Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates

Thibaut Benjamin, Julien Signoles

► **To cite this version:**

Thibaut Benjamin, Julien Signoles. Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates. SAC/SVT (Symposium on Applied Computing / Software Verification and Testing Track), Mar 2023, Tallinn, Estonia. https://doi.org/10.1145/3555776.3577617, 10.1145/3555776.3577617 . cea-04176678

HAL Id: cea-04176678

https://cea.hal.science/cea-04176678

Submitted on 3 Aug 2023

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

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

Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates

Thibaut Benjamin
Université Paris-Saclay, CEA, List
Palaiseau, France
thibaut.benjamin@gmail.com

Julien Signoles
Université Paris-Saclay, CEA, List
Palaiseau, France
julien.signoles@cea.fr

ABSTRACT

Runtime Assertion Checking (RAC) is a lightweight formal method that verifies formal code annotations, typically assertions, at runtime. The main RAC challenge consists in generating code that is both sound and efficient for checking expressive properties. In particular, checking formal arithmetic properties usually requires to use machine integer arithmetic to be efficient, but needs to rely on an exact yet slower arithmetic library to be sound.

This paper formalizes an efficient RAC tool for arithmetic properties, which may include user-defined functions and predicates. Efficient code generation for these routines is based on specialization, allowing to generate efficient functions using machine arithmetic when possible, or slower functions relying on exact arithmetic, according to the calling context. This formalization is implemented in E-ACSL, a runtime assertion checker for C programs.

CCS CONCEPTS

• **Software and its engineering** → *Dynamic analysis*; **Source code generation**; **Semantics**; *Specification languages*;

ACM Reference Format:
Thibaut Benjamin and Julien Signoles. 2023. Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-April 2, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3555776.3577617

1 INTRODUCTION

Runtime Assertion Checking (shortly, RAC) is a lightweight formal method that verifies formal code annotations —typically, assertions written in a formal behavioral specification language (shortly, BISL)— at runtime, during concrete program executions [11]. Associated to a testing technique (e.g., unit testing, fuzzing, etc), it is a powerful way to detect safety and/or security bugs that would remain unobservable by testing only. To do so, RAC tools usually generate executable code (or bytecode) from formal annotations, either directly during the compilation process, or indirectly by generating source code which is in turn compiled into executable code (or bytecode) by a standard compiler. Even though RAC is about as old as the other formal methods, it has not been as extensively studied from a theoretical point of view [26]. For instance, the authors of Spec#, a formal specification language for both RAC and deductive verification of C# programs write that their "run-time checker is straightforward" without more details, but also indicate that its "run-time overhead is prohibitive" [3]. Here is indeed one of the key challenges for RAC: generating code that verifies the annotations at runtime, both *soundly*, i.e. by reporting the correct validity verdicts, and *efficiently*, i.e. by limiting the time and memory overheads.

The goal of this paper consists in formalizing the core mechanism of E-ACSL [27], a RAC tool for C programs, based on Frama-C [5] for generating monitor that translates arithmetic properties expressed in the E-ACSL specification language [12] to C code. We prove that the generated code is sound (i.e. emits an alert as soon as one annotation is invalid), transparent (i.e. preserves the functional behavior of the program when all the annotations are valid) and efficient. Translating such properties in that way is challenging, because modern BISLs rely on mathematical arithmetic. Therefore, naively translating a formal annotation `/*@ assert x+1 == 0;*/` into a code assertion `assert(x+1 == 0);` is possibly unsound because the former is computed over the mathematical integers $\mathbb{Z}$ (assuming `x` is of any integer type), while the latter relies on bounded machine integers (such as `int`), which may overflow. Soundness thus requires to rely on an exact arithmetic library when generating the code, such as the GMP library[1] in C. In that case, the translation is a block of 16 C statements that dynamically allocates memory blocks, calls library functions, and deallocates memory blocks consistently. Executing such a piece of code is very inefficient compared to computing a single machine arithmetic operation. To reconcile both soundness and efficiency of the generated code, we rely on a dedicated type system similar to the one presented in [16] that allows the generator to use efficient machine integers when possible and sound arithmetic integers otherwise. This paper formalizes this code generation step for an arithmetic specification language over a representative C-like programming language.

Our formal specification language also includes (possibly recursive) user-defined logic functions and predicates. For better efficiency, the generated code for logic functions and predicates is specialized according to the call site, which may lead to generating several C routines from a logic routine. For instance, a logic function from $\mathbb{Z}$ to $\mathbb{Z}$ called twice may be specialized into a C function from `int` to `int` called at the first call site, and into another C function from `mpz` to `mpz` called at the second call site, `mpz` being the type of GMP integers. While E-ACSL has documented support for generating expressions in machine integers and GMP [16], the support for logic functions is new.To sum up, this paper presents

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

SAC '23, March 27-April 2, 2023, Tallinn, Estonia
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577617

[1] https://gmplib.org/

a **formalization of a fast runtime assertion checker** for **properties over mathematical integers**, which relies on **function specialization** [2] to efficiently deal with logic functions and predicate calls. It also **proves the correctness** of the translation.

Related Work. Y. Cheon [10] was the first to formally study RAC, in the context of JML [17], a BISL for Java. He did not focus his work on integer arithmetic since, at that time, the JML's arithmetic was exactly the Java's machine arithmetic: the translation function for arithmetic was the identity function. However, we introduce a notion of macro, which is close to his notion of context used for dealing with undefined constructs such as 1/0. Later, H. Lehner [18] formalized in the Coq proof assistant [7] a large subset of the JML's semantics. He also formalized a RAC algorithm for the JML's `assignable` clause, which is independent from, but compatible with, our integer properties. More recently, J. C. Filliâtre and C. Pascutto [13] proposed Ortac, a RAC tool for OCaml. It relies on a similar mechanism than ours for generating efficient arithmetic code, but without details nor formalization for that part. They also do not deal with user-defined logic functions and predicates.

Several works focused on RAC of C programs. We already mentioned E-ACSL's type system [16], which we rely on. This paper does not study the code generation process, stating that "Generating code from the information computed by the type system is quite straightforward". While it is true for the scope of that article, the code generation becomes an intricate problem when handling user-defined logic functions and predicates, and avoiding a combinatorial explosion to prove the translation. G. Petiot et al [23] were the first to formalize RAC for an arithmetic language like ours. However, they did not study how to generate efficient machine arithmetic when possible and did not deal with user-defined logic functions and predicates. D. Ly et al [19] also studied formal RAC, but focused on memory properties. In particular, they only considered machine arithmetic. Our formalization is complementary to theirs, since many practical properties are a combination of memory and arithmetic constructs.

A substantial part of this work relies on the GMP library. We use it as a black box and only specify the needed functions. A large part of this library, containing all the functions we use, has been formally proved with Why3 [21]. We also rely on function specialization [2] for generating efficient function calls. It is a compilation technique often used in optimizing compilers, but which has never been used for RAC as far as we know. This technique has been formalized in Coq for a JIT compiler [4], while related techniques have been studied by S. Blazy and P. Facon [9] for program specialization [20], and by L. Andersen [1] for partial evaluation.

Sec. 2 introduces a running example, while Sec. 3 presents our input programming and specification languages, as well as the output language which the code is generated to. Sec. 4 details the formal translation for the arithmetic part of the language, while Sec. 5 extends it to logic functions and predicates. Finally, Sec. 6 presents the correctness properties that we prove about the translation, and Sec. 7 presents a few implementation details, before concluding and discussing future work in Sec. 8.

```c
/*@ logic integer mean (integer x, integer y) = (x + y) / 2; */

int mean_implem (int a, int b){
  if (a < b) { return a + (b - a) / 2; }
  else{ return b + (a - b) / 2; }
}
void main() (int a, int b){
  /*@ assert mean_implem(5,7) == mean(5,7); */
  /*@ assert mean_implem(16000,24000) == mean(10000,60000); */
}
```

(a) An annotated C program.

```c
1  int mean_1 (int x, int y) { return (x + y) / 2; }
2
3  int mean_2 (int x, int y) {
4    mpz x1, y1, res1, res2, two; int res;
5    mpz_init(x1); mpz_init(y1);
6    mpz_init(res1); mpz_init(res2); mpz_init(two);
7    mpz_set_int(x1, x); mpz_set_int(y1, y); mpz_set_int(two, 2);
8    mpz_add(res1, x1, y1);
9    mpz_div(res2, res1, two);
10   res = mpz_get_int(res2);
11   mpz_clear(x1); mpz_clear(y1);
12   mpz_clear(res1); mpz_clear(res2); mpz_clear(two);
13   return res;
14 }
15
16 int mean_implem (int a, int b) {
17   if (a < b) { return a + (b - a) / 2; }
18   else { return b + (a - b) / 2; }
19 }
20 void main() (int a, int b) {
21   assert(mean_implem(5,7) == mean_1(5,7));
22   assert(mean_implem(10000, 60000) == mean_2(10000, 60000))
23 }
```

(b) Its monitored counterpart.

Figure 1: An annotated C program and its translation.

2 RUNNING EXAMPLE

This article presents a translation mechanism to generate correct monitors for RAC of C programs. It takes as input a C program with formal annotations and generates a new C program that checks the annotations at runtime. Fig. 1a presents such a program. The annotations are enclosed in special comments starting with `@`. This example contains both an implementation `mean_implem`, which takes care of never overflowing, and a mathematical definition `mean` of the mean function. We use runtime assertion to check whether the implementation complies with the mathematical definition on a pair of examples provided as assertions. Fig. 1b shows the C code that our translation generates to monitor these assertions.

This example shows an important feature of the translation: while the semantics of the annotations relies on $\mathbb{Z}$, the set of (mathematical) integers, the semantics of C relies on machine integers, which may overflow. This semantics is shared by all modern BISLs. To interpret mathematical integers soundly, we use the GMP library that provides exact integers through the type `mpz`. This lets us detect the invalid assertion during the second function call. This code requires allocating (e.g. at lines 5 and 6), initializing (e.g. at line 7), and deallocating (e.g. at line 11 and 12) the generated `mpz` variables. Even though there is a single logic function in the input program, we generate two different specializations: `mean_1` when the computation cannot overflow, and `mean_2` when the computation must happen in the type `mpz` (as is the case in our example, assuming a 16-bit architecture where the maximal value for an **`int`** is 65535). More naive approaches can be considered, such as always using type `mpz` or always inlining all function calls, but they are not suitable in practice: the former is not efficient enough, while the latter is unusable with recursive functions.

3 LANGUAGE DEFINITIONS

Formalizing a runtime assertion checker for the whole C programming language would be too large for our study. We restrict it

$$
\begin{array}{lcll}
p & ::= & d^* \, f^* & \text{annotated program} \\
d & ::= & \tau_c \ \mathrm{id} & \text{program declaration} \\
f & ::= & \tau_c \ \mathrm{id}(d^*)\{d^*; s_c\} & \text{program function} \\
 & | & \texttt{/*@ logic}\ \kappa\ \mathrm{id}(\delta^*) = t & \text{logic function} \\
 & | & \texttt{/*@ predicate}\ \mathrm{id}(\delta^*) = p & \text{predicate} \\
s_c & ::= & \texttt{skip;} & \text{empty statement} \\
 & | & \mathrm{id}\ =\ e; & \text{assignment} \\
 & | & \mathrm{id}\ =\ \mathrm{id}(e^*); & \text{function call} \\
 & | & \mathrm{id}(e^*); & \text{procedure call} \\
 & | & s\ s & \text{sequence} \\
 & | & \texttt{if}(e)\ s\ \texttt{else}\ s & \text{conditional} \\
 & | & \texttt{while}(e)\ s & \text{loop} \\
 & | & \texttt{assert}(e); & \text{program assertion} \\
 & | & \texttt{/*@ assert}\ p\ \texttt{*/} & \text{logic assertion} \\
 & | & \texttt{return}(e); & \text{return statement} \\
e & ::= & z_m & \text{machine integer} \\
 & | & \mathrm{id} & \text{variable access} \\
 & | & e \mathbin{\square} e & \square \in \{\texttt{+;-;*;/}\} \\
 & | & e \mathbin{\text{◖}} e & \text{◖} \in \{\texttt{<;<=;>;>=;==;!=}\} \\
\tau_c & ::= & \texttt{int} \mid \texttt{void} & \text{program types}
\end{array}
$$

$$
\begin{array}{lcll}
\delta & ::= & \tau\ \mathrm{id} & \text{logic declaration} \\
p & ::= & \texttt{\textbackslash true} \mid \texttt{\textbackslash false} & \text{truth values} \\
 & | & t \blacktriangleleft t & \blacktriangleleft \in \{<; \leq; >; \geq; \overset{?}{=}; \neq\} \\
 & | & !\ t & \text{negation} \\
 & | & p \mid\mid p & \text{disjunction} \\
 & | & \mathrm{id}(\delta^*) & \text{predicate call}
\end{array}
\qquad
\begin{array}{lcll}
t & ::= & z & \text{integer in } \mathbb{Z} \\
 & | & \mathrm{id} & \text{variable access} \\
 & | & t \diamond t & \diamond \in \{+; -; \times; /\} \\
 & | & p\ ?\ t\ :\ t & \text{conditional term} \\
 & | & \mathrm{id}(\delta^*) & \text{function call} \\
\kappa & ::= & \texttt{int} \mid \texttt{integer} & \text{logic types}
\end{array}
$$

Figure 2: Syntax of mini-C (above) and mini-FSL (below).

to a smaller core language with arithmetic assertions. We denote $f : X \rightharpoonup Y$ partial functions, and dom f the subset of X on which f is defined. For $x \in X$, and $y \in Y$, $f\{x \backslash y\}$ is the partial function that coincides with f at every point except in x, which is mapped to y. The partial function defined nowhere is noted $\bot$, while the type for a list of elements of type A is denoted A^*.

3.1 Syntax

Fig. 2 presents the syntax of our programming language, named mini-C together with the syntax of our specification language, named mini-FSL (for Formal Specification Language). The languages are mutually dependent. mini-C programs are sequences of variable declarations followed by routine definitions, a routine being a mini-C function or procedure, or else a mini-FSL logic function or predicate. The body of each (program) function contains a sequence of statements, including standard control flow structures, function calls with a particular case for procedures, which return no values, and both program and logic assertions. The expressions are usual but all of type `int` for simplicity. User-defined logic functions and predicates respectively have a term and a predicate as body. They may themselves contain (possibly recursive) function and predicate calls. Importantly, terms are either of type `int` (machine integer) or `integer` (mathematical integer), the former being a subtype of the latter. Logic arithmetic operators $\diamond$ are over integers.

The code generated by the runtime assertion checker may call GMP functions. We extend the language mini-C to mini-GMP, introduced in Fig. 3: it provides built-in calls to the GMP API. For the sake of simplicity, we shorten the names of these functions, in practice one should write them as in Fig. 1. The only built-in function not self-explanatory is `set_s`: It assigns to a `mpz` variable the value of an integer represented as a string (hence the type `char *` in the syntax). It allows for assignments from a constant integer that does not fit in `int`, without requiring to allocate a `mpz`.

3.2 Program Structure

From now on, we assume the input program to be well-formed. Specifically, all variables are declared before being used, all functions are defined before being called, and programs are well typed.

$$
\begin{array}{lcllcll}
s & ::= & s_c \mid s_g & \text{statement extension} & & & \\
\tau & ::= & \tau_c \mid \texttt{char *} \mid \texttt{mpz} & \text{type extension} & | & \texttt{set\_z}(\mathrm{id}, \mathrm{id}) & \text{assignment from a } \texttt{mpz} \\
s_g & ::= & \texttt{init}(\mathrm{id}) & \texttt{mpz}\text{ allocation} & | & \texttt{cl}(\mathrm{id}) & \texttt{mpz}\text{ de-allocation} \\
 & | & \texttt{set\_i}(\mathrm{id}, e) & \text{assignment from an } \texttt{int} & | & op(\mathrm{id}, \mathrm{id}, \mathrm{id}) & op \in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}\} \\
 & | & \texttt{set\_s}(\mathrm{id}, l) & \text{assignment from} & | & \mathrm{id} = \texttt{cmp}(\mathrm{id}, \mathrm{id}) & \texttt{mpz}\text{ comparison} \\
 & & & \text{a string literal} & | & \mathrm{id} = \texttt{get\_int}(\mathrm{id}) & \texttt{mpz}\text{ coercion}
\end{array}
$$

Figure 3: Syntax of the mini-GMP language.

We denote $\mathcal{V}$ the set of program variables, $\mathcal{S}$ the set of program statements, $\mathfrak{L}$ the set of logic binders (i.e. the logic variables serving as parameters of user-defined logic functions and predicates), $\mathfrak{Z}$ the set of logical terms and $\mathfrak{B}$ the set of predicates. We also denote $\mathfrak{T}$ the set of mini-GMP types and ty the function that gives the type of a mini-GMP expression. For the sake of simplicity, we model the names of the program routines as program variables in $\mathcal{V}$, and the name of logic routines as binders in $\mathfrak{L}$. The definition of program functions are recorded in a partial function $\mathcal{F} : \mathcal{V} \rightharpoonup \mathcal{V}^* \times \mathcal{S}$, associating to each variable, the list of variables corresponding to its parameters, together with the statement corresponding to its body. Similarly, $\mathcal{P} : \mathcal{V} \rightharpoonup \mathcal{V}^* \times \mathcal{S}$ models the program procedures, $\mathfrak{F} : \mathfrak{L} \rightharpoonup \mathfrak{L}^* \times \mathfrak{Z}$ models the logic functions and $\mathfrak{P} : \mathfrak{L} \rightharpoonup \mathfrak{L}^* \times \mathfrak{B}$ models the user-defined predicates.

3.3 Language Semantics

We consider a single framework to express the semantics of the three languages mini-C, mini-FSL and mini-GMP. We denote $\mathsf{m}_{\texttt{int}}$ and $\mathsf{M}_{\texttt{int}}$ the minimal and the maximal integer representable in type `int`. The set of values that an expression may evaluate to is $\mathbb{V} = \mathsf{Int} \uplus \mathsf{Mpz} \uplus \mathbb{U}_{\texttt{int}} \uplus \mathbb{U}_{\texttt{mpz}}$, with Int being the set of all values of type int, Mpz an enumerable set representing memory locations for values of type `mpz`, and $\mathbb{U}_\tau$ is an enumerable set of undefined values used to denote uninitialized variables of type τ. We retrieve an integer from a value thanks to the functions $\underline{\cdot} : \mathsf{Int} \to \mathbb{Z}$ and $\mathcal{M} : \mathsf{Mpz} \rightharpoonup \mathbb{Z}$. The former is a static function that transcribes the encoding of integers as Int values, while the latter (called *memory state*) changes throughout the execution of the program and represents the current contents of memory locations, which are limited to containing `mpz` integers in our setting. For logical annotations, we denote $\mathbb{B} = \{0, 1\}$ the set of truth values. A *(semantical) environment* Ω is a pair of two partial functions $\Omega_{\mathcal{V}} : \mathcal{V} \rightharpoonup \mathbb{V}$ and $\Omega_{\mathfrak{L}} : \mathfrak{L} \rightharpoonup \mathbb{Z}$. For the sake of simplicity, we treat Ω as a single partial function and use the context to distinguish $\Omega_{\mathcal{V}}$ from $\Omega_{\mathfrak{L}}$. The semantic is expressed by $\Omega, \mathcal{M} \vDash s \Rightarrow \Omega', \mathcal{M}'$ and associates to each statement s in a semantical environment Ω and a memory state $\mathcal{M}$, a new environment Ω' and a new memory state $\mathcal{M}'$. Similarly, for an expression e (resp. a logical term t, predicate p), we denote $\Omega \vDash e \Rightarrow v$ with $v \in \mathbb{V}$ (resp. $\Omega \vDash t \Rightarrow z$ with $z \in \mathbb{Z}$, $\Omega \vDash p \Rightarrow b$ with $b \in \mathbb{B}$) its semantics. Fig. 4 presents the semantic rules for program functions, procedures, logic functions and predicates. The other rules are omitted because almost standard and straightforward (yet provided in Appendix A for completeness). For a mini-C function f, we use a distinguished variable res_f for transmitting the result from the callee to the caller. Fig. 5 presents the rules for evaluating the GMP primitives of mini-GMP. This semantics is compliant with the documentation of GMP.

The semantics is blocking [14, 15]: an incorrect program, or one with invalid assertions has no semantics. It is also non-deterministic: declared but unassigned variables may take any undefined value. A satisfied assertion does not change the environment nor the

$$\frac{\mathcal{F}(f) = (x_1, \ldots, x_n; b) \qquad \Omega \vDash e_1 \Rightarrow z_1; \ldots; \Omega \vDash e_n \Rightarrow z_n \qquad \bot\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\}, \mathcal{M} \vDash b \Rightarrow \Omega', \mathcal{M}' \qquad \Omega'(\mathrm{res}_f) = z}{\Omega, \mathcal{M} \vDash c \;=\; f(e_1, \ldots, e_n) \Rightarrow \Omega\{c \backslash z\}, \mathcal{M}'}$$

$$\frac{\mathcal{P}(p) = (x_1, \ldots, x_n; b) \qquad \Omega \vDash e_1 \Rightarrow z_1; \ldots; \Omega \vDash e_n \Rightarrow z_n \qquad \bot\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\}, \mathcal{M} \vDash b \Rightarrow \Omega', \mathcal{M}'}{\Omega, \mathcal{M} \vDash p(e_1, \ldots, e_n) \Rightarrow \Omega, \mathcal{M}'}$$

$$\frac{\Omega, \mathcal{M} \vDash e \Rightarrow z}{\Omega, \mathcal{M} \vDash \mathsf{return}(e) \Rightarrow \Omega\{\mathrm{res}_f \backslash z\}, \mathcal{M}} \qquad \frac{\Omega \vDash p \Rightarrow 1}{\Omega, \mathcal{M} \vDash \texttt{/*@ assert}\ p\ \texttt{*/} \Rightarrow \Omega, \mathcal{M}}$$

$$\frac{\mathfrak{F}(f) = (x_1, \ldots, x_n; b) \qquad \Omega \vDash t_1 \Rightarrow z_1; \ldots; \Omega \vDash t_n \Rightarrow z_n \qquad \bot\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \vDash b \Rightarrow z}{\Omega \vDash f(t_1, \ldots, t_n) \Rightarrow z}$$

$$\frac{\mathfrak{P}(p) = (x_1, \ldots, x_n; b) \qquad \Omega \vDash t_1 \Rightarrow z_1; \ldots; \Omega \vDash t_n \Rightarrow z_n \qquad \bot\{x_1 \backslash z_1, \ldots, x_n \backslash z_n\} \vDash b \Rightarrow z}{\Omega \vDash p(t_1, \ldots, t_n) \Rightarrow z}$$

Figure 4: Rules for function calls in mini-C and mini-FSL

$$\frac{\forall v \in \mathcal{V}, \Omega(v) \neq z \qquad \Omega_{\mathcal{V}}(x) \in \mathbb{U}_{\mathsf{mpz}}}{\Omega, \mathcal{M} \vDash \mathsf{init}(x); \Rightarrow \Omega\{x \backslash z\}, \mathcal{M}\{z \backslash 0\}} \qquad \frac{\Omega_{\mathcal{V}}(x) = a \qquad u \in \mathbb{U}_{\mathsf{mpz}}}{\Omega, \mathcal{M} \vDash \mathsf{cl}(x); \Rightarrow \Omega\{x \backslash \emptyset_{\mathsf{Mpz}}\}, \mathcal{M}\{a \backslash u\}}$$

$$\frac{\Omega_{\mathcal{V}}(x) = a \qquad \Omega, \mathcal{M} \vDash y \Rightarrow z}{\Omega, \mathcal{M} \vDash \mathsf{set\_i}(x, y); \Rightarrow \Omega, \mathcal{M}\{a \backslash \dot{z}\}} \qquad \frac{\Omega_{\mathcal{V}}(x) = a \qquad \mathcal{M}(\Omega_{\mathcal{V}}(y)) = z \in \mathbb{Z}}{\Omega, \mathcal{M} \vDash \mathsf{set\_z}(x, y); \Rightarrow \Omega, \mathcal{M}\{a \backslash z\}}$$

$$\frac{\Omega_{\mathcal{V}}(x) = a \qquad \mathrm{parse}(s) = z}{\Omega, \mathcal{M} \vDash \mathsf{set\_s}(x, s); \Rightarrow \Omega, \mathcal{M}\{a \backslash z\}} \qquad \frac{\Omega, \mathcal{M} \vDash y \Rightarrow v_y \quad \mathrm{m}_{\mathsf{int}} \leq \mathcal{M}(v_y) \leq \mathrm{M}_{\mathsf{int}}}{\Omega, \mathcal{M} \vDash x = \mathsf{get\_i}(y); \Rightarrow \Omega\{x \backslash \mathcal{M}(v_y)^{\mathsf{int}}\}, \mathcal{M}}$$

$$\frac{\Omega, \mathcal{M} \vDash x \Rightarrow v_x \qquad \mathcal{M}(v_x) = z_1 \qquad \Omega, \mathcal{M} \vDash y \Rightarrow v_y \qquad \mathcal{M}(v_y) = z_2}{\Omega, \mathcal{M} \vDash \mathsf{op}(r, x, y); \Rightarrow \Omega, \mathcal{M}\{\Omega(r) \backslash z_1 \diamond z_2\}}$$

$$\frac{\Omega, \mathcal{M} \vDash x \Rightarrow v_x \qquad \Omega, \mathcal{M} \vDash y \Rightarrow v_y \qquad \mathcal{M}(v_x) \bowtie \mathcal{M}(v_y)}{\Omega, \mathcal{M} \vDash c \;=\; \mathsf{cmp}(x, y); \Rightarrow \Omega\{c \backslash b\}, \mathcal{M}} \qquad b = \begin{cases} 1 & \text{when } \bowtie \text{ is } > \\ -1 & \text{when } \bowtie \text{ is } < \\ 0 & \text{when } \bowtie \text{ is } = \end{cases}$$

Figure 5: Semantics for the GMP instructions

memory state. Hence, if an annotated program has a semantics, it is the same than that of the corresponding non-annotated program. There is no finite derivation tree for the semantics of a call to a non terminating recursive function or predicate.

3.4 Static Analysis: Interval and Type Inference

As explained in Sec. 2, our translation relies on a static analysis in order to decide when a generated expression can safely use machine integer or must use exact GMP integers, of type mpz. This analysis has already been formalized for an integer language without functions and predicates [16] and formalizing it in presence of functions and predicates is left to future work. In the presence of recursive functions, it requires a fixpoint computation. Here, we briefly explain informally its general principle: for each mini-FSL term, this analysis computes an interval that over-approximates the values it may range over, as well as a type in mini-C. Because it only computes an over-approximation (the problem is undecidable in the general case), it is a trade-off between precision and efficiency: the more precise the analysis is, the more time it needs to be computed, but the more efficient the generated program is. In this article, we assume that this analysis has been soundly computed and provides a "precise enough" result for our examples.

More formally, we assume an oracle $\mathcal{I}$ providing the interval $i \in I$ inferred by this analysis. Accessing this oracle requires a *typing environment*, denoted $\Gamma_{\mathcal{I}} : \mathfrak{L} \rightharpoonup I$, that maps logic binders (here, function and predicate parameters) to intervals. Therefore, the oracle $\mathcal{I}$ is a function of type $\mathfrak{Z} \rightarrow (\mathfrak{L} \rightharpoonup I) \rightarrow I$. We denote Θ the operator, formalized in [16], that associates to any interval the mini-GMP type inferred by the static analysis that can represent this interval. We define $\mathcal{T} = \Theta \circ \mathcal{I}$: when translating a term t in an environment Γ, it corresponds to the mini-GMP type of the resulting expression. We assume that the type inferred for a function call is the same as the type inferred for the body of the function in the environment associating to each argument its inferred interval.

Hypothesis 1 (Type Soundness). *In any environment Ω, a term t evaluates to a value z that fits into $\mathcal{T}(t)$.*

Hypothesis 2 (Convergence). *The typing converges: each function and predicate gets typed in finite time. In practice, that means that it gets typed in finitely many environments $\Gamma_{\mathcal{I}}$*

4 TRANSLATION WITHOUT ROUTINES

We now present one of our key contributions: a formalization of the translation from mini-C to mini-GMP for RAC. We split this study into two steps: this section defines the translation for terms and predicates, but ignores functions and user-defined predicates. Then, Sec. 5 specifically deals with function and predicate calls. For terms and predicates, the main challenge consists in accounting for the result of the static analysis when generating int or mpz expressions. For this purpose, we rely on a macro-based translation scheme allowing us to factorize the generated code irrespectively of the types, to prevent a combinatory explosion.

4.1 Translating Declarations

Given a mini-C program P, we denote $[\![P]\!]$ the mini-GMP program generated by our runtime assertion checker. A program is essentially a sequence of statements that are encapsulated in functions, possibly preceded by variable declarations. Translating a function just consists in translating its statements one after the other. However, when translating logic function and predicate calls into function calls in Sec. 5, we will need an *environment of global definitions* $\Psi : \mathfrak{L} \times (\mathfrak{L} \rightharpoonup I) \rightharpoonup \mathcal{V}$ explained in Sec. 5.1. While introduced right now, it remains unused for the time being. Furthermore, for every function or statement f, our translation, in an environment of global definitions Ψ, not only generates a chunk of code, denoted ${}_{\Psi}[\![f]\!]$, but also produces a new environment of global definitions, denoted ${}_{\Psi}[\![f]\!]_{\mathrm{env}}$, which is the list of new routines generated when translating f, that we will only populate in Sec. 5. Since a usual pattern consists in passing this environment to the translation of the next statement, we denote ${}_{\Psi}[\![f_1]\!] \cdot [\![f_2]\!]$ the code chunk ${}_{\Psi'}[\![f_2]\!]$ with $\Psi' = {}_{\Psi}[\![f_1]\!]_{\mathrm{env}}$. We also define a function ${}_{\Psi}[\![f]\!]_{\mathrm{glob}}$ that denotes the mini-C globals generated during the translation of f.

Fig. 6 formally defines the translation of mini-C programs and functions (or procedures): the translation of a program preserves its global variables, then inserts all the generated routines needed to translate the functions and then the translation of all the functions sequentially. The translation of a function is a function where all the statements are translated sequentially, and the routines generated by a function are the ones generated by each statement.

4.2 Translating Statements.

Translating mini-C statements lets anything but logic assertions unchanged. Therefore we only present how these are translated. The translation requires an additional environment called *environment for bindings* $\Gamma_{\mathcal{V}} : \mathfrak{L} \rightharpoonup \mathcal{V} \times I$, which stores the correspondence between a binder and the variable generated to represent it with the interval inferred for this variable. For convenience we sometimes write the components as follows: $\Gamma(x) = (\Gamma_{\mathcal{V}}(x), \Gamma_{\mathcal{I}}(x))$. We denote ${}^{\Gamma}_{\Psi}[\![p]\!]$ the translation of a predicate p in the environments Γ, Ψ. For

```
$[\![\mathrm{decl}; f_1; f_2; \ldots]\!] =$
  | decl;
  | ${}_{\perp}[\![f_1]\!]_{\mathrm{glob}}$
  | ${}_{\perp}[\![f_1]\!] \cdot [\![f_2]\!]_{\mathrm{glob}}$
  | $\ldots$
  | ${}_{\perp}[\![f_1]\!]$
  | ${}_{\perp}[\![f_1]\!] \cdot [\![f_2]\!]$
  | $\ldots$

${}_{\Psi}[\![\tau\ f\ (\mathrm{d})\{\mathrm{decl}; s_1, s_2, \ldots\}]\!]_{\mathrm{env}} =$
  | ${}_{\Psi}[\![s_1]\!] \cdot \ldots \cdot [\![s_n]\!]_{\mathrm{env}}$

${}_{\Psi}[\![\tau\ f\ (\mathrm{d})\{\mathrm{decl}; s_1, s_2, \ldots\}]\!]_{\mathrm{glob}} =$
  | ${}_{\Psi}[\![s_1]\!]_{\mathrm{glob}}$
  | ${}_{\Psi}[\![s_1]\!] \cdot [\![s_2]\!]_{\mathrm{glob}}$
  | $\ldots$

${}_{\Psi}[\![\tau\ f\ (\mathrm{d})\{\mathrm{decl}; s_1, s_2, \ldots\}]\!] =$
  | $\tau\ f\ (\mathrm{d})\{$
  |   decl;
  |   ${}_{\Psi}[\![s_1]\!]$
  |   ${}_{\Psi}[\![s_1]\!] \cdot [\![s_2]\!]$
  |   $\ldots$
  | }
```

Figure 6: Translation of Programs and Functions.

```
DECLS((τ, v)::vars)=τ v; DECLS(vars)
               DECLS([])=skip;

INITS((mpz, z)::vars)=init(z); INITS(vars)
INITS((int, z)::vars)=INITS(vars)
               INITS([])=skip;

CLEARS((mpz, z)::vars)=cl(z); CLEARS(vars)
CLEARS((int, z)::vars)=CLEARS(vars)
               CLEARS([])=skip;

${}^{\Gamma}_{\Psi}[\![$/*@ assert p; */$]\!] =$
  | {
  |   DECLS${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}}$;
  |   INITS${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}}$;
  |   ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{code}}$;
  |   assert(${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{res}}$);
  |   CLEARS${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}}$;
  | }
```

Figure 7: Translation of Logic Assertions.

simplicity, we decompose it into three parts: ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}}$ is the list of fresh variables together with their C type, generated by the translation, ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{res}}$ denotes the distinguished variable containing the result, and ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{code}}$ is a generated chunk of code. It is typically a sequence of statements that uses the variables in ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}}$ and assigns to the variable ${}^{\Gamma}_{\Psi}[\![p]\!]\cdot_{\mathrm{res}}$ the value 0 or 1 corresponding to the truth value of predicate p. We recover the generated code ${}^{\Gamma}_{\Psi}[\![p]\!]$ from this data by handling all the declarations, initializations and deallocations of the variables, as shown in Fig. 7.

The statement translation uses three helper functions DECLS, INITS and CLEARS that respectively generate the declaration, initialization and deallocation of a list of variables. Similarly to functions, ${}^{\Gamma}_{\Psi}[\![p]\!]_{\mathrm{env}}$ and ${}^{\Gamma}_{\Psi}[\![p]\!]_{\mathrm{glob}}$ respectively denote an environment of global definitions associated to the translation of p and the list of routines generated during the translation.

4.3 Macro Definitions

Our translation uses a set of *macros*, introduced in Fig. 8. It helps us factor out some core mechanisms required to generate correct code in the mini-GMP language, independently of the generated types (either `int` or `mpz`). These macros are written in a self-explanatory meta-language. In order to distinguish it from mini-C and mini-GMP, all keywords of this meta-language are capitalized.

The macro int_ASSGN(v, e) (resp. mpz_ASSGN(v, e)) assigns the expression e to the variable v of type `int` (resp. `mpz`), according to the type of e. The macro CMP(c, e_1, e_2, v_1, v_2) assigns to c a non-negative integer if $e_1 > e_2$, a non-positive one if $e_1 < e_2$ and `0` if they are equal. The variables v_1 and v_2 can freely be used for storing intermediate results. The macro $\mathbb{Z}$_ASSGN(τ_z, v, z) assigns to the variable v the value of the integer z, of type τ_z. When the integer z is too large, we switch to a string representation and use `set_s`. The macro $\diamond$_ASSGN$((\tau_c, c), e_1, e_2, r, v_1, v_2)$ assigns to the variable c the result of $e_1 \diamond e_2$ represented in the type τ_c. The variables r, v_1, v_2 can freely be used during the intermediate steps. Here, $\diamond \in \{+, -, \times, /\}$ is a mini-FSL operation, $\square \in \{$`+`, `-`, `*`, `/`$\}$ is the corresponding mini-C operation and op $\in \{$`add`, `sub`, `mul`, `div`$\}$ is the name of the corresponding mini-GMP function.

```
mpz_ASSGN($v$, $e$) :=
  | MATCH ty($e$) WITH :
  |   CASE mpz :
  |     set_z($v$, $e$);
  |   CASE int:
  |     set_i($v$, $e$);

int_ASSGN($v$, $e$) :=
  | MATCH ty($e$) WITH :
  |   CASE int :
  |     $v$ = $e$;
  |   CASE mpz :
  |     $v$ = get_i($e$);

CMP($c$, $e_1$, $e_2$, $v_1$, $v_2$) :=
  | MATCH ty($e_1$), ty($e_2$) WITH :
  |   CASE int, int :
  |     if ($e_1 < e_2$) $c$ = -1;
  |     else if($e_1 > e_2$) $c$ = 1;
  |     else $c$ = 0;
  |   DEFAULT :
  |     mpz_ASSGN($v_1$, $e_1$)
  |     mpz_ASSGN($v_2$, $e_2$)
  |     $c$ = cmp($v_1$, $v_2$);

$\mathbb{Z}$_ASSGN($\tau_z$, $v$, $z$) :=
  | MATCH $\tau_z$ WITH :
  |   CASE int :
  |     $v$ = $z$;
  |   CASE mpz :
  |     set_s($v$, "$z$");

$\diamond$_ASSGN(($\tau$, $c$), $e_1$, $e_2$, $r$, $v_1$, $v_2$) :=
  | MATCH $\tau$, ty($e_1$), ty($e_2$) WITH :
  |   CASE int, int, int :
  |     $c$ = $e_1 \square e_2$;
  |   DEFAULT :
  |     mpz_ASSGN($v_1$, $e_1$)
  |     mpz_ASSGN($v_2$, $e_2$)
  |     MATCH $\tau$ WITH :
  |       CASE int :
  |         op($r$, $v_1$, $v_2$);
  |         int_ASSGN($c$, $r$)
  |       CASE mpz :
  |         op($c$, $v_1$, $v_2$);
```

Figure 8: Macro Definitions.

```
${}_{\Psi}[\![$\true$]\!]\cdot_{\mathrm{decl}} = \{$int, $\bar{c}\}$
${}_{\Psi}[\![$\true$]\!]\cdot_{\mathrm{code}} = \bar{c}$ = 1;
${}_{\Psi}[\![$\true$]\!]\cdot_{\mathrm{res}} = \bar{c}$
${}_{\Psi}[\![$\true$]\!]_{\mathrm{env}} = \Psi$

${}_{\Psi}[\![$\false$]\!]\cdot_{\mathrm{decl}} = \{$int, $\bar{c}\}$
${}_{\Psi}[\![$\false$]\!]\cdot_{\mathrm{code}} = \bar{c}$ = 0;
${}_{\Psi}[\![$\false$]\!]\cdot_{\mathrm{res}} = \bar{c}$
${}_{\Psi}[\![$\false$]\!]_{\mathrm{env}} = \Psi$

${}_{\Psi}[\![t_1 \triangleleft t_2]\!]\cdot_{\mathrm{decl}} =$
  | ${}_{\Psi}[\![t_1]\!]\cdot_{\mathrm{decl}} \cup {}_{\Psi}[\![t_2]\!]\cdot_{\mathrm{decl}} \cup$
  |   $\{($int, $\bar{c}), ($mpz, $v_1), ($mpz, $v_2)\}$

${}_{\Psi}[\![t_1 \triangleleft t_2]\!]\cdot_{\mathrm{code}} =$
  | ${}_{\Psi}[\![t_1]\!]\cdot_{\mathrm{code}}$
  | ${}_{\Psi}[\![t_1]\!] \cdot [\![t_2]\!]_{\mathrm{.code}}$
  | CMP($\bar{c}$, ${}_{\Psi}[\![t_1]\!]\cdot_{\mathrm{res}}$, ${}_{\Psi}[\![t_2]\!]\cdot_{\mathrm{res}}$, $v_1$, $v_2$)
  | $\bar{c} = \bar{c}$ ◖0;

${}_{\Psi}[\![t_1 \triangleleft t_2]\!]\cdot_{\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![t_1 \triangleleft t_2]\!]_{\mathrm{env}} = {}_{\Psi}[\![t_1]\!] \cdot [\![t_2]\!]_{\mathrm{env}}$

${}_{\Psi}[\![!p]\!]\cdot_{\mathrm{decl}} = {}_{\Psi}[\![p]\!]\cdot_{\mathrm{decl}} \cup \{($int, $\bar{c})\}$

${}_{\Psi}[\![!p]\!]\cdot_{\mathrm{code}} =$
  | ${}_{\Psi}[\![p]\!]\cdot_{\mathrm{code}}$
  | if(${}_{\Psi}[\![p]\!]\cdot_{\mathrm{res}}$){ $\bar{c} = 0$; }
  | else { $\bar{c} = 1$; }

${}_{\Psi}[\![!p]\!]\cdot_{\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![!p]\!]_{\mathrm{env}} = {}_{\Psi}[\![p]\!]_{\mathrm{env}}$

${}_{\Psi}[\![p_1 \,||\, p_2]\!]\cdot_{\mathrm{decl}} =$
  | ${}_{\Psi}[\![p_1]\!]\cdot_{\mathrm{decl}} \cup {}_{\Psi}[\![p_1]\!]\cdot_{\mathrm{decl}} \cup ($int, $\bar{c})$

${}_{\Psi}[\![p_1 \,||\, p_2]\!]\cdot_{\mathrm{code}} =$
  | ${}_{\Psi}[\![p_1]\!]\cdot_{\mathrm{code}}$
  | if(${}_{\Psi}[\![p_1]\!]\cdot_{\mathrm{res}}$){$\bar{c} = 1$}
  | else{
  |   ${}_{\Psi}[\![p_2]\!]\cdot_{\mathrm{code}}$
  |   $\bar{c} = {}_{\Psi}[\![p_2]\!]\cdot_{\mathrm{res}}$;
  | }

${}_{\Psi}[\![p_1 \,||\, p_2]\!]\cdot_{\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![p_1 \,||\, p_2]\!]_{\mathrm{env}} = {}_{\Psi}[\![p_1]\!] \cdot [\![p_2]\!]_{\mathrm{env}}$
```

Figure 9: Predicate Translation.

4.4 Translating Predicates

Fig. 9 formally introduces the predicate translation, in which Γ is omitted: it is only used when translating function and predicate calls in Sec. 5. Here, it is just propagated to every sub-term and sub-predicate. Letters with a bar (e.g. $\bar{c}$) are used for denoting fresh variables. Translating the value `\true` (resp. `\false`) just assigns 1 (resp. 0) to the result. Translating comparison operators relies on the translation of arithmetic terms, detailed in the next paragraph, and uses the macro comparison to compute the result $\bar{c}$. Here, $\triangleleft$ is a logic comparison operator and ◖ its corresponding mini-C operator. The inductive cases are trivial by translating each operand, and performing the proper mini-C operation using the results inductively computed, and stores the resulting value in $\bar{c}$. Since predicates are evaluated to 0 or 1, their result always fits in an `int`. In absence of routine call, the environment Ψ is never modified, yet Fig. 9 shows how to propagate it in anticipation for Sec. 5.

4.5 Translating Terms

Translating terms is formally introduced in Fig. 10. A program variable is translated into itself, while we look into the local environment Ω to translate a logic binder into its corresponding program

```
${}_{\Psi}[\![\mathsf{v}]\!]_{\cdot\mathrm{res}} = \mathsf{v}$

${}^{\Gamma}_{\Psi}[\![x]\!]_{\cdot\mathrm{res}} = \Gamma_{\mathcal{V}}(x)$

Denote $\tau = \mathcal{T}(t_1 \diamond t_2, \Gamma_{\mathcal{I}})$ :
${}_{\Psi}[\![t_1 \diamond t_2]\!]_{\cdot\mathrm{decl}} =$
    ${}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{decl}} \cup {}_{\Psi}[\![t_2]\!]_{\cdot\mathrm{decl}} \cup$
      $\{(\tau, \bar{c}), (\mathsf{mpz}, \bar{v}_1), (\mathsf{mpz}, \bar{v}_2), (\mathsf{mpz}, \bar{r})\}$

${}_{\Psi}[\![t_1 \diamond t_2]\!]_{\cdot\mathrm{code}} =$
    ${}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{code}}$
    ${}_{\Psi}[\![t_1]\!] \cdot [\![t_2]\!]_{.\mathrm{code}}$
    $\diamond\_\textsc{assgn}((\tau, \bar{c}), [\![t_1]\!]_{\cdot\mathrm{res}}, [\![t_2]\!]_{\cdot\mathrm{res}},$
                $\bar{r}, \bar{v}_1, \bar{v}_2)$

${}_{\Psi}[\![t_1 \diamond t_2]\!]_{\cdot\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![t_1 \diamond t_2]\!]_{\mathrm{env}} = {}_{\Psi}[\![t_1]\!] \cdot [\![t_2]\!]_{\mathrm{env}}$

${}_{\Psi}[\![z]\!]_{\cdot\mathrm{decl}} = (\mathcal{T}(z, \Gamma_{\mathcal{I}}), \bar{c})$
${}_{\Psi}[\![z]\!]_{\cdot\mathrm{code}} = \mathbb{Z}\_\textsc{assgn}(\mathcal{T}(z, \Gamma_{\mathcal{I}}), \bar{c}, z)$
${}_{\Psi}[\![z]\!]_{\cdot\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![p?t_1 : t_2]\!]_{\cdot\mathrm{decl}} =$
    ${}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}} \cup {}_{\Psi}[\![t1]\!]_{\cdot\mathrm{decl}} \cup {}_{\Psi}[\![t2]\!]_{\cdot\mathrm{decl}} \cup$
      $\{(\mathcal{T}(p?t_1 : t_2, \Gamma_{\mathcal{I}}), \bar{c})\}$

${}_{\Psi}[\![p?t_1 : t_2]\!]_{\cdot\mathrm{code}} =$
    ${}_{\Psi}[\![p]\!]_{\cdot\mathrm{code}}$
    if (${}_{\Psi}[\![p]\!]_{\cdot\mathrm{res}}$) {
      ${}_{\Psi}[\![p]\!] \cdot [\![t_1]\!]_{.\mathrm{code}}$
      $\mathcal{T}(p?t_1 : t_2, \Gamma_{\mathcal{I}})\_\textsc{assgn}(\bar{c}, {}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{res}})$
    } else {
      ${}_{\Psi}[\![p]\!] \cdot [\![t_1]\!] \cdot [\![t_2]\!]_{.\mathrm{code}}$
      $\mathcal{T}(p?t_1 : t_2, \Gamma_{\mathcal{I}})\_\textsc{assgn}(\bar{c}, {}_{\Psi}[\![t_2]\!]_{\cdot\mathrm{res}})$
    }

${}_{\Psi}[\![p?t_1 : t_2]\!]_{\cdot\mathrm{res}} = \bar{c}$

${}_{\Psi}[\![p?t_1 : t_2]\!]_{\mathrm{env}} = {}_{\Psi}[\![p]\!] \cdot [\![t_1]\!] \cdot [\![t_2]\!]_{\mathrm{env}}$
```

Figure 10: Term Translation.

variable. As explained in Section 5, this lookup necessarily succeeds. Translating a binary operation requires to translate its operands and to invoke the associated macro, while translating a conditional requires to translate the condition and to produce a `if` statement performing the test. Translating a constant stores it into a fresh variable for consistency with the other cases, even if it fits into a machine integer. The cases where the environment returned is Ψ or the command generated is `skip;` are omitted.

5 TRANSLATION OF ROUTINE CALLS

5.1 Compilation Scheme

We previously introduced an environment Ψ without describing it. It aims at reusing a function definition already generated when possible. It is not only useful for avoiding the generation of spurious functions, but also mandatory to handle recursive ones. Ψ keeps a mapping from logic functions to their generated counterparts. Though, since our translation uses function specialization, multiple counterparts are possible for a same logic function. For instance, in Fig. 1, the functions `mean_1` and `mean_2` translate the same logic function `mean`. The former is more efficient but may overflow, thus it is only used when the monitor generator can ensure that no overflow is possible (like it is the case for the call at line 24). To account for this, Ψ records the interval inferred by for each argument of a logic function, i.e., the typing environment at the call site. Therefore, denoting $\mathfrak{R} = \operatorname{dom}\mathfrak{F} \cup \operatorname{dom}\mathfrak{P}$ the set of all binders corresponding to a logic routine, Ψ is a partial function $\mathfrak{R} \times (\mathfrak{L} \rightharpoonup I) \rightharpoonup \mathcal{V}$: given a binder denoting a logic function or predicate and a typing environment, it returns a variable denoting a corresponding specialized function, if it exists.

Function duplication. Only reusing a function when both calls define the same typing environment is conservative, and may lead to code duplication. For instance, in Fig. 1, if one added the assertion `mean_implem(15999, 25001)== mean(15999, 25001)`, a new function `mean_3` would be generated, since the inferred intervals are singletons different from the intervals of the other calls. However, `mean_3` would be a mere copy of `mean_2`, which could have been safely used. Avoiding such a code duplication is not easy in the general case and this optimization is left for future work.

```
${}^{\Gamma}_{\Psi}(\!|f|\!) =$
    ${}^{\Gamma}_{\Psi}[\![b]\!]_{\mathrm{glob}}$

    ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{sign}}$ {
      ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{body}}$
    }

${}^{\Gamma}_{\Psi}(\!|f|\!)_{\mathrm{env}} = {}^{\Gamma}_{\Psi}[\![b]\!]_{\mathrm{env}}\{(f, \Gamma)\backslash{}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\}$

${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{sign}} =$
    MATCH $\mathcal{T}(b, \Gamma_{\mathcal{I}})$ WITH:
      CASE int:
        int ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}$ $(\Theta(\Gamma_{\mathcal{I}}(v_1))\ \Gamma_{\mathcal{V}}(v_1),\ \ldots,\ \Theta(\Gamma_{\mathcal{I}}(v_n))\ \Gamma_{\mathcal{V}}(v_n))$
      CASE mpz:
        void ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}$ (mpz ${}^{\Gamma}(\!|f|\!)_{\cdot\mathrm{res}}$, $\Theta(\Gamma_{\mathcal{I}}(v_1))\ \Gamma_{\mathcal{V}}(v_1),\ \ldots,\ \Theta(\Gamma_{\mathcal{V}}(v_n))\ \Gamma_{\mathcal{V}}(v_n))$

${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{body}} =$
    $\textsc{decls}{}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$; $\textsc{inits}{}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$
    ${}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}}$
    MATCH $\mathcal{T}(b, \Gamma_{\mathcal{I}})$ WITH:
      CASE int:
        $\textsc{clears}{}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$
        return ${}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$;
      CASE mpz:
        set_z(${}^{\Gamma}(\!|f|\!)_{\cdot\mathrm{res}}$, ${}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$);
        $\textsc{clears}{}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}$
```

where $(v_1, \ldots, v_n; b) = \mathfrak{F}(f)$; $\hat{\Psi} = \Psi\{(f, \Gamma)\backslash{}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\}$ and ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}$ are fresh names in $\mathcal{V}$

Figure 11: Function Generation Scheme.

Recursive functions. mini-FSL includes recursive logic functions and predicates. For a same recursive call site, we reuse the same generated functions when possible, in order to avoid a huge number of code duplication. Consider for instance a call to `fac(100)` with `fac` defined as follows:

```
logic integer fac (integer n) = n <= 0 ? 1 : n * fac(n - 1);
```

With the presented approach, there are 101 calls to `fac`, all with different intervals, thus 101 functions generated. We delegate to our inference interval mechanism the task to widen the intervals when necessary to avoid generating too many functions for recursive calls, and assume that our oracle performs this task adequately. We let its formal study to future work.

Generating procedures. If the result of a function call is too large to fit into a machine integer, we use the type `mpz` to represent its result. In practice, this GMP type is equivalent to a pointer: it is not allowed by the C standard to allocate the corresponding memory block inside the function and return the corresponding address. Instead, we generate a procedure of type `void` that takes an extra `mpz` parameter for storing the result. At call site, a fresh `mpz` variable is created and provided as argument for this parameter.

5.2 Function Generation

Translating a function call may require to generate a new mini-GMP function from a logic one. Several functions may be generated from a single logic function, depending on the calling context. For this reason, we define the translation of a function call in two environments: given a binder $f \in \operatorname{dom}(\mathfrak{F}(f))$, we define ${}^{\Gamma}_{\Psi}(\!|f|\!)$ to be the function corresponding to f, in the environments Γ, Ψ. If $(f, \Gamma) \notin \operatorname{dom}(\Psi)$, we generate a function as shown in Fig. 11. We rely on sub-primitives: ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{sign}}$ giving the signature of the function and ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{body}}$ its body. Additionally ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\mathrm{env}}$ is the environment of global definitions after the function generation, and ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}$ the name of the generated function. If $(f, \Gamma) \in \operatorname{dom}(\Psi)$, we define ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}} = \Psi(f, \Gamma)$ and ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\mathrm{env}} = \Psi$.

5.3 Translating Function Calls

Translating function calls consists in translating all the arguments, then generating a corresponding function and calling this function on the translation of the arguments, as presented in Fig. 12. The mechanism that avoids function duplication is managed by Ψ and

```
${}^{\Gamma}_{\Psi}[\![f(t_1, \ldots, t_n)]\!]_{\cdot\mathrm{decl}} =$
  | $\{\mathcal{T}(f(t_1, \ldots, t_n), \Gamma_{\mathcal{I}}), \bar{c}\} \cup {}^{\Gamma}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{decl}} \cup \ldots \cup {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\cdot\mathrm{decl}}$

${}^{\Gamma}_{\Psi}[\![f(t_1, \ldots, t_n)]\!]_{\cdot\mathrm{code}} =$
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{code}}$;
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet [\![t_2]\!]_{.\mathrm{code}}$;
  | ...
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\cdot\mathrm{code}}$
  | MATCH $\mathcal{T}(f(t_1, \ldots, t_n), \Gamma_{\mathcal{I}})$ WITH:
  |   CASE int:
  |     $\bar{c} = {}^{\hat{\Gamma}}_{\hat{\Psi}} (\!| f |\!)_{\cdot\mathrm{name}}({}^{\Gamma}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{res}}, \ldots, {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\cdot\mathrm{res}})$;
  |   CASE mpz:
  |     ${}^{\hat{\Gamma}}_{\hat{\Psi}} (\!| f |\!)_{\cdot\mathrm{name}}(\bar{c}, {}^{\Gamma}_{\Psi}[\![t_1]\!]_{\cdot\mathrm{res}}, \ldots, {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\cdot\mathrm{res}})$;

${}^{\Gamma}_{\Psi}[\![f(t_1, \ldots, t_n)]\!]_{\mathrm{glob}} =$
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!]_{\mathrm{glob}}$;
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet [\![t_2]\!]_{\mathrm{glob}}$;
  | ...
  | ${}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\mathrm{glob}}$;
  | ${}^{\hat{\Gamma}}_{\hat{\Psi}} (\!| f |\!)$

${}^{\Gamma}_{\Psi}[\![f(t_1, \ldots, t_n)]\!]_{\cdot\mathrm{res}} = \bar{c}$

${}^{\Gamma}_{\Psi}[\![f(t_1, \ldots, t_n)]\!]_{\mathrm{env}} = {}^{\hat{\Gamma}}_{\hat{\Psi}} (\!| f |\!)_{\mathrm{env}}$

where $\hat{\Psi} = {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\mathrm{env}}$
and for $\mathfrak{F}(f) = x_1, \ldots, x_n; \_,$
  | $\hat{\Gamma} = \bot\{x_1 \backslash (\bar{v}_1, \mathcal{I}(\Gamma_{\mathcal{I}}, \mathtt{t1})), \ldots, x_n \backslash (\bar{v}_n \mathcal{I}(\Gamma_{\mathcal{I}}, \mathtt{tn}))\}$
```

Figure 12: Translating Function Call.

the function generation procedure. Let us explain a few cases. First, the definition of ${}^{\Gamma}_{\Psi}(\!| f |\!)_{\cdot\mathrm{sign}}$ in Fig. 11 makes explicit that, when the result type of a function is mpz, we generate a procedure call and pass the variable that represents the result as an argument. Symmetrically, an extra argument is added when calling such a function in Fig. 12. Second, since the generation of a function occurs after having translated its arguments, the environment for global definitions is updated accordingly before generating the function, which is the role of $\hat{\Psi}$. Third, when translating a call, all the current bindings are forgotten, and a new environment $\hat{\Gamma}$ is set according to the f's calling context. This new typing environment has no information about binders in Γ: they are out of scope during function generation, but it associates to each parameter of f, a variable and the interval of the term given at the call site.

The translation of predicate calls is omitted. It is similar to the translation of function calls, but simpler since they always return 0 or 1. Therefore, we never generate procedures from predicates.

6 PROPERTIES

This section states the key properties of our translation and sketches their proofs. Complete proofs are provided in appendices D to G.

6.1 Well-formedness of the generated program

Theorem 6.1 (Absence of dangling pointers). *At any point, for every value $z \in Mpz$, $\mathcal{M}(z) \neq \bot$ if and only if there exists a unique variable $x \in \mathcal{V}$ such that $\Omega(x) = z$.*

We prove this by first showing that there is no aliasing in the variables of type mpz. This is actually guaranteed in an easy way by our simplified semantics, and is the motivation to distinguish $\mathbb{U}_{\mathtt{int}}$ from $\mathbb{U}_{\mathtt{mpz}}$. Then, every value in $\mathcal{M}$ is initialized (by default at 0) as soon as a variable points to it, and is reset to $\bot$ when clearing it.

Theorem 6.2 (Absence of memory leak). *At the end of the program execution, $\mathcal{M} = \bot$.*

We prove this theorem by showing that the code block generated for each assertion is such that $\mathcal{M}$ is the same when entering and leaving the block, since it ends with the freeing of all the variables of type mpz declared in the block. Since those blocks are the only ones to access $\mathcal{M}$, it is preserved throughout the program execution.

6.2 Correctness of the generated program

To characterize the semantics of the generated program, we introduce a partial order $\sqsubseteq$ on environments, defined by $\Omega \sqsubseteq \Omega'$ if and only if, for every v such that $\Omega(v) \in \mathsf{Int} \cup \mathsf{Mpz}$, $\Omega(v) = \Omega'(v)$, and for every v such that $\Omega(v) \in \mathbb{U}_{\mathtt{int}}$ (resp. $\Omega(v) \in \mathbb{U}_{\mathtt{mpz}}$), $\Omega'(v) \in \mathbb{U}_{\mathtt{int}} \cup \mathsf{Int}$ (resp. $\Omega'(v) \in \mathbb{U}_{\mathtt{mpz}} \cup \mathsf{Mpz}$).

Theorem 6.3 (Correctness of code generation). *The generated program has a semantics if and only if the original program has one. In that case, the semantics of the generated program subsumes the one of the original program. More formally for a program P, there exists an Ω such that $\bot, \bot \vDash P \Rightarrow \Omega, \bot$ if and only if there exists an Ω' such that $\bot, \bot \vDash [\![P]\!] \Rightarrow \Omega', \bot$. If it is the case, then $\Omega \sqsubseteq \Omega'$.*

We prove this theorem by first characterizing the semantics of the macros, and then combining those to characterize the semantics of the entire program. In particular, this theorem proves the transparency of our monitor: the presence of annotations does not change the semantics of the original program. Since the semantics is blocking, it also implies the soundness of the generated code. Indeed, the semantics of logical implication states a valid annotated program has a semantics if and only if all its annotations are satisfied. This theorem shows that this is then also a necessary and sufficient condition for the generated program to have a semantics.

One of the main difficulties is the translation of calls to functions or predicates: we characterize the behavior of Ψ and Γ, used for the translation, through an invariant for each of these environments. Ψ helps reusing a function already generated, and the soundness of this approach is ensured by the fact that a function is reused only when its callsite passes arguments for which the type system infers the exact same intervals. This choice is conservative: There are many cases where it would be safe to reuse the same function, even though the inferred intervals slightly differ. However, this problem is undecidable in the general case and very hard in practice. Our pragmatic choice generates code that is efficient enough, while allowing us to prove its soundness. The presence of function calls also breaks the well foundedness of the induction, since one needs to prove the result on the body of the function, but Hypothesis 2 guarantees the proof termination.

7 IMPLEMENTATION

The formalization presented in this paper is implemented within E-ACSL [27]. We describe here the main gap between our work and the current implementation, and explain how this implementation has been evaluated in practice.

Gap with the Theory. The current E-ACSL implementation is very close to the formalization presented here. Here, we mainly avoid a few optimizations in the generated code for clarity. The scope of the considered languages is the main difference between the paper and the implementation. In practice, E-ACSL can runtime check C programs and not a simple imperative programming language. It also covers a much larger spectrum of its specification language that the one formalized here [25]. In particular, it supports rational arithmetics over $\mathbb{Q}$ in addition to integer arithmetics [16], as well as any C type (integer and non-integer types), not only `int`. In practice, many user-defined predicates are defined over pointers representing C arrays (not formalized here) and manipulating through first-order quantifiers over their indices. Finally, our semantics is a simplified semantics of C, that prevents aliasing, and allows for instance to pass declared but uninitialized values as arguments of functions. In practice the generated code complies with the actual semantics of C.

Empirical Evaluation. This paper claims several times that the generated code is efficient. It is supported by several previous experiments, including examples specifically written for evaluating the E-ACSL type system [6, 16], and evaluations on existing benchmarks [29], or concrete use cases [8, 22, 24, 28]. Beyond demonstrating scalability, all these experiments increase our confidence in the implementation. For instance, Robles *et al* [24] reported that *"the instrumentation of both MetAcsl* [i.e., their own tool] *and E-ACSL does not introduce any bug"*. Together with the proof of the soundness of the algorithm this gives evidence of the correct functioning of our approach.

8 CONCLUSION AND FUTURE WORK

This paper has presented a formalization of efficient RAC for an arithmetic language extended with user-defined (possibly recursive) logic functions and predicates. It is the first work that formally studies the generation of efficient code for runtime checking arithmetic properties and proves its key properties. It is also the first work that formalizes function specialization in this context. This work is implemented in E-ACSL, the runtime assertion checker of Frama-C, and has been evaluated on concrete experiments.

We plan future work in two directions. The first one consists in continuing the formalization effort: extending the formalization of the type system [16] to logic functions and predicates, extending this formalization to other interesting constructs such as inductive and axiomatic predicates, and taking into account undefinedness during RAC [10], i.e. how to formally prevent to generate code containing undefined behaviors when translating undefined terms such as `1/0`. The second axis of improvements consists in improving the current support of recursive logic functions and predicates. In particular, we could design a more precise type system that would allow us to generate more machine integer code. More generally, optimizing the code generated by adapting existing compilation techniques would certainly have a significant effect in practice. For instance, calling GMP functions prevents several compiler optimizations (e.g., constant folding), since the compiler does not know that they implement simple arithmetic operations. Such optimizations could be directly done by the monitor generator.

Acknowledgement

We are grateful to Loïc Correnson and André Maroneze who proofread a first version of this paper. We also thank the anonymous reviewers for their helpful comments. Altogether, they contribute to improve the quality of this paper.

REFERENCES

[1] L. O. Andersen. 1992. Partial Evaluation of C and Automatic Compiler Generation (Extended Abstract). In *Int. Conf. on Compiler Construction (CC).*

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *Comput. Surveys* (1994).

[3] M. Barnett, M. Fähndrich, K. Rustan M. Leino, P. Müller, W. Schulte, and H. Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* (2011).

[4] A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek. 2021. Formally verified speculation and deoptimization in a JIT compiler. In *Int. Conf. on Principles on Programming Languages (POPL).*

[5] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. 2021. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM* (2021).

[6] T. Benjamin, F. Ridoux, and J. Signoles. 2022. Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution. In *Journées Francophones des Langages Applicatifs (JFLA).* In French.

[7] Y. Bertot and P. Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

[8] A. Blanchard, N. Kosmatov, and F. Loulergue. 2018. A Lesson on Verification of IoT Software with Frama-C. In *Int. Conf. on High Performance Computing Simulation (HPCS).*

[9] S. Blazy and P. Facon. 1994. SFAC, a tool for program comprehension by specialization. In *Workshop on Program Comprehension.*

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

[11] L. A. Clarke and D. S. Rosenblum. 2006. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Software Engineering Notes* (2006).

[12] M. Delahaye, N. Kosmatov, and J. Signoles. 2013. Common Specification Language for Static and Dynamic Analysis of C Programs. In *Symp. on Applied Computing (SAC).*

[13] J.-C. Filliâtre and C. Pascutto. 2021. Ortac: Runtime Assertion Checking for OCaml (tool paper). In *Int. Conf. on Runtime Verification (RV).*

[14] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. 2008. Verification of class liveness properties with Java Modeling Language. *IET Software* 2, 6 (Dec. 2008).

[15] P. Herms, C. Marché, and B. Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *Int. Conf. on Verified Software, Theories, Tools and Experiments (VSTTE'12).*

[16] N. Kosmatov, F. Maurica, and J. Signoles. 2020. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *Int. Conf. on Runtime Verification (RV).*

[17] G. T. Leavens, A. L. Baker, and C. Ruby. 1999. *JML: A Notation for Detailed Design.*

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

[19] D. Ly, N. Kosmatov, F. Loulergue, and J. Signoles. 2020. Verified Runtime Assertion Checking for Memory Properties. In *Int. Conf. on Tests and Proofs (TAP).*

[20] R. Marlet. 2012. *Program Specialization.* Wiley.

[21] G. Melquiond and R. Rieu-Helft. 2020. WhyMP, a formally verified arbitrary-precision integer library. In *Int. Symp. on Symbolic and Algebraic Computation (ISSAC).*

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

[23] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *Int. Conf. on Source Code Analysis and Manipulation (SCAM).*

[24] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. 2019. Tame your Annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties. In *Int. Conf. on Tests and Proofs (TAP).*

[25] J. Signoles. 2021. *E-ACSL Version 1.17. Implementation in Frama-C Plug-in E-ACSL 24.0.* http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[26] J. Signoles. 2021. The E-ACSL Perspective on Runtime Assertion Checking. In *Int. Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX).*

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

[28] F. Védrine, M. Jacquemin, N. Kosmatov, and J. Signoles. 2021. Runtime Abstract Interpretation for Numerical Accuracy and Robustness. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI).*

[29] K. Vorobyov, J. Signoles, and N. Kosmatov. 2017. Shadow State Encoding for Efficient Monitoring of Block-level Properties. In *Int. Symp. on Memory Management (ISMM).*

A SEMANTICS OF THE mini-C LANGUAGE

Fig. 13 presents the (standard) semantics of the core part of the mini-C language, without routine calls, already presented in Fig. 4.

Semantics of declarations

$$\frac{\Omega_{\mathcal{V}}(x) = \bot \qquad u \in \mathbb{U}_\tau}{\Omega,\, \mathcal{M} \vDash t\ x \Rightarrow \Omega\{x \backslash \mathbb{U}_\tau\},\, \mathcal{M}}$$

Semantics of statements

$$\frac{}{\Omega,\, \mathcal{M} \vDash \mathtt{skip;} \Rightarrow \Omega,\, \mathcal{M}}$$

$$\frac{\Omega_{\mathcal{V}}(x) \in \mathsf{Int} \cup \mathbb{U}_{\mathtt{int}} \qquad \Omega,\, \mathcal{M} \vDash e \Rightarrow z}{\Omega,\, \mathcal{M} \vDash x\ =\ e \Rightarrow \Omega\{x \backslash z\},\, \mathcal{M}}$$

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

$$\frac{\Omega,\, \mathcal{M} \vDash e \Rightarrow 0^{\mathtt{int}} \qquad \Omega,\, \mathcal{M} \vDash s' \Rightarrow \Omega',\, \mathcal{M}'}{\Omega,\, \mathcal{M} \vDash \mathtt{if}(e)\ \mathsf{then}\ s\ \mathsf{else}\ s' \Rightarrow \Omega',\, \mathcal{M}'}$$

$$\frac{\Omega,\, \mathcal{M} \vDash \mathtt{if}(e)\ \mathsf{then}\ s;\ \mathtt{while}(e)\ s\ \mathsf{else\ skip} \Rightarrow \Omega',\, \mathcal{M}'}{\Omega,\, \mathcal{M} \vDash \mathtt{while}(e)\ s \Rightarrow \Omega',\, \mathcal{M}'}$$

$$\frac{\Omega,\, \mathcal{M} \vDash s \Rightarrow \Omega',\, \mathcal{M}' \qquad \Omega',\, \mathcal{M}' \vDash s' \Rightarrow \Omega'',\, \mathcal{M}''}{\Omega,\, \mathcal{M} \vDash s\ s' \Rightarrow \Omega'',\, \mathcal{M}''}$$

$$\frac{\Omega,\, \mathcal{M} \vDash e \Rightarrow z \qquad z \neq 0}{\Omega,\, \mathcal{M} \vDash \mathtt{assert(e)} \Rightarrow \Omega,\, \mathcal{M}}$$

Semantics of expressions

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

$$\frac{\Omega \vDash e \Rightarrow z \qquad \Omega \vDash e' \Rightarrow z' \qquad \mathrm{m}_{\mathtt{int}} \leq \dot{z} \diamond \dot{z'} \leq \mathrm{M}_{\mathtt{int}}}{\Omega \vDash e \,\square\, e' \Rightarrow (\dot{z}\ \diamond\ \dot{z'})^{\mathtt{int}}}\ (\diamond \text{ models } \square)$$

$$\frac{\Omega \vDash e \Rightarrow z \qquad \Omega \vDash e' \Rightarrow z' \qquad \dot{z} \lhd \dot{z'}}{\Omega \vDash e \Lbag e' \Rightarrow 1^{\mathtt{int}}}\ (\lhd \text{ models } \Lbag)$$

$$\frac{\Omega \vDash e \Rightarrow z \qquad \Omega \vDash e' \Rightarrow z' \qquad \dot{z} \not\lhd \dot{z'}}{\Omega \vDash e \Lbag e' \Rightarrow 0^{\mathtt{int}}}$$

Figure 13: Semantics of the mini-C language

B SEMANTICS OF THE mini-FSL LANGUAGE

Fig. 14 presents the semantics of the mini-FSL specification language, without routine calls, already presented in Fig. 4.

C PROPERTIES OF THE SEMANTICS

For the sake of simplicity, we use $\Omega, \mathcal{M} \sqsubseteq \Omega', \mathcal{M}'$ as a shorthand for $\Omega \sqsubseteq \Omega'$ and $\mathcal{M} \sqsubseteq \mathcal{M}'$.

Lemma C.1 (Weakening of expression semantics). *An expression e evaluates to a value x in an environment Ω if and only if it evaluates to the same value in all the environment that subsume Ω. More formally, $\Omega \vDash e \Rightarrow x$ if and only if for all Ω' such that $\Omega \sqsubseteq \Omega'$, $\Omega' \vDash e \Rightarrow x$.*

Proof. Suppose that for all Ω' such that $\Omega \sqsubseteq \Omega'$, we have $\Omega' \vDash e \Rightarrow x$, in particular, since we have $\Omega \sqsubseteq \Omega$, it is immediate that $\Omega \vDash e \Rightarrow x$. So it suffices to show the converse. Suppose that $\Omega \vDash e \Rightarrow x$, and consider an environment Ω' such that $\Omega \sqsubseteq \Omega'$, we show by induction on the expression e that $\Omega' \vDash e \Rightarrow x$.

Rules for logical assertions

$$\frac{\Omega \vDash p \Rightarrow 1}{\Omega,\, \mathcal{M} \vDash \mathtt{/*@\ assert}\ p\ \mathtt{*/} \Rightarrow \Omega,\, \mathcal{M}}$$

Rules for terms

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

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

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

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

Rules for predicates

$$\frac{}{\Omega \vDash \mathtt{\backslash true} \Rightarrow 1} \qquad \frac{}{\Omega \vDash \mathtt{\backslash false} \Rightarrow 0}$$

$$\frac{\Omega \vDash t \Rightarrow z \qquad \Omega \vDash t' \Rightarrow z' \qquad z \lhd z'}{\Omega \vDash t\ \lhd\ t' \Rightarrow 1} \qquad \frac{\Omega \vDash t \Rightarrow z \qquad \Omega \vDash t' \Rightarrow z' \qquad z \not\lhd z'}{\Omega \vDash t\ \lhd\ t' \Rightarrow 0}$$

$$\frac{\Omega \vDash p \Rightarrow 0}{\Omega \vDash\ !\ p \Rightarrow 1} \qquad \frac{\Omega \vDash p \Rightarrow 1}{\Omega \vDash\ !\ p \Rightarrow 0}$$

$$\frac{\Omega \vDash p \Rightarrow 1}{\Omega \vDash p\ ||\ p' \Rightarrow 1} \qquad \frac{\Omega \vDash p \Rightarrow 0 \qquad \Omega \vDash p' \Rightarrow z}{\Omega \vDash p\ ||\ p' \Rightarrow z}$$

Figure 14: Semantics of the mini-FSL language

- If $e = z_m$ is a machine integer, then the derivation of $\Omega \vDash e \Rightarrow x$ is necessarily of the form

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

and $x = z_m$. The same derivation then shows that $\Omega' \vDash z_m \Rightarrow z_m$.

- If $e = v$ is a variable access, then the derivation of $\Omega \vDash e \Rightarrow x$ is necessarily of the form

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

Since $\Omega \sqsubseteq \Omega'$, we also have $\Omega'_{\mathcal{V}}(v) = x$ which gives a derivation of $\Omega' \vDash v \Rightarrow x$.

- If $e = e_1 \square e_2$ is an arithmetic operation, then the derivation of $\Omega \vDash e \Rightarrow x$ necessarily terminates with the following rule, where $x = (\dot{x}_1 \diamond \dot{x}_2)^{\mathtt{int}}$, with $\diamond$ the mathematical operator corresponding to $\square$.

$$\frac{\Omega \vDash e_1 \Rightarrow x_1 \qquad \Omega \vDash e_2 \Rightarrow x_2 \qquad \mathrm{m}_{\mathtt{int}} \leq \dot{x} \leq \mathrm{M}_{\mathtt{int}}}{\Omega \vDash e \Rightarrow x}$$

By induction, the derivations of $\Omega \vDash e_1 \Rightarrow x_1$ and $\Omega \vDash e_2 \Rightarrow x_2$ imply respectively derivations of $\Omega' \vDash e_1 \Rightarrow x_1$ and $\Omega' \vDash e_2 \Rightarrow x_2$. Using these two derivations, one can build a derivation of $\Omega \vDash e \Rightarrow x$, using the same rule.

- If $e = e_1 \Lbag e_2$ is an arithmetic relation, then $x = 0^{\mathtt{int}}$ or $1^{\mathtt{int}}$. These two cases being identical, we only present one of them here. Assume $x = 1^{\mathtt{int}}$, then the derivation of $\Omega \vDash e \Rightarrow 1^{\mathtt{int}}$

necessarily terminates with the following rule

$$\frac{\Omega \vDash e_1 \Rightarrow x_1 \qquad \Omega \vDash e_2 \Rightarrow x_2 \qquad {x_1}^{\mathsf{int}} \blacktriangleleft {x_2}^{\mathsf{int}}}{\Omega \vDash e \Rightarrow 1^{\mathsf{int}}}$$

with $\blacktriangleleft$ the mathematical relation corresponding to $\lhd$. Then the derivations of $\Omega \vDash e_1 \Rightarrow x_1$ and $\Omega \vDash e_2 \Rightarrow x_2$ give us by induction derivations of $\Omega' \vDash e_1 \Rightarrow x_1$ and $\Omega' \vDash e_2 \Rightarrow x_2$. Using the same rule, we then get a derivation of $\Omega' \vDash e \Rightarrow 1^{\mathsf{int}}$. □

Lemma C.2 (Weakening of statements semantics). *He have the following useful results about the semantics of the same statement in different but related environments:*

(1) *The judgment $\Omega_0, \mathcal{M}_0 \vDash s \Rightarrow \Omega_1, \mathcal{M}_1$ is derivable if and only if for all $\Omega'_0, \mathcal{M}'_0$ such that $\Omega_0, \mathcal{M}_0 \sqsubseteq \Omega'_0, \mathcal{M}'_0$, there exists $\Omega'_1, \mathcal{M}'_1$ such that $\Omega_1, \mathcal{M}_1 \sqsubseteq \Omega'_1, \mathcal{M}'_1$ and the following judgment is derivable*

$$\Omega'_0, \mathcal{M}'_0 \vDash s \Rightarrow \Omega'_1, \mathcal{M}'_1$$

(2) *If $\Omega_0, \mathcal{M}_0 \vDash s \Rightarrow \Omega_1, \mathcal{M}_1$ is derivable and $\Omega_0 \sqsubseteq \Omega'_0$, $\mathcal{M}_0 \sqsubseteq \mathcal{M}'_0$. Then for a derivation of*

$$\Omega'_0, \mathcal{M}'_0 \vDash s \Rightarrow \Omega'_1, \mathcal{M}'_1$$

for every variable $v \notin \mathrm{dom}\,(\Omega_0)$, we have $\Omega'_0(v) = \Omega'_1(v)$ and, for every address x such that there is no v such that $\Omega_0(v) = x$, we have $\mathcal{M}'_0(x) = \mathcal{M}'_1(x)$.

(3) *If $\Omega_0, \mathcal{M}_0 \vDash s \Rightarrow \Omega_1, \mathcal{M}_1$ is derivable, then for $\Omega'_0, \mathcal{M}'_0 \sqsubseteq \Omega_0, \mathcal{M}_0$ such that $\mathrm{dom}\,(\Omega_0) - \mathrm{dom}\,(\Omega'_0)$ contains only variables that do not appear in s, and $\mathrm{dom}\,(\mathcal{M}_0) - \mathrm{dom}\,(\mathcal{M}'_0)$ contains addresses that are in the image by Ω_0 of variables that do not appear in s, then there exists $\Omega'_1, \mathcal{M}'_1$ such that the following is derivable*

$$\Omega'_0, \mathcal{M}'_0 \vDash s \Rightarrow \Omega'_1, \mathcal{M}'_1$$

Proof. The three parts of this lemma are independent. We proceed by induction on the form of the statement s. There are however 10 different cases for statements in the syntax of mini-C, to which are added 8 cases in the syntax of mini-GMP. Each of this case contains 3 results to check, and the induction are straightforward. For this reason we only present two base cases that are representative here.

- For an assignation statement $x = e$;
  (1) If we have a semantics $\Omega_0, \mathcal{M} \vDash x = e;\ \Rightarrow \Omega'_0, \mathcal{M}$, then we necessarily have $\Omega_0 \vDash e \Rightarrow z$ with $\Omega'_0 = \Omega_0\{x\backslash z\}$ and $\Omega_0(x) \in \mathbb{U}_{\texttt{int}} \cup \mathsf{Int}$, so $\Omega_1(x) \in \mathbb{U}_{\texttt{int}} \cup \mathsf{Int}$. By Lemma C.1, we deduce that for all $\Omega_0 \sqsubseteq \Omega_1$ we have $\Omega_1 \vDash e \Rightarrow z$ and thus $\Omega_1, \mathcal{M} \vDash x = e;\ \Rightarrow \Omega_1\{x\backslash z\}, \mathcal{M}$.
  (2) With the notation of the previous point, since $\Omega_0(x) \in \mathbb{U}_{\texttt{int}} \cup \mathsf{Int}$, we have $x \in \mathrm{dom}\,(\Omega_0)$. Thus for $y \notin \Omega_0(x)$, we have $y \neq x$ and thus $\Omega_1(y) = \Omega_1\{x\backslash z\}(y)$.
  (3) Consider $\Omega_0, \mathcal{M}_0$ such that we have a semantics
  $$\Omega_0, \mathcal{M}_0 \vDash x = e;\ \Rightarrow \Omega_0\{x\backslash z\}, \mathcal{M}_0$$
  together with $\Omega'_0, \mathcal{M}'_0 \sqsubseteq \Omega_0, \mathcal{M}_0$ such that $x \notin \mathrm{dom}\,(\Omega_0) - \mathrm{dom}\,(\Omega'_0)$, then we have a derivation of
  $$\Omega'_0, \mathcal{M}'_0 \vDash x = e;\ \Rightarrow \Omega'_0\{x\backslash z\}, \mathcal{M}'_0$$
- For the assignation of a GMP integer $\texttt{set\_z}(v, y)$;,
  (1) Assume that we have a semantics
  $$\Omega_0, \mathcal{M}_0 \vDash \texttt{set\_i}(v, y); \Rightarrow \Omega_0, \mathcal{M}_0\{\Omega_0(v)\backslash\mathcal{M}_0(\Omega_0(y))\}$$
  and consider $\Omega'_0, \mathcal{M}'_0$ such that $\Omega_0, \mathcal{M}_0 \sqsubseteq \Omega'_0, \mathcal{M}'_0$. Then the existence of the semantics implies that $\Omega_0(v), \Omega_0(y) \in \mathsf{Mpz}$, thus $\Omega'_0(v), \Omega'_0(y) \in \mathsf{Mpz}$. Hence we have the following semantics
  $$\Omega'_0, \mathcal{M}'_0 \vDash \texttt{set\_i}(v, y); \Rightarrow \Omega'_0, \mathcal{M}'_0\{\Omega_0(v)\backslash\mathcal{M}_0(\Omega_0(y))\}$$
  (2) With the previous notations, considering an address x such that there is no v such that $\Omega_0(v) = x$. Then we have $\mathcal{M}'_0(x) = \mathcal{M}'_0\{\Omega_0(v)\backslash\mathcal{M}_0\Omega_0(y)\}(x)$.
  (3) Consider $\Omega_0, \mathcal{M}_0$ such that we have a semantics
  $$\Omega_0, \mathcal{M}_0 \vDash \texttt{set\_i}(v, y); \Rightarrow \Omega_0, \mathcal{M}_0\{\Omega_0(v)\backslash\mathcal{M}_0(\Omega_0(y))\}$$
  Then consider $\Omega'_0, \mathcal{M}'_0 \sqsubseteq \Omega_0, \mathcal{M}_0$ such that $v, y \notin \mathrm{dom}\,(\Omega_0) - \mathrm{dom}\,(\Omega'_0)$, and $\mathrm{dom}\,(\mathcal{M}_0) - \mathrm{dom}\,(\mathcal{M}'_0)$ contains addresses that are in the image by Ω_0 of $\{v, y\}$. Then we necessarily have $\Omega'_0(v) = \Omega_0(v)$ and $\Omega'_0(y) = \Omega_0(y)$ as well as $\mathcal{M}'_0(\Omega'_0(v)) = \mathcal{M}_0(\Omega_0(v))$ and $\mathcal{M}'_0(\Omega'_0(y)) = \mathcal{M}_0(\Omega_0(y))$. Thus we have the following derivation
  $$\Omega'_0, \mathcal{M}'_0 \vDash \texttt{set\_i}(v, y); \Rightarrow \Omega'_0, \mathcal{M}'_0\{\Omega_0(v)\backslash\mathcal{M}_0(\Omega_0(y))\}$$

□

D PROOFS OF STRUCTURAL PROPERTIES OF THE TRANSLATION

Lemma D.1. *If the generated program has a semantics, then for every variable v of type* `mpz`*, the value of $\Omega_{\mathcal{V}}(v)$ stays the same at every point between initialization and clearance of v.*

Proof. Since this is an invariant that we ensure in the generated program, we have defined a semantics that has this invariant built-in. Thus, in the semantics, the only way to give a value to a variable of type `mpz` is through the `init` and `cl` instructions. The distinction between $\mathbb{U}_{\texttt{int}}$ and $\mathbb{U}_{\texttt{mpz}}$ serves as a way to ensure this property in the semantics. □

Lemma D.2 (Absence of aliasing). *There cannot exist two variable pointing to the same memory location: For all $z \in Mpz$, there is at most one variable $v \in \mathcal{V}$ such that $\Omega_{\mathcal{V}}(v) = z$ at any point.*

Proof. By Lemma D.1, a variable v of type `mpz` keeps the same value from initialization to clearance in our formalization, so it suffices to check that the initialization rule does not allow for setting a memory location already contained in another variable. This is exactly the premise of the rule. □

Theorem 6.1 (Absence of dangling pointers). *At any point, for every value $z \in Mpz$, $\mathcal{M}(z) \neq \bot$ if and only if there exists a unique variable $x \in \mathcal{V}$ such that $\Omega(x) = z$.*

Proof. Lemma D.2 shows that if there exists a variable $x \in \mathcal{V}$ such that $\Omega_{\mathcal{V}}(x) = z$, then this variable is unique. By design of the translation, every time such a variable is used, it is first declared and initialized. The declaration does not ascribe a value to the variable (which we model with $\emptyset_{\mathsf{Mpz}}$). The initialization rule gives a value z to the variable, and at the same time, sets $\mathcal{M}(z) = 0$. From there onward, the only rule that allows to change the value of $\Omega_{\mathcal{V}}(v)$ to $\emptyset_{\mathsf{Mpz}}$ is the clearance rule, and Lemma D.1 shows that in between initialization and clearance $\Omega_{\mathcal{V}}(x) = z$. Moreover, none of the rule that modify $\mathcal{M}$ except the clearance rule allow to set the value $\perp$. This shows that as long as $\Omega_{\mathcal{V}}(x) = z$ (*i.e.* between initialization and clearance of v), $\mathcal{M}(z) \neq \perp$. Conversely, all the rules that modify a value in $\mathcal{M}$ at location z require the existence of a variable v such that $\Omega_{\mathcal{V}}(v) = z$, except the initialization one. Since we have already proved that $\Omega_{\mathcal{V}}(v) = z$ implies that $\mathcal{M}(z) \neq \perp$, this implies that the only rule that may change the value of $\mathcal{M}(z)$ from $\perp$ to another value is the initialization rule. This rule requires a variable v such that $\Omega_{\mathcal{V}}(v) = z$. The only rule that may change $\mathcal{M}(z)$ to $\perp$ is the clearance rule, which also sets $\Omega_{\mathcal{V}}(v)$ to $\emptyset_{\mathrm{Mpz}}$. Lemma D.1 shows that as long as $\mathcal{M}(z) \neq \perp$ (*i.e.* in between initialization and clearance of v) $\Omega_{\mathcal{V}}(v) = z$, □

Lemma D.3 (preservation of the control flow). *The control flow graph of the program passes through the initialization phase (the section* DECLS$[\![p]\!]_{\cdot\mathrm{decl}}$; INITS$[\![p]\!]_{\cdot\mathrm{decl}}$*) and the clearance phase (the section* CLEARS$[\![p]\!]_{\cdot\mathrm{decl}}$*) of each of the generated blocks*

Proof. This lemma is trivial with our simplified languages: the only way to skip the exit of a block is with the instruction `return`, and there is no way to skip the beginning of a block. The `return` statement is never used inside a generated block, and only used in the generated functions in the $(\!|f|\!)_{\cdot\mathrm{body}}$ translation. Since this is outside the blocks, it does not affect the control flow graph of the programs in the blocks. □

Lemma D.4. *The only variables that are used in the generated code* ${}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{code}}$ *are all specified in* ${}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}}$*, with matching types.*

Proof. By construction this invariant is satisfied. Formally it can be proved by induction on the predicate, proving a similar result for the terms. However, this induction is straightforward and not very insightful. Sec. G shows several proofs using this technique. □

Lemma D.5 (memory transparency of generated code). *At the beginning and at the end of each of the generated code blocks,* $\mathcal{M} = \perp$.

Proof. At the beginning of the program execution, we have $\mathcal{M} = \perp$. Since the mini-C language does not contain instruction whose semantics change $\mathcal{M}$, the only instructions that may change it are the mini-GMP ones generated by the tool. Hence, it suffices to show that if $\mathcal{M} = \perp$ at the entry point of a block, then $\mathcal{M} = \perp$ at the exit point of this block. Consider a block obtained by translating the assertion `assert p`. By Lemma D.4, all the variables used inside the block are contained in ${}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}}$. Using Theorem 6.1 before the call to CLEARS in Fig. 7 shows that at this point, for every value $z \in \mathrm{Mpz}$ such that $\mathcal{M}(z) \neq \perp$, there exists a unique variable v such that $\Omega(v) = z$. Since at the beginning of the block, $\mathcal{M} = \perp$, the same theorem implies that the variable v does not hold the value z at this point. Hence, the variable v is initialized inside the block to the value z inside the block. Hence $(v, \mathtt{mpz}) \in {}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}}$. So the clear instruction is called on the variable v at the end of the block, and thus $\mathcal{M}(z) = \perp$ at the exit point of the block. Since this holds for every z such that $\mathcal{M}(z) \neq \perp$ before the end, this proves that $\mathcal{M} = \perp$ at the exit point of the block. □

Theorem 6.2 (Absence of memory leak). *At the end of the program execution,* $\mathcal{M} = \perp$.

Proof. Lemma D.5 shows that throughout the program’s execution, $\mathcal{M} = \perp$ except inside the code blocks generated by the monitors. By Lemma D.3, the end of the execution is at the same point of the original program, outside of the generated code blocks, hence $\mathcal{M} = \perp$ at the end of the execution. □

E PROOFS OF THE SEMANTICS OF THE MACROS

We characterize the formal semantics of the macros defined in Sec. 4.3. In order to factorize the study of the semantics of the translation, we introduce an operator $\Omega, \mathcal{M} \vDash e \rightsquigarrow z$ which given an expression e in an environment $\Omega, \mathcal{M}$ returns the integer z that this expression represents, independently of which C type is used to represent the integer. The semantics of this operator is defined with the two following rules

$$\frac{\Omega \vDash e \Rightarrow x \qquad \mathrm{ty}(e) = \mathtt{int}}{\Omega, \mathcal{M} \vDash e \rightsquigarrow \dot{x}} \qquad \frac{\Omega \vDash e \Rightarrow x \qquad \mathrm{ty}(e) = \mathtt{mpz}}{\Omega, \mathcal{M} \vDash e \rightsquigarrow \mathcal{M}(x)}$$

Lemma E.1 (semantics of the `mpz`_ASSGN macro). *If the expression e represents the number z, then after the execution of the macro* `mpz`_ASSGN(v, e)*, the variable v contains the representation of the number z in the type* `mpz`*, while the rest of the memory is left unchanged. More precisely, The semantics of the* `mpz`_ASSGN *macro is characterized by the following admissible rule.*

$$\frac{\Omega, \mathcal{M} \vDash e \rightsquigarrow z \qquad \Omega_{\mathcal{V}}(v) = y \in Mpz}{\Omega, \mathcal{M} \vDash \mathtt{mpz}\_\mathit{ASSGN}(v, e) \Rightarrow \Omega, \mathcal{M}\{y \backslash z\}}$$

Proof. We proceed by case disjunction on the C type $\mathrm{ty}(e)$. When $\mathrm{ty}(e) = \mathtt{int}$ (resp. $\mathrm{ty}(e) = \mathtt{mpz}$), the macro `mpz`_ASSGN$(v, e)$ reduces to the single instruction `set_i`(v, e); (resp. `set_z`(v, e);). One can derive the conclusion $\Omega, \mathcal{M} \vDash \mathtt{mpz}\_\mathrm{ASSGN}(v, e) \Rightarrow \Omega, \mathcal{M}\{v \backslash\} z$ if and only if the rule defining the semantics of the statement `set_i` (resp. of the statement `set_z`), which is equivalent to having a derivation of $\Omega, \mathcal{M} \vDash e \Rightarrow x$ with $\dot{x} = z$ (resp. a derivation of $\Omega, \mathcal{M} \vDash e \Rightarrow x$ with $\mathcal{M}(e) = z$). This is equivalent to having a derivation of $\Omega, \mathcal{M} \vDash e \rightsquigarrow z$ in both cases. □

Lemma E.2 (semantics of the `int`_ASSGN macro). *If the expression e represents the number z representable in the type* `int`*, then after the execution of the macro* `int`_ASSGN(v, e)*, the variable v contains the representation of the number z in the type* `mpz`*, while the rest of the environment is left unchanged. More precisely, The semantics of the* `int`_ASSGN *macro is specified by the following admissible rule.*

$$\frac{\Omega(v) \in \mathit{Int} \cup \mathbb{U}_{\mathtt{int}} \qquad \Omega, \mathcal{M} \vDash e \rightsquigarrow z \qquad \mathrm{m}_{\mathtt{int}} \leq z \leq \mathrm{M}_{\mathtt{int}}}{\Omega, \mathcal{M} \vDash \mathtt{int}\_\mathit{ASSGN}(v, e) \Rightarrow \Omega\{v \backslash z^{int}\}, \mathcal{M}}$$

Proof. The proof is essentially the symmetrical to the one of Lemma E.1, by case disjunction on the type $\mathrm{ty}(e)$. In the case $\mathrm{ty}(e) = \mathtt{int}$, we use the equation $\dot{x}^{\mathrm{int}} = x$ to conclude. □

Lemma E.3 (semantics of the $\mathbb{Z}$_ASSGN macro). *After a call to the* $\mathbb{Z}$_ASSGN(τ_z, v, z) *macro, the variable v contains the representation in* τ_z *of the number z. More precisely, the following admissible rules characterize the semantics of the macro.*

$$\frac{\Omega(v) \in \mathit{Int} \cup \mathbb{U}_{\mathtt{int}} \qquad z \in \mathbb{Z} \qquad \tau_z \ = \ \mathtt{int} \qquad \mathrm{m}_{\mathtt{int}} \leq z \leq \mathrm{M}_{\mathtt{int}}}{\Omega,\ \mathcal{M} \vDash \mathbb{Z}\_\mathit{ASSGN}(\tau_z,\ v,\ z) \Rightarrow \Omega\{v \backslash z^{int}\},\ \mathcal{M}}$$

$$\frac{z \in \mathbb{Z} \qquad \tau_z = \texttt{mpz} \qquad \Omega(v) = y \in Mpz}{\Omega, \mathcal{M} \vDash \mathbb{Z}\_\textit{ASSGN}(\tau_z, v, z) \Rightarrow \Omega, \mathcal{M}\{y \backslash z\}}$$

Proof. We proceed by case disjunction on the type τ_z. In the case where $\tau_z = \texttt{int}$, the macro reduces to a single assignation and the result is exactly the rule that defines the semantics of the assignation. In the case where $\tau_z = \texttt{int}$, the macro reduces to a single instruction `set_s`, and the result is given by the rule that defines the semantics of this instruction. □

Lemma E.4 (semantics of the cmp macro). *After the execution of* $\textsc{cmp}(c, e_1, e_2, v_1, v_2)$*, the variable* c *contains the result of the comparison of* e_1 *and* e_2*, and everything else is unchanged, except potentially for the values associated to* v_1 *and* v_2 *in memory. More precisely, for* Ω *such that* $\Omega_{\mathcal{V}}(v_1) \neq \bot$ *and* $\Omega_{\mathcal{V}}(v_2) \neq \bot$*, there exists an* $\mathcal{M}'$ *such that for all* v *distinct from* v_1 *and* v_2*,* $\mathcal{M}(v) = \mathcal{M}'(v)$*, and the following admissible rule characterizes the semantics of the macro.*

$$\frac{\Omega(v) \in \mathit{Int} \cup \mathbb{U}_{\texttt{int}} \qquad \Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1 \qquad \Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2 \qquad z_1 < z_2}{\Omega, \mathcal{M} \vDash \textit{CMP}(c, e_1, e_2, v_1, v_2) \Rightarrow \Omega\{c \backslash a\}, \mathcal{M}'}$$

$$\textit{where } a = \begin{cases} 1 & \textit{when } z_1 < z_2 \\ 0 & \textit{when } z_1 = z_2 \\ -1 & \textit{when } z_1 > z_2 \end{cases}$$

Proof. We prove this by case disjunction on the types $\text{ty}(e_1)$ and $\text{ty}(e_2)$

- If we have both $\text{ty}(e_1) = \texttt{int}$ and $\text{ty}(e_2) = \texttt{int}$, then the premises $\Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1$ and $\Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2$ values $x_1, x_2 \in \text{Int}$, such that $\Omega, \mathcal{M} \vDash e_1 \Rightarrow x_1$ and $\Omega, \mathcal{M} \vDash e_2 \Rightarrow x_2$, with $\dot{x_1} = z_1$, and $\dot{x_2} = z_2$. The macro $\textsc{cmp}(c, e_1, e_2, v_1, v_2)$ reduces to the following piece of code

```
if (e_1 < e_2) c=-1;
else if (e_1 > e_2) c=1;
else c = 0;
```

  If $z_1 < z_2$, the rule defining the semantics of comparison expressions gives a derivation for $\Omega, \mathcal{M} \vDash e_1 < e_2 \Rightarrow 1$, and then the rule defining the semantics of **if** statements shows that the semantics of this bloc of code is the same as that of the assignation `c = -1`. This gives a derivation of $\Omega, \mathcal{M} \vDash \textsc{cmp}(c, e_1, e_2, v_1, v_2) \Rightarrow \Omega\{c \backslash -1\}, \mathcal{M}$. The same kind of reasoning using the definition of the semantics of comparison, **if** statements, and assignation, specifies that the semantics of the macro as desired when $z_1 > z_2$ and when $z_1 = z_2$. Conversely, since the semantics of each of the **if** statement and of the assignation are defined by a single rule in each of the three above cases, we can check that if this macro application has a semantics, it is necessarily given by this rule.

- Otherwise, the macro reduces to the following piece of code

```
mpz_assign(v_1,e_1);
mpz_assign(v_2,e_2);
c = cmp(v_1,v_2);
```

  Then, Lemma E.1 together with the rules defining the semantics of GMP statement $c = \texttt{cmp}(v_1, v_2)$; and the rule for concatenation of statements give the semantics of the macro. More precisely, it gives a derivation for the judgment

$$\Omega, \mathcal{M} \vDash \textsc{cmp}(c, e_1, e_2, v_1, v_2) \Rightarrow \Omega\{c \backslash a\}, \mathcal{M}'$$

  where $\mathcal{M}' = \mathcal{M}\{\Omega_{\mathcal{V}}(v_1) \backslash z_1\}\{\Omega_{\mathcal{V}}(v_2) \backslash z_2\}$. Conversely, we can check using the same techniques, that if this macro has a semantics, it is necessarily obtained by application of this rule.

□

Lemma E.5 (semantics of the ⋄_assgn macro). *After the execution of* $\diamond\_\textsc{assgn}((\tau, c), e_1, e_2, r, v_1, v_2)$*, the variable* c *contains the representation in the type* τ *of the operation* $\diamond$ *on* e_1 *and* e_2*, and everything else is unchanged, except potentially for the values associated to* v_1*,* v_2 r *in memory. More precisely, for* Ω *such that* $\Omega_{\mathcal{V}}(v_1) \neq \bot$*,* $\Omega_{\mathcal{V}}(v_2) \neq \bot$ *and* $\Omega_{\mathcal{V}}(r) \neq \bot$ *there exists an* $\mathcal{M}'$ *such that for all* v *distinct from* v_1*,* v_2 *and* r*,* $\mathcal{M}(v) = \mathcal{M}'(v)$*, and the following admissible rules characterize the semantics of this macro.*

$$\frac{\begin{array}{c}\Omega(v) \in \mathit{Int} \cup \mathbb{U}_{\texttt{int}} \\ \Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1 \qquad \Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2 \qquad \mathrm{m}_{\texttt{int}} \leq z_1 \diamond z_2 \leq \mathrm{M}_{\texttt{int}} \qquad \tau = \texttt{int}\end{array}}{\Omega, \mathcal{M} \vDash \diamond\_\textit{ASSGN}((\tau, c), e_1, e_2, r, v_1, v_2) \Rightarrow \Omega\{c \backslash (z_1 \diamond z_2)^{\mathit{int}}\}, \mathcal{M}'}$$

$$\frac{\Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1 \qquad \Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2 \qquad \tau = \texttt{mpz} \qquad \Omega_{\mathcal{V}}(c) = y \in Mpz}{\Omega, \mathcal{M} \vDash \diamond\_\textit{ASSGN}((\tau, c), e_1, e_2, r, v_1, v_2) \Rightarrow \Omega, \mathcal{M}'\{y \backslash z_1 \diamond z_2\}}$$

Proof. We proceed by case induction, following the form of the macro

- If $\tau = \texttt{int}$, $\text{ty}(e_1) = \texttt{int}$ and $\text{ty}(e_2) = \texttt{int}$ then the macro reduces to the following piece of code

$$c = e_1 \,\square\, e_2 \ ;$$

  The semantic rules for assignation and operation, then show that the following rule is admissible and any semantics for the macro reduces to an application of this rule.

$$\frac{\Omega \vDash e_1 \Rightarrow z_1 \qquad \Omega \vDash e_2 \Rightarrow z_2 \qquad \mathrm{m}_{\texttt{int}} \leq \dot{z_1} \diamond \dot{z_2} \leq \mathrm{M}_{\texttt{int}}}{\Omega, \mathcal{M} \vDash \diamond\_\textsc{assgn}((\tau, c), e_1, e_2, r, v_1, v_2) \Rightarrow \Omega\{c \backslash \dot{z_1} \diamond \dot{z_2}^{\texttt{int}}\}}$$

- If $\tau = \texttt{int}$ but $\text{ty}(e_1) \neq \texttt{int}$ or $\text{ty}(e_2) \neq \texttt{int}$ then the code of the macro reduces to
  $\texttt{mpz}\_\textsc{assgn}(v_1, e_1)$
  $\texttt{mpz}\_\textsc{assgn}(v_2, e_2)$
  $\texttt{op}(r, v_1, v_2)$;
  $\texttt{int}\_\textsc{assgn}(c, r)$
  Then Lemma E.1 together with the semantic rule for the op keyword and Lemma E.2 show that the semantics is completely characterized by the following rule

$$\frac{\Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1 \qquad \Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2 \qquad \mathrm{m}_{\texttt{int}} \leq z_1 \diamond z_2 \leq \mathrm{M}_{\texttt{int}}}{\Omega, \mathcal{M} \vDash \diamond\_\textsc{assgn}((\tau, c), e_1, e_2, r, v_1, v_2) \Rightarrow \Omega\{c \backslash z_1 \diamond {z_2}^{\texttt{int}}\}, \mathcal{M}'}$$

  with $\mathcal{M}' = \mathcal{M}\{\Omega_{\mathcal{V}}(v_1) \backslash z_1\}\{\Omega_{\mathcal{V}}(v_2) \backslash z_2\}\{\Omega_{\mathcal{V}}(r) \backslash z_1 \diamond z_2\}$
- If $\tau = \texttt{mpz}$ the code of the macro reduces to
  $\texttt{mpz}\_\textsc{assgn}(v_1, e_1)$
  $\texttt{mpz}\_\textsc{assgn}(v_2, e_2)$
  $\texttt{op}(c, v_1, v_2)$;
  Then Lemma E.1 together with the semantic rule for the op keyword and is completely characterized by the following rule

$$\frac{\Omega, \mathcal{M} \vDash e_1 \rightsquigarrow z_1 \qquad \Omega, \mathcal{M} \vDash e_2 \rightsquigarrow z_2 \qquad \Omega_{\mathcal{V}}(c)}{\Omega, \mathcal{M} \vDash \diamond\_\textsc{assgn}((\tau, c), e_1, e_2, r, v_1, v_2) \Rightarrow \Omega, \mathcal{M}'}$$

  with $\mathcal{M}' = \mathcal{M}\{\Omega_{\mathcal{V}}(v_1) \backslash z_1\}\{\Omega_{\mathcal{V}}(v_2) \backslash z_2\}\{y \backslash z_1 \diamond z_2\}$

□

F INVARIANTS FOR ROUTINE TRANSLATION

Useful notations for environments. Considering a environment Ω and a memory $\mathcal{M}$, we denote $(\Omega, \mathcal{M}) + +(\tau, v, z)$ the environment and memory obtained by adding a variable v representing z in the type τ.

- If $\tau = \mathtt{int}$ and $z \in \mathbb{Z}$ such that $\mathrm{m}_{\mathtt{int}} \leq z \leq \mathrm{M}_{\mathtt{int}}$ or $z \in \mathbb{U}_{\mathtt{int}}$, we define
$$(\Omega, \mathcal{M}) + +(\tau, v, z) = \Omega\{v \backslash z^{\mathrm{int}}\}, \mathcal{M}$$
- If $\tau = \mathtt{mpz}$ and $z \in \mathbb{Z}$, we pick an address $x \in \mathrm{Mpz}$ which is not in the image of Ω, and define
$$(\Omega, \mathcal{M}) + +(\tau, v, z) = \Omega\{v \backslash x\}, \mathcal{M}\{x \backslash z\}$$

If A is a set of such triples with all distinct variables, we denote $(\Omega, \mathcal{M}) + +A$ the result of applying this operation successively on all elements of A, which does not depend on the order in which we pick the elements.

The synchronicity invariant. We define the *Synchronicity of binders* invariant, which characterizes the environment for bindings throughout the translation. It states that the environment for binders always abstracts accurately the state of the logical part of the semantical environment. We do not prove this invariant here, but just define the statement. We say that Ω and Γ satisfy (I1) when the two following conditions are satisfied

$$\begin{cases} \mathrm{dom}\, \Omega_{\mathfrak{L}} = \mathrm{dom}\, \Gamma \\ \forall x \in \mathrm{dom}\, \Gamma, \min \Gamma_{\mathcal{I}}(x) \leq \Omega_{\mathfrak{L}}(x) \leq \max \Gamma_{\mathcal{I}}(x) \end{cases} \tag{I1}$$

The suitability invariant. We introduce another invariant, which concerns the environment Ψ and states that at any point, it only contains name of program functions that have a semantics that translate the logic function they model, under the assumption that they are called with arguments varying within the right intervals. Before introducing this invariant, we first formalize what we mean by this semantical condition.We consider the following pieces of data

- a logic function or a predicate f, with $\mathfrak{F}(f) = (v_1, \ldots, v_n; b)$ (or $\mathfrak{P}$ equals the same thing in the case of a predicate)
- a program function or procedure ϕ with $\mathcal{F}(\phi) = (x_1, \ldots, x_n; s)$ (or $\mathcal{P}(\phi) = (x_1, \ldots, x_n, x_{n+1}; s)$ in case of a procedure)
- an environment for bindings Γ

Consider a family of integers $z_1, \ldots, z_n, z$ such that $z_1 \in \Gamma_{\mathcal{I}}(v_1), \ldots, z_n \in \Gamma_{\mathcal{I}}(v_n)$. We denote

$$\Omega^f, \mathcal{M}^f = \begin{cases} (\bot, \bot) + +\{(\Theta(\Gamma_{\mathcal{I}}(v_i)), x_i, z_i) | i = 1 \ldots n\} & \text{if } \mathcal{T}(b, \Gamma_{\mathcal{I}}) = \mathtt{int} \\ (\bot, \bot) + +\{(\mathtt{mpz}, v_0, 0)\} \cup \{(\Theta(\Gamma_{\mathcal{I}}(v_i)), x_{i+1}, z_i) | i = 1 \ldots n\} & \text{if } \mathcal{T}(b, \Gamma_{\mathcal{I}}) = \mathtt{mpz} \end{cases}$$

We say that ϕ is *suitable* to represent f in Γ when for every integers $z_1, \ldots, z_n, z$ such that $z_1 \in \Gamma_{\mathcal{I}}(v_1), \ldots, z_n \in \Gamma_{\mathcal{I}}(v_n)$, one of the following equivalence is satisfied

- If $\mathcal{T}(b, \Gamma_{\mathcal{I}}) = \mathtt{int}$: ϕ is a function and there exists a derivation of $\bot\{v_1 \backslash z_1, \ldots, v_n \backslash z_n\} \vDash b \Rightarrow z$ if and only if there exists Ω such that there is a derivation of
$$\Omega^f, \mathcal{M}^f \vDash s \Rightarrow \Omega, \mathcal{M}^f$$
with $\Omega(\mathrm{res}_f) = z^{\mathrm{int}}$
- If $\mathcal{T}(b, \Gamma_{\mathcal{I}}) = \mathtt{mpz}$: ϕ is a procedure and there exists a derivation of $\bot\{v_1 \backslash z_1, \ldots, v_n \backslash z_n\} \vDash b \Rightarrow z$ with $z < \mathrm{m}_{\mathtt{int}}$ or $\mathrm{M}_{\mathtt{int}} < z$ if and only if there exists $\Omega, \mathcal{M}$ such that there is a derivation of
$$\Omega^f, \mathcal{M}^f \vDash s \Rightarrow \Omega, \mathcal{M}^f\{\Omega^f(x_1) \backslash z\}$$

We can now state our invariant for the environment by using this notion of suitability: We say that Ψ respects the suitability invariant (I2) if the following is true

$$\Psi(f, \Gamma) \neq \bot \implies \Psi(f, \Gamma) \text{ is suitable to represent } f \text{ in } \Gamma \tag{I2}$$

G PRESERVATION OF THE SEMANTICS

The objective of this section is to prove Theorem 6.3. The technical difficulty of this theorem mostly resides in three lemmas that we prove by mutual induction, and which characterize the semantics of the pieces of codes generated by the term and predicate translation and routine calls. The generated code needs to be evaluated in a semantic environments containing more program variables. In particular it needs to contain program variables that correspond to all the logical binders as well as all the program variables that are generated during the translation. For this reason, given a semantic environment Ω and an environment for bindings Γ that respect the synchronicity of binders (I1), we build $\Omega^\Gamma, \mathcal{M}^\Gamma$ as follows

$$\Omega^\Gamma, \mathcal{M}^\Gamma = (\Omega, \mathcal{M}) + +\{\Theta(\Gamma_{\mathcal{I}}(v)), v, \Omega_{\mathfrak{L}}(v) | v \in \mathrm{dom}\,(\Gamma)\}$$

The environment $\Omega^\Gamma, \mathcal{M}^\Gamma$ correspond to adding the representation a representation as program variables of the values of the logical binders as prescribed by Γ. We then extend this environment further, by building for a term or predicate t $\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t}$. For this we define, for $(\tau, v) \in {}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{decl}}$, the value z_v to be 0 if $\tau = \mathtt{mpz}$ or a fresh value in $\mathbb{U}_{\mathtt{int}}$ if $\tau = \mathtt{int}$. It is the value after declaration of a variable of type `int` or after declaration and initialization of a variable of type `mpz`. We then define

$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} = \Omega^t, \mathcal{M}^t + +\{(\tau, v, z_v) | (\tau, v) \in {}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{decl}}\}$$

This represent the minimum extension of the environment that lets us evaluate the code generated for a term or a predicate that evaluates in an environment Ω. It lets us ignore the declaration and initialization phases of each block at first. See Th. G.4 to a verification that this environment accurately models these phases.

Lemma G.1 (Semantics of term translation). *Consider a term t and two environments Ω, Γ that satisfy the synchronicity for the binders* (I1) *as well as a Ψ satisfying suitability* (I2)*. Then the judgment $\Omega \vDash t \Rightarrow z$ has a derivation if and only if there are exists $\Omega', \mathcal{M}'$ such that $\Omega^\Gamma \sqsubseteq \Omega'$, $\mathcal{M}^\Gamma \sqsubseteq \mathcal{M}'$ and the following judgment is derivable*

$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash {}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

When it is the case, the relation $\Omega', \mathcal{M}' \vDash {}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{res}} \rightsquigarrow z$, is satisfied, more specifically

$$\begin{cases} \Omega'_{\mathcal{V}}({}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{res}}) = z^{int} & \textit{if } \mathcal{T}(\Gamma, t) = \mathtt{int} \\ \mathcal{M}'(\Omega'({}^{\Gamma}_{\Psi}[\![t]\!]_{\cdot\mathrm{res}})) = z & \textit{if } \mathcal{T}(\Gamma, t) = \mathtt{mpz} \end{cases}$$

Proof. We proceed by induction on the term t. Verifying that in the recursive calls the environment also satisfy the invariants (I1) and (I2) is straightforward in most cases since the environments do not change. For the sake of simplicity, we omit this verification when it is immediate, and even omit the environments Γ and Ψ in our notation when they do not intervene. The induction is mutual with that of Lemmas G.2 and G.3.

- If $t = v$ is a mini-C variable (thus of type `int`), then we have $\llbracket t \rrbracket_{\cdot\mathrm{decl}} = \emptyset$, thus $\Omega^{,\Gamma,t} = \Omega^{\Gamma}$ and $\mathcal{M}^{\Gamma,t} = \mathcal{M}^{\Gamma}$. The generated code $\llbracket t \rrbracket = \texttt{skip;}$ then has the following semantics
$$\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash \texttt{skip;} \Rightarrow \Omega^{\Gamma}, \mathcal{M}^{\Gamma}$$
The term $v = \llbracket t \rrbracket_{\cdot\mathrm{res}}$ has a semantics in the mini-FSL language $\Omega, \mathcal{M} \vDash v \Rightarrow \dot{x}$ if and only if $\Omega^{\Gamma}_{\mathcal{V}}(v) = x$.
- If $t = z$ is an integer, then it always have a mini-FSL semantics, and we distinguish two sub-cases:
  - If $\mathcal{T}(t, \Gamma_{\mathcal{I}}) = \texttt{int}$, then $\llbracket t \rrbracket_{\cdot\mathrm{decl}} = \{(\llbracket t \rrbracket_{\cdot\mathrm{res}}, \texttt{int})\}$, thus we chose $u \in \mathbb{U}_{\texttt{int}}$ such that $\Omega^{\Gamma,t} = \Omega^{\Gamma}\{\llbracket t \rrbracket_{\cdot\mathrm{res}} \backslash u\}$ and $\mathcal{M}^{\Gamma,t} = \mathcal{M}^{\Gamma}$. Then Hypothesis 1 then ensures that $\mathrm{m}_{\texttt{int}} \leq z \leq \mathrm{M}_{\texttt{int}}$, and thus Lemma E.3 provides the following semantics
$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma} \vDash \llbracket t \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega^{\Gamma,t}\{\llbracket t \rrbracket_{\cdot\mathrm{res}} \backslash z^{\mathrm{int}}\}, \mathcal{M}^{\Gamma}$$
  - If $\mathcal{T}(t, \Gamma_{\mathcal{I}}) = \texttt{mpz}$, then there exist a fresh value $x \in \mathsf{Mpz}$ such that $\Omega^{\Gamma,t} = \Omega^{\Gamma}\{\llbracket t \rrbracket_{\cdot\mathrm{res}} \backslash x\}$ and $\mathcal{M}^{\Gamma,t} = \mathcal{M}^{\Gamma}\{x \backslash 0\}$. Lemma E.3 then provides the following semantics
$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash \llbracket t \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t}\{\Omega^{\Gamma,t}(\llbracket t \rrbracket_{\cdot\mathrm{res}}) \backslash z\}$$
- If $t = t_1 \diamond t_2$ is the application of an operation, then we have the following relation between the semantics environments
$$\begin{cases} \Omega^{\Gamma,t_1}, \mathcal{M}^{\Gamma,t_1} \sqsubseteq \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \\ \Omega^{\Gamma,t_2}, \mathcal{M}^{\Gamma,t_2} \sqsubseteq \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \\ \mathrm{dom}\,(\Omega^{\Gamma,t_1}) \cap \mathrm{dom}\,(\Omega^{\Gamma,t_2}) = \mathrm{dom}\,(\Omega^{\Gamma}) \\ \mathrm{dom}\,(\mathcal{M}^{\Gamma,t_1}) \cap \mathrm{dom}\,(\mathcal{M}^{\Gamma,t_2}) = \mathrm{dom}\,(\mathcal{M}^{\Gamma}) \end{cases}$$
Suppose that there is a semantics $\Omega \vDash t_1 \diamond t_2 \Rightarrow z$, then we necessarily have the semantics $\Omega \vDash t_1 \Rightarrow z_1$ and $\Omega \vDash t_2 \Rightarrow z_2$ with $z = z_1 \diamond z_2$. Then using induction and weakening provided by Lemma C.2.1, this shows that we have a semantics
$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$
with $\Omega^{\Gamma} \sqsubseteq \Omega'$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$, and $\Omega', \mathcal{M}' \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{res}} \rightsquigarrow z_1$. Then Lemma C.2.2 shows that we have $\Omega^{\Gamma,t_2} \sqsubseteq \Omega'$ and $\mathcal{M}^{\Gamma,t_2} \sqsubseteq \mathcal{M}'$. We again use induction with Lemma C.2.1, showing that we have a semantics
$$\Omega', \mathcal{M}' \vDash \llbracket t_2 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$
with $\Omega^{\Gamma} \sqsubseteq \Omega''$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}''$, and $\Omega'', \mathcal{M}'' \vDash \llbracket t_2 \rrbracket_{\cdot\mathrm{res}} \rightsquigarrow z_2$. Note that $\llbracket t_1 \rrbracket_{\cdot\mathrm{res}} \notin \mathrm{dom}\,(\Omega^{\Gamma,t_2})$ thus by Lemma C.2.2, we also have $\Omega'', \mathcal{M}'' \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{res}} \rightsquigarrow z_1$. Lemma E.3 then apply to show that we have a derivation of
$$\Omega'', \mathcal{M}'' \vDash \diamond\_\textsc{assgn}((\mathcal{T}(t_1 \diamond t_2, \Gamma_{\mathcal{I}}), \llbracket t_1 \diamond t_2 \rrbracket_{\cdot\mathrm{res}}), \llbracket t_1 \rrbracket_{\cdot\mathrm{res}}, \llbracket t_2 \rrbracket_{\cdot\mathrm{res}}, \bar{r}, \bar{v}_1, \bar{v}_2) \Rightarrow \Omega''', \mathcal{M}'''$$
with $\Omega^{\Gamma} \sqsubseteq \Omega''$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}''$, and the following evaluation $\Omega''', \mathcal{M}''' \vDash \llbracket t_1 \diamond t_2 \rrbracket_{\cdot\mathrm{res}} \rightsquigarrow z_1 \diamond z_2$ with the type given by $\mathcal{T}(t_1 \diamond t_2, \Gamma_{\mathcal{I}})$. The semantics of statement sequencing lets us conclude that we have the following
$$\Omega^{\Gamma,t_1 \diamond t_2}, \mathcal{M}^{\Gamma,t_1 \diamond t_2} \vDash \llbracket t_1 \diamond t_2 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega''', \mathcal{M}'''$$
with $\Omega''', \mathcal{M}'''$ satisfying the desired conditions.
Conversely, suppose that there is a semantics for the translated term in the mini-GMP language as follows
$$\Omega^{\Gamma,t_1 \diamond t_2}, \mathcal{M}^{\Gamma,t_1 \diamond t_2} \vDash \llbracket t_1 \diamond t_2 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega^{(2)}, \mathcal{M}^{(2)}$$
Then the semantics for sequencing shows that we then must have the two following derivations
$$\Omega^{\Gamma,t_1 \diamond t_1}, \mathcal{M}^{\Gamma,t_1 \diamond t_2} \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega^{(0)}, \mathcal{M}^{(0)}$$
$$\Omega^{(0)}, \mathcal{M}^{(0)} \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega^{(1)}, \mathcal{M}^{(1)}$$
By Lemma C.2.3, we get a semantics for $\llbracket t_1 \rrbracket_{\cdot\mathrm{code}}$ in $\Omega^{t_1}, \mathcal{M}^{t_1}$, thus by induction we get a semantics of the mini-FSL term t_1 in the environment Ω. This implies that $\Omega^{(0)} = \Omega'$ and $\mathcal{M}^{(0)} = \mathcal{M}'$, and thus $\Omega^{\Gamma,t_2} \sqsubseteq \Omega^{(0)}$ and $\mathcal{M}^{\Gamma,t_2} \sqsubseteq \Gamma^{(0)}$. Lemma C.2.3 again applies to show that t_2 has a semantics in Ω. The semantics of t_1 and t_2 give a semantics for $t_1 \diamond t_2$, and applying the opposite direction shows that $\Omega^{(1)} = \Omega''$, $\Omega^{(2)} = \Omega'''$ and $\mathcal{M}^{(1)} = \mathcal{M}''$, $\mathcal{M}^{(2)} = \mathcal{M}'''$.
- If $t = p ? t_1 : t_2$ is a conditional term, then we have the following relations
$$\begin{cases} \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \sqsubseteq \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \\ \Omega^{\Gamma,t_1}, \mathcal{M}^{\Gamma,t_1} \sqsubseteq \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \\ \Omega^{\Gamma,t_2}, \mathcal{M}^{\Gamma,t_2} \sqsubseteq \Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \\ \mathrm{dom}\,(\Omega^{\Gamma,p}) \cap \mathrm{dom}\,(\Omega^{\Gamma,t_1}) = \mathrm{dom}\,(\Omega^{\Gamma}) \\ \mathrm{dom}\,(\mathcal{M}^{\Gamma,p}) \cap \mathrm{dom}\,(\mathcal{M}^{\Gamma,t_1}) = \mathrm{dom}\,(\mathcal{M}^{\Gamma}) \\ \mathrm{dom}\,(\Omega^{\Gamma,p}) \cap \mathrm{dom}\,(\Omega^{\Gamma,t_2}) = \mathrm{dom}\,(\Omega^{\Gamma}) \\ \mathrm{dom}\,(\mathcal{M}^{\Gamma,p}) \cap \mathrm{dom}\,(\mathcal{M}^{\Gamma,t_2}) = \mathrm{dom}\,(\mathcal{M}^{\Gamma}) \end{cases}$$
This term has a semantics $\Omega \vDash t \Rightarrow z$ if and only if either of those two conditions are satisfied
(1) There is a semantics $\Omega \vDash p \Rightarrow 1$ together with a semantics $\Omega \vDash t_1 \Rightarrow z$. Then using the mutual induction case with Lemma G.2 together with the weakening of Lemma C.2.1 shows that the first condition gives a derivation of
$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash \llbracket p \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$
with $\Omega^{\Gamma} \sqsubseteq \Omega'$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$, and $\Omega'_t(\llbracket p \rrbracket_{\cdot\mathrm{res}}) = 1^{\mathrm{int}}$. Then Lemma C.2.2 shows that we have $\Omega^{\Gamma,t_1} \sqsubseteq \Omega'$ and $\mathcal{M}^{\Gamma,t_1} \sqsubseteq \mathcal{M}'$. Then induction with Lemma C.2.1 show that the second condition gives a derivation of
$$\Omega', \mathcal{M}' \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$
with $\Omega^{\Gamma} \sqsubseteq \Omega''$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}''$, and $\Omega''_t, \mathcal{M}''_t \vDash \llbracket t_1 \rrbracket_{\cdot\mathrm{res}} \rightsquigarrow z$. The semantics for **`if`** statements and of the $\mathcal{T}\_\textsc{assgn}(\Gamma, t)$ macro given by Lemma E.1 or E.2 then gives a derivation of
$$\Omega_t, \mathcal{M}_t \vDash \llbracket t \rrbracket_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$
(2) There is a semantics $\Omega \vDash p \Rightarrow 0$ together with a semantics $\Omega \vDash t_2 \Rightarrow z$: This is symmetrical to the previous case.
Conversely, the semantics of the **`if`** statements imply that if $\llbracket t \rrbracket_{\cdot\mathrm{code}}$ has a semantics, then it necessarily falls in either of those two cases, and using Lemma C.2.3 shows that there necessarily p has semantics 1 and t_1 has a semantics, or p has semantics 0 and t_2 has a semantics. In both cases, the proof for the forward direction apply to show that the initial

semantics of the translated code respects the semantics of the logical term.

- If $t = x$ is a logical binder, then the translation is given by $[\![x]\!]_{\cdot\text{code}} = \texttt{skip;}$, so we have the semantics

$$\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash \texttt{skip;} \Rightarrow \Omega^{\Gamma}, \mathcal{M}^{\Gamma}$$

The term $v = [\![t]\!]_{\cdot\text{res}}$ has a semantics in the mini-FSL language $\Omega, \mathcal{M} \vDash v \Rightarrow z$ if and only if $\Omega_{\mathfrak{L}}(v) = z$, which is equivalent to $\Gamma_{\mathcal{V}}(v) \neq \bot$ by (I1). This is by construction equivalent to $\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash \Gamma_{\mathcal{V}}(v) \rightsquigarrow z$.

- If $t = f(\kappa_1\ t_1, \cdots, \kappa_n\ t_n)$ is a call to a logic function, with $\mathfrak{F}(f) = (v_1, \ldots, v_n; b)$. Suppose that there is a semantics $\Omega \vDash f(\kappa_1\ t_1, \ldots, \kappa_n\ t_n) \Rightarrow z$, then we necessarily have semantics the following semantics

$$\Omega \vDash t_1 \Rightarrow z_1 \quad \ldots \quad \Omega \vDash t_n \Rightarrow z_n$$

then using successive inductions together with Lemmas C.2.1 and C.2.2, as well as the rule for sequencing, we show that there is a semantics

$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash [\![t_1]\!]_{\cdot\text{code}}; \ldots; [\![t_n]\!]_{\cdot\text{code}} \Rightarrow \Omega', \mathcal{M}'$$

with $\Omega^{\Gamma} \sqsubseteq \Omega'$, $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$ and $\Omega', \mathcal{M}' \vDash [\![t_i]\!]_{\cdot\text{res}} \rightsquigarrow z_i$. Let us now build the environment

$$\hat{\Gamma} = \bot\{x_1 \backslash (\bar{v}_1, \mathcal{I}(\Gamma_{\mathcal{I}}, t_1)), \ldots, x_n \backslash (\bar{v}_n \mathcal{I}(\Gamma_{\mathcal{I}}, t_n))\}$$
$$\hat{\Psi} = {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\text{env}}$$

By induction $\hat{\Psi}$ satisfy the invariant (I2). Moreover, since $v_1, \ldots, v_n \in \hat{\Gamma}$ Lemma G.3 shows that ${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)$ is suitable to represent f in $\hat{\Gamma}$. We then distinguish two cases:

  - If $\mathcal{T}(f(t_1, \ldots, t_n), \Gamma) = \texttt{int}$, then the translation ends with the following line of code

$$[\![f(t_1, \ldots, t_n)]\!]_{\cdot\text{res}} \ = \ {}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\text{name}}([\![t_1]\!]_{\cdot\text{res}}, \ldots, [\![t_n]\!]_{\cdot\text{res}})$$

The semantics of the logical term then gives a derivation

$$\bot\{v_1 \backslash z_1, \ldots, v_n \backslash z_n\} \vDash b \Rightarrow z$$

Note that we can chose addresses such that $\Omega^f \sqsubseteq \Omega'$ and $\mathcal{M}^f \sqsubseteq \mathcal{M}'$. The suitability of ${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)$ to represent f in $\hat{\Gamma}$ together with Lemma C.2.1, shows that, denoting

$$\mathcal{F}({}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\text{name}}) = (x_1, \ldots, x_n; s)$$

we have a derivation of the following semantics

$$\bot\{x_i \backslash \Omega'([\![t_i]\!]_{\cdot\text{res}})\}, \mathcal{M}' \vDash s \Rightarrow \Omega'', \mathcal{M}'$$

with $\Omega''(\text{res}_f) = z^{\text{int}}$. The semantics of function calls then shows that this gives a semantics

$$\Omega', \mathcal{M}' \vDash [\![t]\!]_{\cdot\text{res}} \ = \ {}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\text{name}}([\![t_1, \ldots, [\![t_n]\!]_{\cdot\text{res}}]\!]_{\cdot\text{res}}) \Rightarrow \Omega'\{[\![t]\!]_{\cdot\text{res}} \backslash z^{\text{int}}\}, \mathcal{M}'$$

  - If $\mathcal{T}(f(t_1, \ldots, t_n), \Gamma) = \texttt{mpz}$, then the translation ends with the following line of code

$${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\text{name}}([\![f(t_1, \ldots, t_n)]\!]_{\cdot\text{res}}, [\![t_1]\!]_{\cdot\text{res}}, \ldots, [\![t_n]\!]_{\cdot\text{res}})$$

The semantics of the logical term then gives a derivation

$$\bot\{v_1 \backslash z_1, \ldots, v_n \backslash z_n\} \vDash b \Rightarrow z$$

Note that we can chose addresses such that $\Omega^f \sqsubseteq \Omega'$ and $\mathcal{M}^f \sqsubseteq \mathcal{M}'$. The suitability of ${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)$ to represent f in $\hat{\Gamma}$ together with Lemma C.2.1, shows that, denoting

$$\mathcal{F}({}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\text{name}}) = (x_1, \ldots, x_n; s)$$

we have a derivation of the following semantics

$$\bot\{x_1 \backslash \Omega'[\![t]\!]_{\cdot\text{res}}\}\{x_{i+1} \backslash \Omega'([\![t_i]\!]_{\cdot\text{res}})\}, \mathcal{M}' \vDash s \Rightarrow \Omega'', \mathcal{M}'\{[\![t]\!]_{\cdot\text{res}} \backslash z\}$$

The semantics for procedure calls then implies that we have

$$\Omega', \mathcal{M}' \vDash t \Rightarrow \Omega', \mathcal{M}'\{[\![t]\!]_{\cdot\text{res}} \backslash z\}$$

Note that $[\![t]\!]_{\cdot\text{res}} \notin \text{dom}(\mathcal{M}^{\Gamma})$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$ by induction, so we have $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'\{[\![t]\!]_{\cdot\text{res}} \backslash z\}$.

Conversely, we can perform the same reasoning the other way around, distinguishing the two above cases, and using Lemma C.2.3, in order to show the equivalence.

This case is the only one where the environment Ψ gets modified, and a new program function is added. Lemma G.3 shows that this function is suitable, and thus the new environment Ψ still satisfies (I2). □

Lemma G.2 (Semantics of predicate translation). *Consider a predicate p and two environments Ω, Γ that satisfy the synchronicity for the binders* (I1) *as well as a Ψ satisfying suitability* (I2). *Then for $b \in \mathbb{B}$, the judgment $\Omega \vDash p \Rightarrow b$ has a derivation if and only if there are $\Omega^{\Gamma} \sqsubseteq \Omega'$ and $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$ such that the following judgment is derivable*

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash {}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\text{code}} \Rightarrow \Omega', \mathcal{M}'$$

When it is the case, we necessarily have $\Omega'_{\mathcal{V}}({}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\text{res}}) = b^{int}$.

Proof. We proceed by induction on the predicate p and mutual induction with Lemma G.1.

- If $p = \texttt{\textbackslash true}$ (or $p = \texttt{\textbackslash false}$) is a truth value: Since both cases are symmetric, we only prove for the case $p = \texttt{\textbackslash true}$. In that case, in any environment Ω, we have $\Omega \vDash p \Rightarrow 1$. Moreover, the piece of code ${}_{\Psi}[\![p]\!]_{\cdot\text{code}}$ reduces to $[\![p]\!]_{\cdot\text{res}} = \texttt{1;}$.Since $\Omega^{\Gamma,p}([\![p]\!]_{\cdot\text{res}}) \in \mathbb{U}_{\texttt{int}}$, the rule defining the semantics of the assignation statements and of the machine integer expressions then gives

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p]\!]_{\cdot\text{code}} \Rightarrow \Omega^{\Gamma,p}\{[\![p]\!]_{\cdot\text{res}} \backslash 1^{\text{int}}\}, \mathcal{M}^{\Gamma,p}$$

- If $p = t_1 \triangleleft t_2$ is a relation, then the environments satisfy the following relations

$$\begin{cases} \Omega^{\Gamma,t_1}, \mathcal{M}^{\Gamma,t_1} \sqsubseteq \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \\ \Omega^{\Gamma,t_2}, \mathcal{M}^{\Gamma,t_2} \sqsubseteq \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \\ \text{dom}(\Omega^{\Gamma,t_1}) \cap \text{dom}(\Omega^{\Gamma,t_2}) = \text{dom}(\Omega^{\Gamma}) \\ \text{dom}(\mathcal{M}^{\Gamma,t_1}) \cap \text{dom}(\mathcal{M}^{\Gamma,t_2}) = \text{dom}(\mathcal{M}^{\Gamma}) \end{cases}$$

The semantics $\Omega \vDash p \Rightarrow 1$ is derivable if and only if we have a derivation of $\Omega \vDash t_1 \Rightarrow z_1$ and $\Omega \vDash t_2 \Rightarrow z_2$ with $z_1 \triangleleft z_2$. By induction from Lemma G.1 and using weakening provided by Lemma C.2.1 and LemmaC.2.3, the former is equivalent to having a derivation of

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![t_1]\!]_{\cdot\text{code}} \Rightarrow \Omega', \mathcal{M}'$$

with $\Omega^\Gamma \sqsubseteq \Omega'$ and $\mathcal{M}^\Gamma \sqsubseteq \mathcal{M}'$ and $\Omega', \mathcal{M}' \vDash [\![t_1]\!]_{\cdot\mathrm{res}} \rightsquigarrow z_1$. Then Lemma C.2.2 shows that $\Omega^{\Gamma,t_2}, \mathcal{M}^{\Gamma,t_2} \sqsubseteq \Omega', \mathcal{M}'$. Lemma G.1 and using weakening provided by Lemma C.2.1 and LemmaC.2.3, then apply to show that the semantics of t_2 is equivalent to having a semantics

$$\Omega', \mathcal{M}' \vDash [\![t_2]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$

with $\Omega^\Gamma \sqsubseteq \Omega''$ and $\mathcal{M}^\Gamma \sqsubseteq \mathcal{M}''$ and $\Omega'', \mathcal{M}'' \vDash [\![t_2]\!]_{\cdot\mathrm{res}} \rightsquigarrow z_2$. Then Lemma C.2.2 shows that we also have $\Omega'', \mathcal{M}'' \vDash [\![t_1]\!]_{\cdot\mathrm{res}} \rightsquigarrow z_1$. By Lemma E.4, these two evaluations are equivalent to having the semantics

$$\Omega'', \mathcal{M}'' \vDash \textsc{cmp}([\![p]\!]_{\cdot\mathrm{res}}, [\![t_1]\!]_{\cdot\mathrm{res}}, [\![t_2]\!]_{\cdot\mathrm{res}}, \bar{v}_1, \bar{v}_2) \Rightarrow \Omega''\{[\![p]\!]_{\cdot\mathrm{res}} \backslash 1^{\mathsf{int}}\}, \mathcal{M}''$$

By statement sequencing, having a semantics $\Omega \vDash p \Rightarrow 1$ is thus equivalent to having a semantics

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega''\{[\![p]\!]_{\cdot\mathrm{res}} \backslash 1^{\mathsf{int}}\}, \mathcal{M}''$$

with $\Omega^\Gamma \sqsubseteq \Omega''\{[\![p]\!]_{\cdot\mathrm{res}} \backslash 1^{\mathsf{int}}\}$ and $\mathcal{M}^\Gamma \sqsubseteq \mathcal{M}''$. The case where the semantics of p evaluates to 0 is symmetric to this case.

- If $p = \,!p_1$ is a negation, then $\Omega^{\Gamma,p_1}, \mathcal{M}^{\Gamma,p_1} \sqsubseteq \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p}$. By induction, and Lemmas C.2 and C.2 there is a derivation $\Omega \vDash p_1 \Rightarrow 1$ if and only if there is a derivation of $\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p_1]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$ with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega', \mathcal{M}'$, and $\Omega', \mathcal{M}' \vDash [\![p_1]\!]_{\cdot\mathrm{res}} \rightsquigarrow 1^{\mathsf{int}}$. Then the semantics for the `if` statement and assignation shows that this is equivalent to a derivation

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![!p_1]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega'\{[\![p]\!]_{\cdot\mathrm{res}} \backslash 0^{\mathsf{int}}\}, \mathcal{M}'$$

  In a symmetric way, there is a semantics $\Omega \vDash p_1 \Rightarrow 0$ if and only if there is a semantics

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![!p_1]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega'\{[\![p]\!]_{\cdot\mathrm{res}} \backslash 1^{\mathsf{int}}\}, \mathcal{M}'$$

  with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega'\{[\![p_1]\!]_{\cdot\mathrm{res}} \backslash 0^{\mathsf{int}}\}, \mathcal{M}'$.
- If $p = p_1 \,|\,| p_2$ is a disjunction, then we have

$$\begin{cases} \Omega^{\Gamma,p_1}, \mathcal{M}^{\Gamma,p_1} \sqsubseteq \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \\ \Omega^{\Gamma,p_2}, \mathcal{M}^{\Gamma,p_2} \sqsubseteq \Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \\ \mathrm{dom}\,(\Omega^{\Gamma,p_1}) \cap \mathrm{dom}\,(\Omega^{\Gamma,p_2}) = \mathrm{dom}\,(\Omega^{\Gamma}) \\ \mathrm{dom}\,(\mathcal{M}^{\Gamma,p_1}) \cap \mathrm{dom}\,(\mathcal{M}^{\Gamma,p_2}) = \mathrm{dom}\,(\mathcal{M}^{\Gamma}) \end{cases}$$

  There is a derivation of $\Omega \vDash p \Rightarrow b$ if and only if either of these two cases is satisfied

  (1) There is a derivation of $\Omega \vDash p_1 \Rightarrow 1$, in which case $b = 1$. By induction and weakening (Lemma C.2.1), this implies that there is a derivation

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p_1]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

  with $\Omega' \vDash [\![p_1]\!]_{\cdot\mathrm{res}} \Rightarrow 1^{\mathsf{int}}$ and $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega', \mathcal{M}'$. Then the semantic rules for the `if` statements and the assignation give a semantics

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$

  with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega'', \mathcal{M}''$ and $\Omega''([\![p]\!]_{\cdot\mathrm{res}}) = 1^{\mathsf{int}}$.

  (2) There is a derivation of $\Omega \vDash p_1 \Rightarrow 0$ and of $\Omega \vDash p_2 \Rightarrow b$. By induction and weakening (Lemma C.2.1), this implies that we have a derivation

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p_1]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

  with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega', \mathcal{M}'$ and $\Omega' \vDash [\![p_1]\!]_{\cdot\mathrm{res}} \Rightarrow 0^{\mathsf{int}}$. Then Lemma C.2.2 shows that we have $\Omega^{\Gamma,p_2}, \mathcal{M}^{\Gamma,p_2} \sqsubseteq \Omega', \mathcal{M}'$. Then again, by induction and Lemma C.2.1, we show that we get a derivation of

$$\Omega', \mathcal{M}' \vDash [\![p_2]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega'', \mathcal{M}''$$

  with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega'', \mathcal{M}''$ and $\Omega'' \vDash {}_{\Psi}[\![p_2]\!]_{\cdot\mathrm{res}} \Rightarrow b^{\mathsf{int}}$. Then the semantic rules for the `if` statements and the assignation give a semantics

$$\Omega^{\Gamma,p}, \mathcal{M}^{\Gamma,p} \vDash [\![p]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega''\{[\![p]\!]_{\cdot\mathrm{res}} \backslash b^{\mathsf{int}}\}, \mathcal{M}''$$

  with $\Omega^\Gamma, \mathcal{M}^\Gamma \sqsubseteq \Omega''\mathcal{M}''$.

  Conversely, the semantics of the `if` statements and of the assignation together with induction, shows using Lemma C.2.3 that if the generated code has a semantics, then it falls in one of the three above cases.
- If $p = f(\kappa_1\ \ t_1,\ \ldots\ , \kappa_n\ \ t_n)$ is a predicate call, with $\mathfrak{P}(f) = (v_1, \ldots, v_n; b)$. We proceed in a similar way as for the case of functions for the term translation. Suppose that there is a semantics

$$\Omega \vDash f(\kappa_1\ t_1, \ldots, \kappa_n\ t_n) \Rightarrow z$$

  then we necessarily have semantics the following semantics

$$\Omega \vDash t_1 \Rightarrow z_1 \quad \ldots \quad \Omega \vDash t_n \Rightarrow z_n$$

  then using successive inductions together with Lemmas C.2.1 and C.2.2, as well as the rule for sequencing, we show that there is a semantics

$$\Omega^{\Gamma,t}, \mathcal{M}^{\Gamma,t} \vDash [\![t_1]\!]_{\cdot\mathrm{code}}; \ldots ; [\![t_n]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

  with $\Omega^\Gamma \sqsubseteq \Omega'$, $\mathcal{M}^\Gamma \sqsubseteq \mathcal{M}'$ and $\Omega', \mathcal{M}' \vDash [\![t_i]\!]_{\cdot\mathrm{res}} \rightsquigarrow z_i$. Let us now build the environment

$$\hat{\Gamma} = \bot\{x_1 \backslash (\bar{v}_1, \mathcal{I}(\Gamma_{\mathcal{I}}, t_1)), \ldots, x_n \backslash (\bar{v}_n \mathcal{I}(\Gamma_{\mathcal{I}}, t_n))\}$$
$$\hat{\Psi} = {}^{\Gamma}_{\Psi}[\![t_1]\!] \bullet \ldots \bullet [\![t_n]\!]_{\mathrm{env}}$$

  By induction $\hat{\Psi}$ satisfy the invariant (I2). Moreover, since $v_1, \ldots, v_n \in \hat{\Gamma}$ Lemma G.3 shows that ${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)$ is suitable to represent f in $\hat{\Gamma}$. The translation ends with the following line of code

$$[\![f(t_1, \ldots, t_n)]\!]_{\cdot\mathrm{res}} \ = \ {}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\mathrm{name}}([\![t_1]\!]_{\cdot\mathrm{res}}, \ldots, [\![t_n]\!]_{\cdot\mathrm{res}})$$

  The semantics of the logical term then gives a derivation

$$\bot\{v_1 \backslash z_1, \ldots, v_n \backslash z_n\} \vDash b \Rightarrow z$$

  Note that we can chose addresses such that $\Omega^f \sqsubseteq \Omega'$ and $\mathcal{M}^f \sqsubseteq \mathcal{M}'$. The suitability of ${}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)$ to represent f in $\hat{\Gamma}$ together with Lemma C.2.1, shows that, denoting

$$\mathcal{F}({}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\mathrm{name}}) = (x_1, \ldots, x_n; s)$$

  we have a derivation of the following semantics

$$\bot\{x_i \backslash \Omega'([\![t_i]\!]_{\cdot\mathrm{res}})\}, \mathcal{M}' \vDash s \Rightarrow \Omega'', \mathcal{M}'$$

  with $\Omega''(\mathrm{res}_f) = z^{\mathsf{int}}$. The semantics of function calls then shows that this gives a semantics

$$\Omega', \mathcal{M}' \vDash [\![p]\!]_{\cdot\mathrm{res}} \ = \ {}^{\hat{\Gamma}}_{\hat{\Psi}}(\!|f|\!)_{\cdot\mathrm{name}}([\![t_1, \ldots, [\![t_n]\!]_{\cdot\mathrm{res}}]\!]_{\cdot\mathrm{res}}) \Rightarrow \Omega'\{[\![p]\!]_{\cdot\mathrm{res}} \backslash z^{\mathsf{int}}\}, \mathcal{M}' \quad \square$$

Lemma G.3. *The function and procedure generation to translate logic functions and predicates generates a function that has the same semantics as the logic function, for a call with arguments that range in intervals as given by the environment $\Gamma_{\mathcal{I}}$. More precisely, for every logic function (resp. logic predicate) f with $\mathfrak{F}(f) = (v_1, \dots, v_n; b)$ (resp with $\mathfrak{P}(f) = (v_1, \dots, v_n; b)$) and every Γ such that* $\mathrm{dom}(\Gamma) = \{v_1, \dots v_n\}$*, and Ψ satisfying* (I2)*, the function ${}^{\Gamma}_{\Psi}(\!|f|\!)$ is suitable to represent f in Γ*

Proof. We prove this result by induction mutual with Lemma G.1 and Lemma G.2, and distinguish two cases

- If $\Psi(f, \Gamma) \neq \bot$, then this is simply given by the invariant (I2).
- If $\Psi(f, \Gamma) = \bot$, we define $\hat{\Psi} = \Psi\{(f, \Gamma) \backslash {}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\}$. The function is defined as in Fig. 11 and we distinguish two subcases:
  - If $\mathcal{T}(b, \Gamma_{\mathcal{I}}) =$ `int`, then the generated is as follows

$$\begin{array}{l}
\texttt{int}\ {}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\ (\Theta(\Gamma_{\mathcal{I}}(v_1))\ \Gamma_{\mathcal{V}}(v_1),\ \dots,\ \Theta(\Gamma_{\mathcal{I}}(v_n))\ \Gamma_{\mathcal{V}}(v_n))\{ \\
\quad \text{DECLS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad \text{INITS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}} \\
\quad \text{CLEARS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad \texttt{return}\ {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}\,; \\
\}
\end{array}$$

Consider a family of integer $z_i \in \Gamma_{\mathcal{I}}(v_i)$, we build the semantic environment $\Omega = \bot\{v_1 \backslash z_1, \dots, v_n \backslash z_n\}$. Note that by definition Ω and Γ satisfy the invariant (I1), thus we consider $\Omega^{\Gamma}, \mathcal{M}^{\Gamma} = \Omega^f, \mathcal{M}^f$. By definition of the environment, we have the following semantics:

$$\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash \text{DECLS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}; \text{INITS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \Rightarrow \Omega^{\Gamma,b}, \mathcal{M}^{\Gamma,b}$$

Applying Lemma G.1 by mutual induction shows that there is a semantics $\Omega \vDash b \Rightarrow z$ if and only if there is a semantics $\Omega^{\Gamma,b}, \mathcal{M}^{\Gamma,b} \vDash {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$ with $\Omega'({}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}) = z$ and $\Omega^{\Gamma} \sqsubseteq \Omega'$, $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$. Using the semantics for statement sequencing as well as the one for the `return` statement, this is equivalent to a semantics

$$\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash^{\Gamma}_{\hat{\Psi}} (\!|f|\!)_{\cdot\mathrm{body}} \Rightarrow \Omega'', \mathcal{M}''$$

with $\Omega''(\mathrm{res}\,{}_{\Gamma}(\!|\Psi|\!)_{\cdot\mathrm{name}} f) = z$. Note that by Lemma D.1, for each of the variable $(x, \texttt{mpz}) \in {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}$, we have $\Omega'(x) = \Omega^{\Gamma,b}(x)$, and thus the CLEARS call only clears the variables added in the INITS call. Since $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$, this entails that $\mathcal{M}'' = \mathcal{M}^{\Gamma}$.
  - If $\mathcal{T}(b, \Gamma_{\mathcal{I}}) =$ `mpz`: it is similar, but slightly more intricate. The generated function is as follows

$$\begin{array}{l}
\texttt{void}\ {}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\ (\texttt{mpz}\ {}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}},\ \Theta(\Gamma_{\mathcal{I}}(v_1))\ \Gamma_{\mathcal{V}}(v_1),\ \dots,\ \Theta(\Gamma_{\mathcal{I}}(v_n))\ \Gamma_{\mathcal{V}}(v_n))\{ \\
\quad \text{DECLS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad \text{INITS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}} \\
\quad \text{CLEARS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}} \\
\quad \texttt{set\_z(}{}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}},\ {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}}); \\
\}
\end{array}$$

Consider a family of integer $z_i \in \Gamma_{\mathcal{I}}(v_i)$, we build the semantic environment $\Omega = \bot\{v_1 \backslash z_1, \dots, v_n \backslash z_n\}$. Note that by definition Ω and Γ satisfy the invariant (I1), thus we consider $\Omega^{\Gamma}, \mathcal{M}^{\Gamma}$. Note that we have $\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \sqsubseteq \Omega^f, \mathcal{M}^f$. The same reasoning as the previous case, using the induction of Lemma G.1 shows that there is a semantics $\Omega \vDash b \Rightarrow z$ if and only if there is a semantics

$$\Omega^{\Gamma}, \mathcal{M}^{\Gamma} \vDash \text{DECLS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}; \text{INITS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}; {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

with $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$ and $\mathcal{M}'(\Omega'({}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}})) = z$. Moreover, $\mathrm{dom}(\Omega^f) - \mathrm{dom}(\Omega^{\Gamma}) = \{{}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}\}$ and by definition, it is a fresh variable, so it is not a variable in the original mini-C program nor a variable in the image of $\Gamma_{\mathcal{V}}$. Since those are the only variables generated by the term translation, it follows that ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}$ cannot appear in ${}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}}$. Thus Lemma C.2.3 shows that this is equivalent to having a semantics

$$\Omega^f, \mathcal{M}^f \vDash \text{DECLS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}; \text{INITS}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{decl}}; {}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}'$$

with $\mathcal{M}^{\Gamma} \sqsubseteq \mathcal{M}'$ and $\mathcal{M}'(\Omega'({}^{\Gamma}_{\hat{\Psi}}[\![b]\!]_{\cdot\mathrm{res}})) = z$. Using the semantics of sequencing, as well at that for the statements `set_z` and kwcl, we have that the latter is equivalent to a semantics

$$\Omega^f, \mathcal{M}^f \vDash^{\Gamma}_{\Psi} (\!|f|\!)_{\cdot\mathrm{body}} \Rightarrow \Omega'', \mathcal{M}^f\{\Omega^f({}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{res}}) \backslash z\} \qquad \square$$

In both cases, ${}^{\Gamma}_{\Psi}(\!|f|\!)$ is suitable to represent f in Γ, the environment ${}^{\Gamma}_{\Psi}(\!|f|\!)_{\mathrm{env}} = \Psi\{(f, \Gamma) \backslash {}^{\Gamma}_{\Psi}(\!|f|\!)_{\cdot\mathrm{name}}\}$ satisfies the invariant (I2).

The mutual induction performed in these three lemmas is not a priori well formed. Indeed, in the case of function calls, the lemma are applied to the body of the function which is not structurally a sub-term of the caller term. We can however notice that at each function call that requires generating a new function, the domain of the environment Ψ strictly increases. Since there are finitely many logic functions, and that by Hypothesis 2 each of the logic function can only be associated with finitely many environments for bindings, the size of this domain has an upper bound. This variant thus ensures the termination, and justify the mutual induction that we have performed.

Theorem G.4 (Soundness of assertion translation). *For every predicate p, the judgment $\Omega \vDash p \Rightarrow 1$ is derivable if and only if there exists an environment Ω' such the following judgment is derivable*

$$\Omega, \bot \vDash^{\Gamma}_{\Psi} [\![\texttt{/*@ assert p; */}]\!] \Rightarrow \Omega', \bot$$

Proof. Since we have already proven that after a generated we always have $\mathcal{M} = \bot$, the rule defining the semantics for the `assert` statements shows that there is a derivation of

$$\Omega, \bot \vDash^{\Gamma}_{\Psi} [\![\texttt{/*@ assert p; */}]\!] \Rightarrow \Omega', \bot$$

if and only if there is a derivation of

$$\Omega, \bot \vDash \text{DECLS}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}}; \text{INITS}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{decl}}; {}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{code}} \Rightarrow \Omega', \mathcal{M}$$

with a value $x \neq 0^{\mathsf{int}}$ and a derivation of $\Omega' \vDash {}^{\Gamma}_{\Psi}[\![p]\!]_{\cdot\mathrm{res}} \Rightarrow x$. Denote $\Omega_p, \mathcal{M}_p$ as defined in Lemma G.2, and Ω_0 the environment Ω_p with all values replaced by undefined values of the right type (in $\mathbb{U}_{\tau}$). Then the first two parts of the translation have the following semantics

$$\Omega, \bot \vDash \text{DECLS}[\![p]\!]_{\cdot\mathrm{decl}} \Rightarrow \Omega_0, \bot \quad \text{and} \quad \Omega_0, \bot \vDash \text{INITS}[\![p]\!]_{\cdot\mathrm{decl}} \Rightarrow \Omega_p, \mathcal{M}_p$$

Then the semantics of juxtaposition lets us apply Lemma G.2 exactly to show that the translation of the assertion has a semantics if and only if $p \vDash \Omega \Rightarrow 1$. □

Theorem G.5 (Transparency of assertion translation). *For every predicate, p, assume we have a derivation of the following judgment*

$$\Omega, \bot \vDash^{\Gamma}_{\Psi} [\![\texttt{/*@ assert p; */}]\!] \Rightarrow \Omega', \bot$$

then $\Omega \sqsubseteq \Omega'$

Proof. We proceed as in the proof of Theorem G.4 to show that Ω' is necessarily the Ω'_p defined in Lemma G.2, and we have $\Omega \sqsubseteq \Omega_p \sqsubseteq \Omega'_p$ by direct application of this Lemma. □

Theorem 6.3 (Correctness of code generation). *The generated program has a semantics if and only if the original program has one. In that case, the semantics of the generated program subsumes the one of the original program. More formally for a program P, there exists an* Ω *such that* $\bot, \bot \vDash P \Rightarrow \Omega, \bot$ *if and only if there exists an* Ω' *such that* $\bot, \bot \vDash [\![P]\!] \Rightarrow \Omega', \bot$. *If it is the case, then* $\Omega \sqsubseteq \Omega'$.

Proof. This is an immediate consequence of the successive application of Theorem G.4 and Theorem G.5 on each assertion of the program, together with Lemma C.2.1 to manage the fact that the environment of the translated program does not stay strictly the same as the one of the original at the corresponding point, but still always subsumes it. □