Verified interoperability between OCaml and C: Local roots for the Garbage Collector

GURVAN DEBAUSSART*, Université Paris-Saclay, Intern, Laboratoire Méthodes Formelles, France
ARMAËL GUÉNEAU, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Internship supervisor, France

1 INTRODUCTION

Programming languages are designed with very different goals in mind. Some languages (OCaml, Python, ...) are very good for higher-level programs where we can abstract some details of implementation like memory management, while some other languages (C, Ada, ...) are more adequate to implement program where performance is crucial. To be able to work with the best of both worlds, a solution is often to combine multiple languages within a same project, combined and linked under a single binary. This language interoperability strategy relies on a *foreign function interface* (FFI): this is the way a language exposes itself to the outside world, a specification on how to properly communicate.

In the UNIX world, it is often required to be able to communicate with C code to be able to perform system calls. A lot of popular libraries are also implemented in C. Most of language interoperability is therefore done between C and another language.

In this work, we are particularly interested in the OCaml and C FFI. Combining multiple languages together can be very tricky. An important difficulty come from the fact that OCaml memory is managed automatically, with a *garbage collector* (GC). Since the memory is shared across all languages we work with, we need to be careful about correctly communicating our memory needs to the garbage collector. In OCaml, this is done through a process called *rooting*, which is explained in Section 2. The FFI provides us different mechanisms to use these roots which can be more or less easy to use. The easiest and most commonly used mechanism is *local roots*, which are roots local to a function. Melocoton currently has no support for this mechanism, and instead only relies on program wide *global roots*. A presentation of the OCaml FFI and its associated difficulty can be found in Section 2.

Formal methods have been used to guarantee the correctness of a program for a long time, but reasoning on programs which combine multiple languages is a relatively new topic. Melocoton[1] is a very recently published state of the art framework to prove that a program written in the OCaml and C FFI is correct. It is implemented in the Iris[5] separation logic[8] framework. An overview is presented in Section 3, with more details being available in Johannes Hostert master's thesis [3].

The main selling point of the Iris framework is the clear separation between language semantics and the reasoning rule used in proofs to prove properties about them. Adding a new feature to Melocoton is therefore split into three parts:

(1) Extending the semantics of a language with the new features
(2) Extending the reasoning rules to use our new feature in proofs
(3) Proving in Coq that the reasoning rules correctly represent the underlying semantics

Authors' addresses: Gurvan Debaussart*, Université Paris-Saclay, Intern, Laboratoire Méthodes Formelles, France, gurvan.debaussart@universite-paris-saclay.fr; Armaël Guéneau, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Internship supervisor, 91190, Gif-sur-Yvette, France, armael.gueneau@inria.fr.

When extending the reasoning rules, an important difficulty is the preservation of language local reasoning. Since we are working of formalising multiple languages which need to work together, we want to be able to reason about a program in OCaml without having to think about C and the FFI. Likewise, we want to be able to prove properties of a purely C part of a program without having to take care of the FFI. We therefore need to carefully choose how we extend the reasoning rules to do so.

We present Melocoton-local, an extension of Melocoton with support for local roots. Code can be freely accessed at https://github.com/logsem/melocoton on various branches. The full contribution of this work consists of:

- Boxed integers, explained in Section 4. They are an extension of the ML language supported by Melocoton which allows the use of types in the standard library such as `Int64.t`, `Int32.t` or `NativeInt.t` or They have been merged into the `main` branch of Melocoton.
- Local C variables, explained in Section 5.1. They can be seen in the `c_local` branch.
- Local roots for the garbage collector, explained in Section 5.2. They can be seen in the `local_roots` branch.

2 BACKGROUND: INTEROPERABILITY BETWEEN OCAML AND C

2.1 Simple example

As an introduction to the OCaml FFI, we present `minitime`, a simplified form of the `gmtime` function from the `Unix` module of the standard library. The `gmtime` function takes a floating point value representing the time elapsed since the Unix Epoch, and returns a structure containing a human readable date and time.

Our simplified version takes as input an `Int64` and only returns the corresponding minute and hour. Its implementation can be found in Figure 1. It closely resembles the one of `Unix.gmtime` in the standard library.

Writing code in the OCaml and C requires three different kind of codes. We first have OCaml code, where we declare our type and the signature of the function we want to create. In our example, we have the type `time`, and an OCaml function `minitime`. We declare the function using the `external` keyword and associate it to the caml_minitime symbol. This symbol corresponds to the C function in `minitime.c`. It is written in the second kind of code, *glue code*. Like its name suggests, its goal is to glue code written in the two other languages, pure C and OCaml. It is only a *wrapper* around another external `minitime` pure C function which does the actual computations. This glue code has access to special functions and macros which are part of the OCaml FFI (`CAMLparam1`, `CAMllocal1`, `caml_alloc`, `Int64_val`, `CAMLreturn`, ...). We call these functions and macros *primitives* of the FFI. To have a better understanding of how they work, we first need to understand OCaml and C memory models.

2.2 Same memory, different models

When we work with language interoperability, very different languages need to be able to communicate to perform computations. The same data could be encoded in different ways from one language to another. This is the case with OCaml and C. At runtime, OCaml values are either integers or locations. A location is a pointer to a block managed by the GC. A block is a combination of a tag, which describe the block content, and arbitrary data.

Blocks are created with the `caml_alloc` function. This function takes two parameters: the size of the block and its tag. In `caml_minitime`, we use `caml_alloc` on line 4 to allocate the result structure. We allocate a block of size 2 to be able to store our two fields `min` and `hour`, and the tag 0 to say that the block represent a structure. A structure always has tag 0. We then update the block

```
 1 type time = {
 2     min:  int;
 3     hour: int;
 4 }
 5
 6 external minitime: Int64.t -> time = "caml_minitime"
```

minitime.ml

```
 1 value caml_minitime(value t) {
 2     CAMLparam1(t);
 3     CAMLlocal1(r);
 4     r = caml_alloc(2, 0);
 5
 6     time_t timer  = (time_t) Int64_val(t);
 7     struct tm *tm = minitime(&timer);
 8
 9     Store_field(r, 0, Val_int(tm->tm_min));
10     Store_field(r, 1, Val_int(tm->tm_hour));
11
12     CAMLreturn(r);
13 }
```

minitime.c

Fig. 1. Implementation of minitime

contents with `Store_field`. We cannot store the C integers directly. This is due to the fact that OCaml reserves the lowest significance bit to be able to tell the difference between a location and an integer at runtime. An OCaml integer is therefore a bit smaller than a C one. We thus need to convert them according to our needs. We use the `Val_int` primitive on line 8 and 9 to transforms a C integer into OCaml integers.

We use the `Int64_val` primitive on line 6 to fix a similar problem for long integers. An `Int64` value is represented at runtime by an immutable foreign block. Foreign blocks are a special kind of block which contains arbitrary data.

When working with different languages, we need to keep in mind that they operate on the same memory. Code written in one language can modify memory which is also used by another language, and vice versa. This is important when we consider memory which is automatically managed by a language. One language could invalidate some memory which is still in use by another language that hasn't properly cooperated on its memory usage.

This is a problem faced when interfacing OCaml and C. Blocks are managed by the OCaml garbage collector (GC). They can be moved around when it is running, for example when memory need to be allocated. Variables storing values represented at runtime with blocks are simply pointers. To keep them valid, they have to be updated when the block they point to is moved around by the GC. In our example, if the `t` pointer is not updated, it could be invalid after the `caml_alloc` on line 4. When we try to use it on line 6, we will have a memory corruption bug. Since the GC behavior is highly dependant on the current state of the program, its behavior can be quite hard to predict, making it very hard to debug.

The OCaml mechanism for keeping locations valid is called rooting. It is done on line 2 and 3 by the `CAMLparam` and `CAMLlocal` primitives. Like their name suggests, they are use respectively for rooting parameters and local variables. They are a special form of roots which are local to the function. On exit, we use the `CAMLreturn` primitive to unroot all of our local roots at once.

We could have used other primitives for rooting our values: global roots. Like their name suggests, global roots are global to the whole program. We can create a global root with the `caml_register_global_root` primitive. When we want to unroot them, we have to do it one variable at a time with the `caml_remove_global_root` primitives.

3 BACKGROUND: FORMALISING MULTI-LINGUAL INTEROPERABILITY WITH MELOCOTON

Interoperability between OCaml and C can be tricky, and its related bugs hard to find. To make sure that the programs we write are bug free, we can prove their correctness in Melocoton.

Melocoton is implemented in the Iris separation-logic framework. This framework has a particular methodology for making proofs. It distinguish clearly the operational semantics of languages and the reasoning rules we use to prove properties about them. Reasoning rules are expressed through separation-logic. A theorem of *adequacy* has to be proven so that we are sure the reasoning rules corresponds to the operational semantics, and that the properties we can prove with them are real properties of the program they represent.

In the context of language interoperability, Iris allows us to preserve *language local reasoning*. Each one of the language we use can have a distinct operational semantics and reasoning rules.

3.1 Operational semantics

Melocoton formalizes two languages λ_{ML} and λ_{C}. To allow λ_{ML} and λ_{C} to work together, we need to link the two languages together. We do so with a generic linking operator $\oplus$.

For any two languages λ_L, λ_R, this linking operator allows us to create a new language $\lambda_L \oplus \lambda_R$. Any expression e of this new language can be in two states. It is either executing in λ_L or in λ_R. When an external call is met, the execution continues in the other language and the resulting value is given back using continuations.

This last point may be a problem in some situations. The value given back by the execution of a λ_L function is a λ_L value. If we want to be able to use it back in a λ_R program, this λ_L value also has to be valid in λ_R.

This is a problem found with λ_{ML} and λ_{C}. Instead of using λ_{ML} directly, we first put our language into a wrapper $[-]_{FFI}$. This wrapper transforms values into their lower-level runtime memory representation. We can then create the language $[\lambda_{\mathsf{ML}}]_{FFI} \oplus \lambda_{\mathsf{C}}$.

The linking operator allows having a clear separation between language. Programs can be written purely in C or in the FFI. When we later want to prove properties on our programs, we can prove purely C parts without having to take care about the existence of the FFI.

The $[\lambda_{\mathsf{ML}}]_{FFI}$ wrapped language not only transforms values into their underlying memory representation but also models the entirety of the OCaml runtime. Its syntax and semantics are given in Figure 2. It notably contains the different primitives which are made available when using the FFI such as `Val_int`, and the current state of the garbage collector. The values lv of the wrapper are not directly C values, even though they could be mapped one to one to a C value w as either an integer n or a pointer a. This is because we need to account for the garbage collector. We don't care about the exact location of our variable in the memory, but only their *logical location*. They could be moved around, but as long as they are rooted we don't care. Logical locations are mapped to the memory using the address map θ. When performing an allocation using `CAMLalloc`, this logical map is non-deterministically changed to represent the possible movement of our blocks. We use the notation $\overline{\theta}$ to represent the map θ where keys are unchanged but values could be anything. Since all possible maps have to be valid, we have to rely on *demonic* non-determinism.

The wrapped language can be in two states: either executing ML code or wrapped C code. When executing ML code, the C memory mem still exists but is inaccessible.

$$\begin{array}{rcl}
lv \in \mathrm{LVal} & ::= & n \in \mathbb{N} \mid \gamma \in \mathrm{LLoc} \\
m \in \mathrm{Mutability} & ::= & \mathrm{mut} \mid \mathrm{imm} \\
b \in \mathrm{Block} & ::= & \mathrm{VBlock}(m, tag, lvs \in \mathrm{List}(\mathrm{LVal})) \mid \mathrm{Closure}(x_1, x_2, e) \mid \mathrm{Foreign}(w) \\
\zeta \in \mathrm{BlockStore} & \triangleq & \mathrm{LLoc} \rightharpoonup \mathrm{Block} \\
\chi \in \mathrm{LocMap} & \triangleq & \mathrm{LLoc} \rightharpoonup \mathrm{Loc} \\
\theta \in \mathrm{AddrMap} & \triangleq & \mathrm{LLoc} \rightharpoonup \mathrm{Addr} \\
rm \in \mathrm{RootsMap} & \triangleq & \mathrm{Addr} \rightharpoonup \mathrm{LVal} \\
\rho \in \mathrm{State} & ::= & \mathrm{ML}((\zeta, \chi, rm, mem), \sigma) \mid \mathrm{C}((\zeta, \chi, \theta, rs), mem)
\end{array}$$

$$\textsc{RegisterS}\quad \frac{a \notin rs \qquad rs' = rs \cup \{a\}}{\texttt{CAMLregister}(a), \mathrm{C}((\zeta, \chi, \theta, rs), mem) \rightarrow_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rs'), mem)}$$

$$\textsc{UnregisterS}\quad \frac{a \in rs \qquad rs' = rs \setminus \{a\}}{\texttt{CAMLunregister}(a), \mathrm{C}((\zeta, \chi, \theta, rs), mem) \rightarrow_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rs'), mem)}$$

$$\textsc{AllocS}\quad \frac{\theta' = \overline{\theta}[\gamma \mapsto a] \qquad \zeta' = \zeta[\gamma \mapsto \mathrm{VBlock}(mut, tag, [0, ..., 0])] \qquad \forall i \in \{0, ..., size\}, mem'[a +_l i] = 0}{\texttt{CAMLalloc}(tag, size), \mathrm{C}((\zeta, \chi, \theta, rs), mem) \rightarrow_{FFI} a, \mathrm{C}((\zeta', \chi', \theta', rs), mem')}$$

Fig. 2. Syntax, state and semantics of $[\lambda_{\mathsf{ML}}]_{FFI}$

3.2 Reasoning in Melocoton program logic: A blueprint for proving the correctness of `minitime`

We want to prove that the glue code of `minitime` is correct. We are only interested in proving that the execution *does not go wrong*, but the correctness of the returned value is not important here. A program does not go wrong if the rules imposed by the FFI are respected and there is no undefined behavior.

In our example, we mostly have to check that we are rooting locations correctly. This means that any value still in use after an allocation has to be rooted, and that we must unroot values that we don't need anymore. We also have to check that we are converting between values correctly.

We want to do a formal proof of correctness inside Melocoton. We do so in a language-specific separation logic **Iris**$_{[\mathsf{ML}]_{FFI}}$. Each language has its own separation logic. Just like we can write a pure program in a language without having to acknowledge the existence of other languages, we can prove properties on a pure program without having to acknowledge the existence of other languages.

Separation logic uses *resources* to track the underlying state of our programs. Each resource can be considered as providing a view of separate point the system. We can combine resources together to gather more informations on the state of our system. We can think about the resources which we will need to do our proof.

We first need a way to track values stored inside variables and, more generally, resources to track the current state of the memory. We use the notation $a \mapsto_C w$ to note that a C address a contains the value w in memory. Specifications to use these pointers are presented in Figure 3. Specifications uses a Hoare triple notation[2] in which every free variable is universally quantified. The topmost brackets represent a pre-condition and the bottommost a post-condition. They are expressed using separation logic: resources in the pre-condition are consumed. If we want to still be able to use a resource afterwards, we have to give the resource back in the post-condition.

$$\begin{array}{lcr} \{ & a \mapsto_C w & \} \\ & {*a} & \\ \{ & \mathsf{RET}(w).\ a \mapsto_C w & \} \end{array} \qquad \begin{array}{lcr} \{ & a \mapsto_C w & \} \\ & a \leftarrow w' & \\ \{ & \mathsf{RET}(0).\ a \mapsto_C w' & \} \end{array}$$

Fig. 3. Specifications for reading and writing in λ_C

The next thing we need is a way to track the state of the garbage collector and its roots. We use a token $\mathrm{GC}\ \theta$, parameterized by the current logical memory state θ. In our function code, we use two primitives `CAMLparam` and `CAMLlocal` to root local variables. They both take as a parameter a C pointer and root them. The rooting mechanism takes a valid OCaml value and assert that this value will remain valid as long as it is rooted. We can model this as an exchange: We are exchanging our weak C pointer to a block which we know is valid at the moment of the swap for a stronger root pointer which will remain valid until unrooted.

We know that a C value w is valid if there is a possible corresponding block level value lv in the current logical memory state θ. We use the following notation:

$$w \sim_\theta lv$$

We can also say that this block level representation lv could represent the OCaml value v with the notation:

$$w \sim_\theta lv \sim v$$

Note that the same runtime value could represent different OCaml values.

Once a pointer is rooted, the exact position of the pointed block in the memory can be abstracted away. We use a new resource $a \mapsto_{\mathrm{root}} lv$, and the associated specification presented in Figure 4.

$$\begin{array}{lcr} \{ & \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv & \} \\ & {*a} & \\ \{ & \mathsf{RET}(lv).\ \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv & \} \end{array} \qquad \begin{array}{lcr} \{ & \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv * \lceil w \sim_\theta lv' \rceil & \} \\ & a \leftarrow w & \\ \{ & \mathsf{RET}(0).\ \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv' & \} \end{array}$$

Fig. 4. Specification for reading and writing to root pointers in $[\lambda_{\mathrm{ML}}]_{FFI}$

Both primitives can be modeled as a single `CAMLregister` primitive. Inverting it gives us the `CAMLunregister` primitive. Specifications for both of these primitives can be found in Figure 5. We use the separating conjunction $A * B$ to mark the fact that A and B are separated assertions, which talk about separated part of our system. Logical fact which does not talk about any resources are presented using the notation $\lceil A \rceil$. They are said to be *pure* statements, which will always be true no matter the current state of the system.

Allocating new blocks is done through the `CAMLalloc` primitive. We present its specification in Figure 6. Allocating a new block modify the parameter θ of the GC token, transforming it into a

$$\begin{array}{lcr} \{ & \mathrm{GC}\ \theta * a \mapsto_C w * \lceil w \sim_\theta lv \rceil & \} \\ & \texttt{CAMLregister}(a) & \\ \{ & \mathsf{RET}(0).\ \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv & \} \end{array} \qquad \begin{array}{lcr} \{ & \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv & \} \\ & \texttt{CAMLunregister}(a) & \\ \{ & \exists w\ \mathsf{RET}(0).\ \mathrm{GC}\ \theta * a \mapsto_C w * \lceil w \sim_\theta lv \rceil & \} \end{array}$$

Fig. 5. Specification for the CAMLregister and CAMLunregister primitive

$$\begin{array}{lcr} \{ & \mathrm{GC}\ \theta * a \mapsto_C w * \lceil w \sim_\theta lv \rceil & \} \\ & \texttt{CAMLregister}(a) & \\ \{ & \mathsf{RET}(0).\ \mathrm{GC}\ \theta * a \mapsto_{\mathrm{root}} lv & \} \end{array}$$

$$\begin{array}{lcr} \{ & \mathrm{GC}\ \theta & \} \\ & \texttt{CAMLalloc}(tag, size) & \\ \{ & \exists \theta', \gamma, w\ \mathsf{RET}(w).\ \mathrm{GC}\ \theta' * \gamma \mapsto_{\mathrm{blk}} \mathrm{VBlock}(tag, [0, \ldots 0]) * \lceil w \sim_\theta lv \rceil & \} \end{array}$$

Fig. 6. Specification for the CAMLalloc

new state θ'. All facts of the form $w \sim_\theta lv$ are therefore made useless, since we cannot use them together with the token GC θ'.

We cannot express the specification of `caml_minitime` in Melocoton. This is due to the fact that `caml_minitime` takes as a parameter an `Int64` value, which is not implemented in Melocoton. We will see how to do it in Melocoton-local instead.

4 ADDING BOXED INTEGERS TO MELOCOTON

The first thing we need which is missing from Melocoton is `Int64` values. In OCaml, `Int64`, `Int32` and `NativeInt` are *boxed*. This means that they are represented at runtime as immutable foreign blocks. We must box them because they cannot fit inside a standard OCaml integers, due to the reserved bit mentioned in Section 2. In this section, we present how we add boxed integers to Melocoton-local.

$$\begin{array}{rcl} v \in \mathrm{Val} & ::= & n \in \mathbb{N} \mid \ell \in \mathrm{Loc} \mid f \in \texttt{rec}\ f\ x.\ e \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \texttt{()} \mid \langle v, v \rangle \\ e \in \mathrm{Expr} & ::= & v \mid x \in \mathrm{Var} \mid \langle e, e \rangle \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \bullet e \mid e_0 \otimes e_1 \mid e_0(\vec{e}) \\ & & \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\ \sigma \in \mathrm{State} & \triangleq & \mathrm{Loc} \rightharpoonup v \uplus \lightning \end{array}$$

Fig. 7. Syntax and state of λ_{ML}

The λ_{ML} language is described in Figure 7. It is a standard expression based functional programming language. We extend $\lambda_{\mathsf{ML\_local}}$ from λ_{ML} with a new kind of values $\boxed{n \in \mathbb{N}}$ representing boxed integers. We also extend the semantics of the various operators to work on boxed integers. We use boxed integers in $\lambda_{\mathsf{ML\_local}}$ exactly the same way we would use regular integers. Since both are represented using natural numbers, they can both hold numbers of any size. The main difference is reasoning rules about them, and which primitive we need to use in our programs.

We then need to extend the wrapper to model the runtime representation of our new values. Boxed integers are represented at runtime as *immutable foreign blocks*. While $[\lambda_{\mathsf{ML}}]_{FFI}$ contains a notion of foreign blocks, they can only be considered mutable. We therefore extend them in Melocoton-local with the notion of mutability, as presented in Figure 8.

$$b \in \text{Block} ::= \text{VBlock}(m, tag, lvs \in \text{List}(\text{LVal})) \mid \text{Closure}(x_1, x_2, e) \mid \text{Foreign}(m, w)$$

Fig. 8. Extension of blocks in Figure 2 for Melocoton-local

We can then extend the ml representation inductive:

$$\zeta[\gamma] = \text{Foreign}(imm, n) \Rightarrow \gamma \sim \boxed{n}$$

Lastly, we need to be able to actually use our BoxedInt within C code. We need to be able to read and create them. This first need can be satisfied by using the `CAMLreadforeign` primitive. To create them, we can manually allocate a new foreign block with `CAMLalloccustom`, set it's content with `CAMLwriteforeign` and then return it. All of these primitives are already part of the semantics of Melocoton.

The second step in adding a new feature to Melocoton is extending the language-specific separation logic. We therefore extend $\textbf{Iris}_{[\texttt{ML}]_{FFI}}$ with new reasoning rules on boxed integers.

As mentioned, boxed integers are represented at runtime with foreign *immutable* blocks. We need to correctly respect the semantics associated with immutability in OCaml. This means that, when we expose a foreign block from C to OCaml, we need to provide a guarantee that we won't change the content of the block. We implement it in our separation-logic by exchanging our strong pointer to a block for a weaker one which doesn't allow modifications. This rule is not a specification, but only an exchange. We can use it whenever we want in a proof, and we don't have to insert a particular freezing expression in our program. It is presented with the separation-logic implication $A \twoheadrightarrow B$, which means we can exchange the resource A against the resource B. Specifications and rules associated to immutability and foreign blocks are presented in Figure 9. They use a new resource $\gamma \mapsto_{\text{foreign}[\text{m}]} w$ to express the fact that a logical location γ points to a foreign block with mutability m containing the C value w.

$$\text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{mut}]} w \twoheadrightarrow \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{imm}]} w$$

$$\begin{array}{c} \{ \quad \text{GC } \theta \quad \} \\ \texttt{CAMLalloccustom}() \\ \{ \quad \exists a, \texttt{RET}(\gamma).\ \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{mut}]} w \quad \} \end{array} \qquad \begin{array}{c} \{ \quad \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{m}]} w \quad \} \\ \texttt{CAMLreadforeign}(\gamma) \\ \{ \quad \texttt{RET}(w).\ \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{m}]} w \quad \} \end{array}$$

$$\begin{array}{c} \{ \quad \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{mut}]} \_ \quad \} \\ \texttt{CAMLwriteforeign}(\gamma, a) \\ \{ \quad \texttt{RET}(w').\ \text{GC } \theta * \gamma \mapsto_{\text{foreign}[\text{mut}]} a \quad \} \end{array}$$

Fig. 9. Specifications for interacting with foreign blocks in Melocoton-local

With these new resources, we can finally propose a first specification of `caml_minitime` in Figure 10. However, we will not be able to do the full proof of correction, because the body of the function uses primitives related to local roots which are unimplemented in Melocoton.

5 ADDING LOCAL ROOTS FOR THE GARBAGE COLLECTOR

We want to add local roots to Melocoton-local. Local roots relies on the use of local C variables, which are both unrooted and freed at function exit. We therefore start by adding local C variables to Melocoton-local.

$$\begin{array}{lcr}\{ & \mathrm{GC}\ \theta * \lceil\, lv \sim \boxed{n}\, \rceil & \} \\ & \texttt{caml\_minitime}(lv) & \\ \{ & \exists\, lv'\ v_1\ v_2, \mathtt{RET}(lv').\ \mathrm{GC}\ \theta' * \lceil\, lv' \sim \langle v_1, v_2\rangle\, \rceil & \}\end{array}$$

Fig. 10. First proposal for a caml_minitime specification

5.1 Local variables

$$\begin{array}{rcl}
\bullet \in \mathrm{Uop} & ::= & - \mid \neg \mid \sim \mid \texttt{(int)} \mid \texttt{(void*)} \\
\otimes \in \mathrm{Bop} & ::= & + \mid - \mid \times \mid \div \mid \% \mid \& \mid \mid \mid \hat{} \mid \ll \mid \gg \mid \le \mid < \mid = \mid +_l \mid -_l \\
w \in \mathrm{Word} & ::= & n \in \mathbb{N} \mid a \in \mathrm{Addr} \mid f \in \mathrm{Fun} \\
e \in \mathrm{Expr} & ::= & w \mid x \in \mathrm{Var} \mid \texttt{let}\ x = e_0\ \texttt{in}\ e_1 \mid *e \mid e_0 \leftarrow e_1 \mid \texttt{malloc}(e) \mid \texttt{free}(e_0, e_1) \\
 & & \mid \bullet e \mid e_0 \otimes e_1 \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ e_0\ \texttt{do}\ e_1 \mid e_0(\vec{e}) \\
\texttt{mem} \in \mathrm{State} & \triangleq & \mathrm{Addr} \rightharpoonup w \uplus \{ ↯, \star \}
\end{array}$$

$$\textsc{LetS}\quad \frac{}{\texttt{let}\ x = w\ \texttt{in}\ e, \texttt{mem} \longrightarrow_C e[x \leftarrow w], \texttt{mem}} \qquad \textsc{StoreS}\quad \frac{a \in \mathrm{dom}(\texttt{mem})}{a \leftarrow w, \texttt{mem} \longrightarrow_C 0, \texttt{mem}[a \mapsto w_1]}$$

$$\textsc{MallocS}\quad \frac{\forall_{i \in \{0,\ldots,n\}}, a +_l i \notin \mathrm{dom}(\texttt{mem})}{\texttt{malloc}(n), \texttt{mem} \longrightarrow_C a, \texttt{mem}[a \mapsto \star]} \qquad \textsc{FreeS}\quad \frac{\forall_{i \in \{0,\ldots,n\}}, \sigma[a +_l i] \in \mathrm{Word} \uplus \{\star\}}{\texttt{free}(a, n), \texttt{mem} \longrightarrow_C 0, \texttt{mem}[\forall_{i \in \{0,\ldots,n\}} a +_l i \mapsto ↯]}$$

$$\textsc{LoadS}\quad \frac{a \in \mathrm{dom}(\texttt{mem})}{*a, \texttt{mem} \longrightarrow_C \texttt{mem}[a], \texttt{mem}}$$

Fig. 11. Syntax, state and semantics of λ_C

We want to extend λ_C to a new language λ_{C_local} with proper support for local variables. The λ_C language is a standard expression based language. Its syntax, state and partial semantics are presented in Figure 11. When we want to use a local variable, we can use two different constructions.

If we want to use a simple variable which will only be initialised once and never modified, we can use a `let` binding. Variables which need to be modified after their creations have to be implemented by the user with a memory allocation at the beginning of the function and a free on exit. Memory allocations are done through the malloc(n) expression. It takes as a parameter an integer value n representing the size of the memory to allocate and return a pointer a to a new memory area. We can use the pointer offset operator $+_l$ to access each memory cell from the a pointer. When freshly allocated, cells are marked allocated but uninitialised with the star symbol $\star$.

When we don't have any more use for our memory, we can free it using the free(a, n) expression. The memory cells are marked freed with the lightning symbol ↯, and cannot be used anymore.

If we want to modify a variable content with a new value w, we have to update the corresponding memory a with a store $a \leftarrow w$. We can read back the value with a memory load $*a$.

This process is very cumbersome. It has two main drawbacks:

(1) We have to manually insert free and allocations. It is possible to forget to free a local C variable and get a correct C program which would be different from the one we want to prove.
(2) Having to do one allocation per each local variable can make our proofs harder to write and to check in an automated prover like Coq.

We start by fixing the first drawback by automating the memory allocation and freeing for local variables. We proceed by doing a simple syntactic check for possibly mutated local variables when we enter a function. A variable is considered possibly mutated if we take its address using the unary operator &, which is new in $\lambda_{\texttt{C\_local}}$. The C instruction $x = y$, where x is a variable name and y an expression, is translated into λ_{C} as $\&x \leftarrow y$.

For each possibly mutated variable found, we simply insert the corresponding `malloc` and `free`. If a variable is not mutated, we use a simple let bindings.

Fixing the second drawback can be similar to encoding a compiler optimisation inside of our semantics. A C compiler will insert a single allocation for a memory area which can fit every local variables. The pointer to this memory area is called the frame pointer, or fp. Each local variable will then be mapped to a unique offset from the frame pointer. When we exit the function, we can free all of the memory taken up by local variables by freeing the frame pointer.

The frame allocation is handled by a new expression $\texttt{alloc\_frame}(f)\ \texttt{in e}$, whose semantics is presented in Figure 12. It takes as a parameter a frame `f`, which is the list of variable that should be allocated, and an expression `e` in which the frame should be allocated.

$$\textsc{AllocFrameS}\quad \frac{e' = e[\&x_i \leftarrow fp +_l i, x_i \leftarrow \star(fp +_l i)] \qquad \forall_{\mathtt{i}\in\{0,\ldots,\mathtt{n}\}}, (fp +_l \mathtt{i}) \notin \mathrm{dom}(\mathtt{mem})}{\texttt{alloc\_frame}(\vec{x})\ \texttt{in e}, \texttt{mem} \longrightarrow_C e'; \texttt{free}(fp, n), \texttt{mem}[fp +_l i \mapsto \star]}$$

Fig. 12. Semantics of alloc_frame in $\lambda_{\texttt{C\_local}}$

This expression is not meant to be used directly by the user, but is inserted automatically on each function call by the aforementioned syntactic check. We have to be a bit careful. When doing the syntactic check, we must assert that we are not allowing incorrect behavior. We have three things to check:

(1) No variables should share the same name. This would cause a problem of capture in the substitution of addresses.
(2) A variable shouldn't be used before it is declared.
(3) There should be no frame allocation in the body of the function.

An advantage of modeling local variables the way we did is that we don't have to extend the separation logic **Iris**$_{\text{C}}$ at all. Existing reasoning mechanisms are enough to fully use them. The only thing we need is a rule `wp_allocframe` for the weakest precondition of our new expression.

5.2 Operational semantics of local roots

With these new C local variables in our hand, we now have all the cards required to add our local roots. In Melocoton, global roots are tracked as a set of addresses rs. We want to be able to associate to each function frame the roots they track, while being able to quickly remove the set containing the roots of the last frame when doing a return. The natural data structure to represent frames is therefore a simple list. In Melocoton-local, we therefore represent roots as a list of set of addresses rss, which we call the root stack frame. Each element of the list corresponds to a different frame, with the top-most frame at the beginning of the list. Allocating a new frame is done with the new `CAMLinitlocal` primitive. It simply pushes an empty set on top of the root stack frame. We can register a new address `a` to the top-most frame with `CAMLregisterlocal(a)`. When exiting a function, we can free the top-most frame with `CAMLunregisterlocal`.

The last element of the roots list correspond to global roots. We consider them as a special form of local roots which live in a function frame only exited on program exit. This allows treating roots in a simple uniform way. We therefore also need to change the global rooting mechanism. The `CAMLregister` and `CAMLunregister` primitives simply always act on the last element of the root list.

The semantics for each one of our new primitives and the extended state of the wrapper can be found in Figure 13.

$$rms \in \mathrm{List}(RootsMap) \qquad\qquad rss \in \mathrm{List}(RootsSet)$$

$$\rho \in \mathrm{State} \triangleq \mathrm{ML}((\zeta, \chi, rms, mem), \sigma) \mid \mathrm{C}((\zeta, \chi, \theta, rss), mem)$$

RegisterS
$$\frac{rss = locals +\!\!+ [global] \qquad a \notin globals \qquad rss' = locals +\!\!+ [global \cup \{a\}]}{\texttt{CAMLregister}(a), \mathrm{C}((\zeta, \chi, \theta, rss), mem) \to_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rss'), mem)}$$

UnregisterS
$$\frac{rss = locals +\!\!+ [global] \qquad a \in globals \qquad rss' = locals +\!\!+ [global \setminus \{a\}]}{\texttt{CAMLunregister}(a), \mathrm{C}((\zeta, \chi, \theta, rss), mem) \to_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rss'), mem)}$$

InitLocalS
$$\frac{}{\texttt{CAMLinitlocal}(), \mathrm{C}((\zeta, \chi, \theta, rss), mem) \to_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, \emptyset :: rss), mem)}$$

RegisterLocalS
$$\frac{rss = local :: other \qquad a \notin local \qquad rss' = (local \cup \{a\}){+\!\!+}other}{\texttt{CAMLregisterlocal}(a), \mathrm{C}((\zeta, \chi, \theta, rss), mem) \to_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rss'), mem)}$$

UnregisterLocalS
$$\frac{rss = local :: other \qquad rss' = other}{\texttt{CAMLunregisterlocal}(), \mathrm{C}((\zeta, \chi, \theta, rss), mem) \to_{FFI} 0, \mathrm{C}((\zeta, \chi, \theta, rss'), mem)}$$

Fig. 13. Extended state and semantics of Melocoton-local

5.3 Resources for local roots

We introduce three new resources to use our local roots:

(1) $a \mapsto_{\mathrm{root}[\mathtt{f}]} lv$, a generalisation of roots pointer parameterized by the frame `f` in which they live.
(2) `local_roots f` rs, associating a frame f to its tracked roots rs.
(3) `current_fc fc`, tracking the frames currently alive At the highest level, we track which frames are currently alive with the `current_fc fc` resource. It is parameterized by a list of frame `fc`. We use this resources when unregistering local roots to know which frame to destroy.

The specification for new protocols is presented in Figure 14 using these new resources, along with reasoning rules for using our new local root pointers. Since the `local_roots f` rs resources already track the full set of variable in the frame, the user does not have to give back his roots pointers when unregistering local roots. To make sure he will not be able to use his roots pointer after unregistering them, we request the local roots resources when he needs to use them.

$$\begin{array}{lcr}
\{ & \mathsf{GC}\ \theta * a \mapsto_{\mathrm{root}[\mathtt{f}]} lv * \mathtt{local\_roots\ f}\ rs & \} \\
 & {*}a & \\
\{ & \mathsf{RET}(lv).\ \mathsf{GC}\ \theta * a \mapsto_{\mathrm{root}[\mathtt{f}]} lv * \mathtt{local\_roots\ f}\ rs & \}
\end{array}$$

$$\begin{array}{lcr}
\{ & \mathsf{GC}\ \theta * a \mapsto_{\mathrm{root}[\mathtt{f}]} lv * \mathtt{local\_roots\ f}\ rs * \lceil w \sim_\theta lv' \rceil & \} \\
 & a \leftarrow w & \\
\{ & \mathsf{RET}(0).\ \mathsf{GC}\ \theta * a \mapsto_{\mathrm{root}[\mathtt{f}]} lv' * \mathtt{local\_roots\ f}\ rs & \}
\end{array}$$

$$\begin{array}{lcr}
\{ & \mathsf{GC}\ \theta * \mathtt{current\_fc\ fc} & \} \\
 & \mathtt{CAMLinitlocal}() & \\
\{ & \mathsf{GC}\ \theta * \mathtt{current\_fc\ f::fc} * \mathtt{local\_roots\ f}\ \emptyset & \}
\end{array}$$

$$\begin{array}{lcr}
\{ & \mathsf{GC}\ \theta * a \mapsto_C w * \mathtt{current\_fc\ f::fc} * \mathtt{local\_roots\ f}\ rs * \lceil w \sim_\theta lv \rceil & \} \\
 & \mathtt{CAMLregisterlocal}(a) & \\
\{ & \mathsf{GC}\ \theta * a \mapsto_{\mathrm{root}[\mathtt{f}]} lv * \mathtt{current\_fc\ f::fc} * \mathtt{local\_roots\ f}\ (a \cup rs) & \}
\end{array}$$

$$\begin{array}{lcr}
\{ & \mathsf{GC}\ \theta * \mathtt{current\_fc\ f::fc} * \mathtt{local\_roots\ f}\ rs & \} \\
 & \mathtt{CAMLunregisterlocal}() & \\
\{ & \mathsf{GC}\ \theta * \mathtt{current\_fc\ fc} \mathop{\Large *}_{a \in rs} \exists w, a \mapsto_C w & \}
\end{array}$$

Fig. 14. Specifications for the new roots primitives of Melocoton-local

When we modeled our root pointers, we actually had a choice. We could have modeled roots as a simple resources which would have modeled the entirety of the roots map. Specifications would have to take the whole resource every time they need to express even the smallest properties on roots. This would be very cumbersome and would make us loose a lot of flexibility.

A better approach is given to us by using *ghost variables*. In program verification, ghost variables are a common way to track the current state of the program. We create a first resource which simply state that there exists a roots map rm which satisfies certain conditions. This resources assert the *ownership* of the roots map. It will be needed when doing modification. We then create separate resources which gives us more information on this map. We could have a resources which would tell us that the address a points to the value lv, or a resource which would simply say that the address a is rooted. We can then create specifications which simply talks about those second kind of resources. This choice of design gives us a lot more flexibility for proofs, by allowing us to have more local reasoning. When we want to increase our comprehension of the whole system, we can combine the resources together. We associate a name to each resource to know what is the actual object that the resource is talking about.

This is exactly how roots frame pointers are implemented. The name associated to a root frame pointer $a \mapsto_{\mathrm{root}[\mathtt{f}]} lv$ simply is f.

Our new resources expose a new view to the programs roots. A partial view is already provided by the GC token. We need to makes sure that all of these views are coherent. Under the hood, the GC token also relies on ghost variables. The ownership of the different resources is therefore split between the GC token and our new resources. Each part of the ownership is associated to a fraction, and we can only modify the object if we own the full fraction. We can merge and split the ownership at will.

The last thing we want to do is proving that the exposed logic actually make sense relative to the underlying semantics. Compared to the previous logic for boxed integers, the new logic exposed is complex, and the proof of adequacy is harder. The difficulty of proving the adequacy is reconciling the global view given by the operational semantics to the very local one on the resource side. One difficulty comes from the preservation of frame across function calls. It is closely related to well-bracketed control flows[9]. The proof is located in the `interop/wp_simulation` file.

5.4 Proving the correctness of `minitime`

With all of these new resources, we can finally express the complete specification of `caml_minitime` and prove it. The specification we want to prove is presented in Figure 15. The complete proof can be found in Melocoton in the file `examples/gmtime.v`.

$$\begin{array}{lcr} \{ & \mathsf{GC}\ \theta * \texttt{current\_fc fc} * \lceil lv \sim \boxed{n} \rceil & \} \\ & \texttt{caml\_minitime}(lv) & \\ \{ & \exists\, lv'\ v_1\ v_2, \mathsf{RET}(lv').\ \mathsf{GC}\ \theta' * \texttt{current\_fc fc} * \lceil lv' \sim \langle v_1, v_2 \rangle \rceil & \} \end{array}$$

Fig. 15. `caml_minitime` real specification

6 RELATED WORK

Since formalising multi-lingual interoperability is a new topic, related works are sparse. Most of the related works are related to the formalisation of local C variables. Comcert[7] provide a formalisation of the C standard. It is implemented in Coq but not using separation-logic. *The C standard formalised in Coq*[6] is implemented in Coq using separation logic. Its implementation is more robust than the one provided in this work, having support for combinations of local variables with non-local control flows such as `goto`. They are implemented using Huet's Zipper [4]. It differs from this work by the simplicity of implementation. The focus of this work being local roots, we chose to use a representation which would be easy to implement without having to do complex proofs and deep changes on how resources are implemented.

7 FUTURE WORK

In parallel of this work, a proposition for adding exceptions to Melocoton has been made. Combining local roots with complex control flows such as this one can be difficult. Raising an exception means an early return of a function frame which still need to be de-allocated, both in the memory and in the roots. A difficulty could come from the current implementation of the C function frames. The C semantics currently does not track it's frames. We could fix this by reworking the implementation of C local variables, adding a form of stack to the C state which would be tracked by the semantics and where we could easily free the top frame. Its implementation could be similar to the one given for local roots.

This implementation could also fix another minor problem: roots frame leak. There is currently a strong distinction between the roots frame and the local variable frame. Nothing forbids us in the semantics to register a root without properly initialising a new root frame first. We chose to ignore this problem as programs which does not properly initialise their frame cannot be expressed in a real FFI program, as they would be compile time errors. We also cannot prove any properties of such program with our rules, because we force that any function returns the local roots frame untouched, and we cannot unregister a single roots from the roots frame.

Acknowledgements. I'd like to thank Armaël for his support all the way throughout the internship. I'm very thankful for the chance of working with him. On top of giving me very good insight on my work and giving away a lot of his free time to give me feedback and help anytime I had the need, he was also one of the friendliest figure I had the chance to meet. I couldn't have hoped for a better supervisor.

REFERENCES

[1] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. Melocoton: A program logic for verified interoperability between ocaml and c. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.
[2] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.
[3] Johannes Hostert. Logical Foundations Of Language Interoperability Between OCaml And C, July 2023.
[4] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
[5] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.
[6] Robbert Krebbers. The C standard formalized in Coq. In *The C standard formalized in Coq*, 2015.
[7] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, November 2009.
[8] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
[9] Amin Timany, Armaël Guéneau, and Lars Birkedal. The logical essence of well-bracketed control flow. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.