M_6 = M_5[E(\texttt{max}) \rightarrow \mathcal{T}(\texttt{max})(\mathit{max})]$,
and because of Lemma 7, the values of t1, t2, one and k are still available in M_6.
Finally, *var_assignment* is well typed because $[\![\texttt{t2}, \ \mathit{Ctx}]\!]_{tr}.\mathit{decl}$ tells the type of max (t_k) is the largest of the type of t1 or of t2+1, so all possible values of $[\![\texttt{t}, \ \mathit{Ctx}]\!]_{t1}.\mathit{res}$ can fit in max.
In the following, we note *max* for the value of max in the current memory.
***int_assignment*(sum, 0):**
because

$$\frac{\mathit{min\_int} \dot{\leq} 0 \dot{\leq} \mathit{max\_int} \qquad M_7 = M_6[E(\texttt{sum}) \rightarrow \mathcal{T}(\texttt{sum})(0)]}{E, M_6 \models_m \mathit{int\_assignment}(\texttt{sum}, \texttt{0}) \Rightarrow M_7}$$

we have $M_6 \models_m \mathit{int\_assignment}(\texttt{sum}, \texttt{0}) \Rightarrow M_7$ and $M_7 = M_6[E(\texttt{sum}) \rightarrow \mathcal{T}(\texttt{sum})(0)]$,
and because of Lemma 7, the values of t1, t2, k and max are still available in M_6.
Finally $(M_7 \circ E)(\texttt{k}) = \texttt{t\_k}(\mathit{min}) = \texttt{t\_k}(0)$ and $(M_7 \circ E)(\texttt{sum}) = \texttt{t\_sum}(0) = \texttt{t\_sum}(\sum_{q=min}^{min-1} \mathit{lambda}_q)$.

Let **B** be the body of the loop. Let $C = (E, M)$ be a semantic context such as $E, M \models_s \mathbf{B} \Rightarrow M'$. We prove that for all i such that $(M \circ E)(\texttt{k}) = \texttt{t\_k}(i)$ and $(M \circ E)(\texttt{sum}) = \texttt{t\_sum}(\sum_{q=min}^{i-1} \mathit{lambda}_q)$ then $M' \circ E(\texttt{k}) = \texttt{t\_k}(i+1)$ and $M' \circ E(\texttt{sum}) = \texttt{t\_sum}(\sum_{q=min}^{i} \mathit{lambda}_q)$.
Therefore, $(M \circ E)(\texttt{k}) = \texttt{t\_k}(i)$ and $(M \circ E)(\texttt{sum}) = \texttt{t\_sum}(\sum_{q=min}^{unwrap((M \circ E)(\texttt{k}))-1} \mathit{lambda}_q)$ is the

invariant of the loop, and we observe it is satisfied in M_7 (last remark in step ***int_assignment*(`sum`, `0`)**).

As above, we apply the semantic rules statements after statements, starting from a memory $M_1^{\mathbf{B}}$ that is supposed to satisfy the invariant (e.g., memory M_7).

$[\![$**t3,** $\boldsymbol{Ctx}$**[x→k]**$]\!]_{tr}$***.code*** **:**

assume $E, M_1^{\mathbf{B}} \models [\![\mathtt{t3}, Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.code \Rightarrow M_2^{\mathbf{B}}$
then $(M_2^{\mathbf{B}} \circ E)(\mathtt{k}) = \mathtt{t\_k}(i)$ and $(M_2^{\mathbf{B}} \circ E)([\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res) = \mathcal{T}([\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res)(lambda_i)$,
and because the code generated for t2 is assumed to be transparent and sound, the values of t1, t2, k, max and sum are still available in $M_2^{\mathbf{B}}$.

***var_assignment*(`lbda`, $[\![$ t3, *Ctx*[x→ k]$]\!]_{tr}$*.res*):**

because

$$\frac{\begin{array}{c}(M_2^{\mathbf{B}} \circ E)([\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res) = \mathcal{T}([\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res)(lambda_1)\\ M_3^{\mathbf{B}} = M_2^{\mathbf{B}}[b \to \mathcal{T}(\mathtt{lbda})(lambda_i)]\end{array}}{E, M_2^{\mathbf{B}} \models_s var\_assignment(\mathtt{lbda}, [\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res) \Rightarrow M_3^{\mathbf{B}}}$$

we have $M_2^{\mathbf{B}} \models_s var\_assignment(\mathtt{lbda}, [\![\mathtt{t3},\ Ctx[\mathtt{x} \to \mathtt{k}]]\!]_{tr}.res) \Rightarrow M_3^{\mathbf{B}}$
and $M_3^{\mathbf{B}} = M_2^{\mathbf{B}}[b \to \mathcal{T}(\mathtt{lbda})(lambda_i)]$,
and because of Lemma 7 the values of t1, t2, k, max and sum are still available in $M_3^{\mathbf{B}}$. In particular, $(M_3^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q)$.

***addition_assignment*(`sum`, `lbda`):**

because $\mathcal{T}(\mathtt{lbda}) = \mathcal{T}(\mathtt{sum}) = \mathtt{t\_sum}$ and $(M_3^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q)$ (last remark of previous item), and the semantics of *addition_assignment*,
we have

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{lbda}) = \mathcal{T}(\mathtt{sum})\\ (M_3^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathcal{T}(\mathtt{sum})(\displaystyle\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q) \qquad (M_3^{\mathbf{B}} \circ E)(\mathtt{lbda}) = \mathcal{T}(\mathtt{sum})(lambda_i)\\ min\_type(\mathcal{T}(\mathtt{sum})) \dot{\leq} \displaystyle\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q \dot{+} lambda_i \dot{\leq} max\_type(\mathcal{T}(\mathtt{sum}))\\ M_4^{\mathbf{B}} = M_3^{\mathbf{B}}[E(\mathtt{k}) \to \mathcal{T}(\mathtt{sum})(\displaystyle\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))} lambda_q)]\end{array}}{E, M_3^{\mathbf{B}} \models_m addition\_assignment(\mathtt{sum}, \mathtt{lbda}) \Rightarrow M_4^{\mathbf{B}}}$$

Furthermore, $\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q \dot{+} lambda_i = \sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))} lambda_q$ and $min\_type(\mathcal{T}(\mathtt{sum})) \dot{\leq} \sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q \dot{+} lambda_i \dot{\leq} max\_type(\mathcal{T}(\mathtt{sum}))$ according to Lemma 6.

Therefore $(M_4^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_3^{\mathbf{B}} \circ E)(\mathtt{k}))} lambda_q)$.

***addition_assignment*(k, one):**

because $\mathcal{T}(\mathtt{one}) = \mathcal{T}(\mathtt{k}) = \mathtt{t\_k}$
and

$$\frac{\begin{array}{c}\mathcal{T}(\mathtt{one}) = \mathcal{T}(\mathtt{k}) \qquad (M_4^{\mathbf{B}} \circ E)(\mathtt{k})) = \mathcal{T}(\mathtt{k})(i) \qquad (M_4^{\mathbf{B}} \circ E)(\mathtt{one})) = \mathcal{T}(\mathtt{k})(1)\\ min\_type(\mathcal{T}(\mathtt{one})) \dot{\leq} i \dot{+} 1 \dot{\leq} max\_type(\mathcal{T}(\mathtt{one})) \qquad M_5^{\mathbf{B}} = M_4^{\mathbf{B}}[E(\mathtt{k}) \to \mathcal{T}(\mathtt{k})(i \dot{+} 1)]\end{array}}{E, M_4^{\mathbf{B}} \models_s addition\_assignment(\mathtt{k}, \mathtt{one}) \Rightarrow M_5^{\mathbf{B}}}$$

we have $(M_5^{\mathbf{B}} \circ E)(\mathtt{k}) = \mathtt{t\_k}(unwrap((M_4^{\mathbf{B}} \circ E)(\mathtt{k})) \dot{+} 1)$
and $(M_5^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_4^{\mathbf{B}} \circ E)(\mathtt{k}))} lambda_q) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_5^{\mathbf{B}} \circ E)(\mathtt{k}))-1} lambda_q)$.
Note that this incrementation does not overflow $(min\_type(\mathcal{T}(\mathtt{one})) \dot{\leq} i \dot{+} 1 \dot{\leq} max\_type(\mathcal{T}(\mathtt{one})))$ because the inferred type, `t_k`, has been chosen to contain at least the value of t2 + 1 (see last paragraph in Section 3.2).

The loop body satisfies the invariant (see last remark of item ***addition_assignment*(k, one)** above). Furthermore, the loop stops when the condition is true, i.e., $condition(\mathtt{k}, \mathtt{max})$.
It is true when $unwrap((M_5^{\mathbf{B}} \circ E)(\mathtt{k})) \dot{>} (M_5^{\mathbf{B}} \circ E)(\mathtt{max})$.
Therefore, when the loop exits
we have $(M_5^{\mathbf{B}} \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M_5^{\mathbf{B}} \circ E)(\mathtt{max}))} lambda_q)$.

Finally:

- Either $min \dot{\leq} max$ and the loop computes a memory M where $M \circ E(\mathtt{k}) = \mathtt{t\_k}(max \dot{+} 1)$ and $(M \circ E)(\mathtt{sum}) = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M \circ E)(\mathtt{max}))} lambda_q)$.
- or $min \dot{>} max$ and the loop computes a memory M where $M \circ E(\mathtt{k}) = \mathtt{t\_k}(min)$ and $(M \circ E)(\mathtt{sum}) = 0 = \mathtt{t\_sum}(\sum_{q=min}^{unwrap((M \circ E)(\mathtt{max}))} lambda_q)$.

In the two cases, we have $E, M_1 \models_s [\![\mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)}, Ctx]\!]_{tr}.code \Rightarrow M_{sum}$, and $M_{sum}(E([\![\mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)}, Ctx]\!]_{tr}.res)) =$
$\mathcal{T}([\![\mathtt{\backslash sum(t1,t2,\backslash lambda\ x;t3)}, Ctx]\!]_{tr}.res)(\sum_{k=min}^{max} lambda_k)$ and $\widehat{C} \subseteq (E, M_{sum})$
Then Lemma 9 is true. □

THEOREM 3. *The code generation for all term t is sound and transparent.*

PROOF OF THEOREM 3. By Lemmas 8 and 9. □