Verified interoperability with exceptions between OCaml and C

VALERAN MAYTIE, Université Paris-Saclay, Stagiaire, Laboratoire Méthodes Formelles, France
ARMAËL GUÉNEAU, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Internship supervisor, France

Additional Key Words and Phrases: foreign-function interfaces, exceptions, OCaml, C, program logics, multi-language semantics, separation logic, Iris, Coq

1 INTRODUCTION

For several years, research in program logics has led to the development of tools for verifying programs in various languages, typically focusing on one language at a time. For example, there is Why3 [Bobot et al. 2015] for OCaml, Creusot [Denis et al. 2022] for Rust, and FramaC [Cuoq et al. 2012] for C. However, in practice, many software projects use several languages linked together through a mechanism called Foreign Function Interface (FFI). This is particularly useful in higher-level languages to access system libraries written in C/Assembly or various libraries written in C, C++, or Rust, such as graphical libraries or numeric computing libraries.

Linking different languages can often be challenging because memory models can vary significantly, and the use of a Garbage Collector (GC) can introduce additional complications. Therefore, writing FFI code is subtle and requires a strong understanding of both languages involved. It is, therefore, important to develop tools that enable us to reason about the interoperability between languages.

In this work, we are particularly interested in the behavior of exceptions within the OCaml FFI. Exceptions in a single language can make the control flow non-linear, so combining them with the FFI is challenging. This is especially true because the OCaml FFI includes primitives for making calls to OCaml code from C, which can interleave OCaml and C stack frames. As a result, an exception can unwind multiple stack frames, potentially disrupting some GC roots. More details on handling exceptions within the FFI are provided in Section 2.

To prove the correctness of complex programs, we aim to use formal methods, specifically Melocoton [Guéneau et al. 2023], a recently published, state-of-the-art tool. Melocoton is a multi-language program verification system designed for reasoning about OCaml, C, and their interactions through the OCaml FFI. Melocoton first defines the multi-language semantics between OCaml and C, which represents a substantial subset of the OCaml FFI. Additionally, Melocoton employs program logics based on the Iris [Jung et al. 2018] separation logic [Reynolds 2002] framework, enabling reasoning about multi-language systems. Important Melocoton concepts related to exceptions are explained in Section 4. For more details, Johannes Hostert's master's thesis provides a comprehensive overview of the project [Hostert 2023].

The goal of this work is to extend Melocoton to handle exceptions, enabling reasoning about OCaml-C programs that involve exceptions. This is a challenging task because Melocoton's control flow is currently completely linear. Moreover, for the sake of modularity, the modeled languages in

Authors' addresses: Valeran Maytie, Université Paris-Saclay, Stagiaire, Laboratoire Méthodes Formelles, France, valeran.maytie@universite-paris-saclay.fr; Armaël Guéneau, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Internship supervisor, 91190, Gif-sur-Yvette, France, armael.gueneau@inria.fr.

This work is licensed under a Creative Commons Attribution 4.0 International License.

Melocoton are designed to be independent of each other, which complicates control flow manipulation.

This work contributed to improving Melocoton by adding exception handling to the OCaml modeling language. This enhancement allowed for the modeling of communication between two very different languages, including the transfer, triggering, and recovery of exceptions. It also increased the expressive power of reasoning rules by incorporating exceptions into postconditions. To achieve this, we extended Melocoton's Coq code by adding definitions and proof code, as documented in the following pull requests: 18, 20, 23, 25, and 27 (all of which have been merged) and the "exception" branch. The code can be found at https://github.com/logsem/melocoton. We also tested the modeling on a simple example in Coq.

These contributions can be summarized as follows:

- Adding exception handling to Melocoton's ML core language by introducing two expressions: *raise* and *try* (Section 3).
- Propagate exception transfer between two languages with different value representations by modifying intermediate languages for communicating values (Sections 5.1 and 5.2).
- Adding OCaml FFI primitives related to exceptions: **`caml_callback_exn`** and **`caml_raise`**, including the definition of their semantics and reasoning rules (Section 5.3).
- Testing the modeling on examples (Section 6).

2 FFI WITH EXCEPTION

Before discussing Melocoton, we begin by introducing the OCaml-C Foreign Function Interface (FFI), particularly focusing on the behavior of exceptions within this FFI. This section is based on Chapter 22 of the OCaml manual [oca 2024, Chap. 22]. The OCaml-to-C FFI allows you to execute C code within an OCaml program.

The first step in using the FFI is to declare a C function with the name `ocaml_add_one` in the OCaml environment, represented by the name `add_one` and the type `int -> int` using the following pattern:

```
external add_one : int -> int = "ocaml_add_one"
```

The function `add_one` can then be called as a regular OCaml function, with the parameters and results adhering to the types specified in the declaration.

Now that the function has been declared, we need to define it in the C environment. This step is more complex because the programmer must understand OCaml's memory model and garbage collector (GC).

To start, OCaml values are represented by the `value` type because, during the execution of an OCaml program, the OCaml runtime is unaware of the specific types of values being manipulated. OCaml uses a block-based memory model, meaning that from the runtime's perspective, there are two types of values: integers and addresses pointing to blocks. These blocks contain metadata for the GC, such as block size and an array of values. The runtime distinguishes between integers and pointers using the least significant bit, which is set to one for integers and to zero for addresses.

To manipulate these values in C, the FFI provides several primitives:

- **`Val_int`**: returns the OCaml value as a C integer.
- **`Int_val`**: returns the C integer as a OCaml value.
- **`Val_unit`**: returns a *unit* value.
- **`String_val`**: returns a array of *char* which represents an OCaml string.

```c
void caml_read_file(value fun, value v) {
  CAMLparam2(fun, f);
  char *fname = String_val(v);
  FILE *file = fopen(fname, "r");
  caml_result r;
  char c;
  while ((c = fgetc(file)) != EOF) {
    r = caml_callback_exn(fun,Val_int(c));
    if (res.is_exception) {
      fclose(file);
      caml_raise(res.data);
    }
  }
  fclose(file);
  CAMLreturn(Val_unit);
}
```

file.c

```ocaml
external read_file :
  (int -> unit) -> string -> unit =
  "caml_read_file"

let iter c =
  if c = 0
  then failwith "c = EOF"
  else Printf.printf "%c" (Char.chr c)

let () =
  try read_file iter "text.txt"
  with Failure v -> print_string v
```

file.ml

Fig. 1. Representative example illustrating the combination of exception and resource management

There are many more primitives described in the manual. However, using these primitives can be unsafe because if the programmer provides an integer value instead of an address value, the primitives might crash the program or corrupt memory (e.g., `String_val(Val_unit)`).

Since OCaml has a Garbage Collector (GC), values must be registered with the GC. Because the GC can move blocks during execution, the addresses of these blocks can change. Therefore, the GC must also update the addresses within the C function. To register a value with the GC, we can use primitives like `CAMLparm2`, which registers two parameters with the GC. This aspect of the FFI is complex and prone to bugs, but it is not our primary concern regarding exceptions.

The OCaml FFI contains two essential primitives for handling exceptions. The first primitive, `caml_raise`, is used to throw an ML exception in C code. It takes as a parameter a value representing the exception. This primitive functions similarly to OCaml's **raise** keyword. However, it is even more subtle due to the FFI's support for callbacks via the `caml_callback` primitive. This allows for a stack with fully interleaved C and OCaml frames, where a raise can potentially unwind both OCaml and C stack frames. Another important primitive is `caml_callback_exn`, which allows a value to be applied to an ML closure while blocking all exceptions. This primitive takes two values as parameters: the first represents the ML closure, and the second is the value applied to the closure. Once execution is complete, the primitive returns a structure of type `bool; value`. The second field contains an exception if the first field is true; otherwise, it contains the normal return value[1].

2.1 Using exception in FFI by example

To make the FFI more concrete, we present an example in Figure 1, which is a simplified adaptation of a pattern commonly found on GitHub. This example includes a C function, `caml_read_file`, which takes as parameters a function that accepts an integer and returns unit, as well as a file name. This function is declared in the OCaml file as `read_file` (lines 1-3). We pass a function, `iter` (line 5), which displays each character of the file being read one by one. If the character is an end-of-file (`EOF`), we throw an exception.

The `iter` function, written in C, may take a closure that raises an exception, so care must be taken when executing this closure. To begin, we must register the two parameters with the GC using the

[1]This behavior is currently being implemented in the OCaml compiler (see: https://github.com/ocaml/ocaml/pull/13013)

$$
\begin{aligned}
v \in \mathit{Val} &::= (n \in \mathbb{Z}) \mid (b \in \mathbb{B}) \mid \langle\rangle \mid (\ell \in \mathit{Loc}) \mid \langle v,v\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{rec}\ f\ x.\ e \mid \textcircled{\textit{i}} \\
e \in \mathit{Expr} &::= v \mid x \mid e\ e \mid \ominus\ e \mid e \otimes e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
&\quad \langle e,e\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ e\ e \mid \\
&\quad \mathsf{alloc}\ e\ e \mid e.(e) \mid e.(e) \leftarrow e \mid \mathsf{length}\ e \mid \mathsf{call}\ fn\ \vec{e} \\
\sigma \in \mathit{State} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathrm{list}(\mathit{Val}) \uplus \{\lightning\} \\
E \in \mathit{Ectx_item} &::= \square \otimes v \mid e \otimes \square \mid \square.(v) \leftarrow v \mid e.(\square) \leftarrow v \mid e.(e) \leftarrow \square \mid \ldots \\
K \in \mathit{Ectx} &::= E \circ \ldots \circ E
\end{aligned}
$$

BetaS

$$\frac{}{(\mathsf{rec}\ f\ x.\ e\ v, \sigma) \rightarrowtail (e\{f \leftarrow \mathsf{rec}\ f\ x.\ e\}\{x \leftarrow v\}, \sigma)}$$

AllocNS

$$\frac{n \geq 0 \qquad \ell \notin \mathrm{dom}\ \sigma}{(\mathsf{alloc}\ n\ v, \sigma) \rightarrowtail (\ell, \{\ell \mapsto [\underbrace{v, \ldots, v}_{n}]; \sigma\})}$$

Prim_step

$$\frac{(e_1', \sigma_1) \rightarrowtail (e_2', \sigma_2)}{(K[e_1'], \sigma_1) \rightarrow_{\mathsf{ML}} (K[e_2'], \sigma_2)}$$

Store

$$\{l \mapsto_{\mathrm{ML}} \vec{v}\}\ l.(i) \leftarrow v\ \{\langle\rangle.\ l \mapsto_{\mathrm{ML}} \vec{v}[i := v]\}_{\mathrm{ML}}$$

Alloc

$$\{n \geq 0\}\ \mathsf{alloc}\ n\ v\ \{l.\ \exists\, \vec{v}.\ l \mapsto_{\mathrm{ML}} \vec{v} * \vec{v} = n * \forall i.\ \vec{v}[i] = v\}_{\mathrm{ML}}$$

Fig. 2. Syntax, state, context, semantics and selected reasoning rules of λ_{ML} without exceptions

`CAMLparam2` primitive. Afterward, we retrieve the file name and open the file. We then loop through the file, processing each character by applying the function passed as a parameter. If the function raises an exception, the file must be closed properly. For this, we use the `caml_callback_exn` primitive, which allows us to catch the exception, close the file, and then re-raise the exception normally.

This is a common pattern because, in the C world, resources are often not managed by the GC. If a function raises an exception, this could lead to memory leaks. Therefore, we use this pattern to ensure proper resource management in the C environment.

3 EXTENDING MELOCOTON'S ML CORE LANGUAGE WITH EXCEPTIONS

Melocoton defines two core languages, λ_{ML} and λ_{C}, which respectively model essential features of OCaml and C. In this section, we describe how we extend λ_{ML} with exceptions that are similar to OCaml exceptions. Later, in Section 5, we will discuss how λ_{ML} exceptions interact with the FFI and λ_{C}. In Section 3.1, we present the syntax and semantics of the current form of λ_{ML}. Then, in Section 3.2, we introduce $\lambda_{\mathsf{ML+exn}}$, an extension of λ_{ML} that includes exceptions. Finally, in Section 3.3, we explain the Hoare-style reasoning rules, or triples, of $\lambda_{\mathsf{ML+exn}}$.

3.1 Syntax, semantics and program logic of Melocoton's ML language

Figure 2 defines the syntax and some of the operational semantics rules for λ_{ML}.

To begin, λ_{ML} includes basic functional values and expressions, such as integers and booleans, with unary $\ominus$ and binary $\otimes$ operations. It also supports the unit value $\langle\rangle$ and pairs using the $\langle,\rangle$

constructor, along with destructors fst and snd. With these features, we can construct n-tuples as follows: $\langle e_1, \ldots, e_n\rangle \triangleq \langle e_1, \langle \ldots, \langle e_n, \langle\rangle\rangle\rangle\rangle$. Additionally, there are also variant constructors inl, inr with the destructor case. We can represent algebraic types such as the *option* types : None $\triangleq$ inl $\langle\rangle$ and Some e $\triangleq$ inr e. To represent functions, there are recursive closures $\mathsf{rec}\, f\, x.\, e$. We use the notation : $\lambda x.e \triangleq \mathsf{rec}\, _\, x.\, e$, $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleq (\lambda x.e_1)\ e_2$ and $e_1; e_2 \triangleq \mathsf{let}\ \langle\rangle = e_1\ \mathsf{in}\ e_2$.

Melocoton's ML language also includes arrays and references. Memory is modeled as a map σ from locations ℓ to an array of arbitrary values. Both arrays and references are simply locations in this memory map. The expression $\mathsf{alloc}\, n\, v$ allocates a block of n consecutive heap cells initialized with the value v (a reference is a block of size 1). There is no explicit memory release expression, as memory management is handled automatically by a garbage collector. It is possible to read from memory using $\ell.(n)$, write to memory with $\ell.(n) \leftarrow v$, and obtain the size of a memory block pointed to by a location with $\mathsf{length}\, \ell$.

There are three features of λ_{ML} to interact with other languages. First, the main FFI mechanism is the external function call, modeled by the expression $\mathsf{call}\ fn\, \vec{e}$. This expression calls the function fn of another language with the parameters $\vec{e}$. We can also have values that come from another language which are modeled using foreign values $\textcircled{\iota}$. These values can only be manipulated by external calls. Finally, the $\not\iota$ special heap contents represent locations temporarily owned by another language. See [Guéneau et al. 2023] for more details.

The operational semantics of λ_{ML} is a small-step semantic. To define it, we use a reduction on simple expressions ($\rightarrowtail$). This reduction takes an expression e and a state σ and returns new ones. For example, the rule BetaS apply a value v to a closure $\mathsf{rec}\, f\, x.\, e$, the state does not change, so σ is returned unchanged. The allocation rule modifies the state by adding a new list of values to the location ℓ chosen non-deterministically. Then there is the main relation ($\rightarrow_{\mathsf{ML}}$) which applies to an entire program. This relation uses large evaluation context K which is a composition of small evaluation contexts E with a single hole. There is an operation that fills the hole of the small context with an expression $E[e]$. This relation contains only Prim_Step which applies a small reduction inside a large expression and returns a new state.

One of Melocoton's key idea is to verify the correctness of program in λ_{ML} (and now $\lambda_{\mathsf{ML+exn}}$) code that contains FFI calls. Unfortunately, making proofs using operational semantics alone is very laborious, so to solve this problem Melocoton use resoning rules on the form of Hoare triple [Hoare 1969]. The logic inside triples is an extension of separation logic [Reynolds 2002] provided by the Iris framework [Jung et al. 2018], this instantiation is called **Iris**$_{\mathsf{ML}}$. Figure 2 shows a selection of **Iris**$_{\mathsf{ML}}$. A rule is a triple composed of an expression and two predicate: precondition and postcondition. The precondition contains properties that need to be checked and resources which are consumed after the expression is executed. The postcondition takes the form of a function which is applied to a result, giving us new resources and properties. For example, the rule Store in Figure 4 take $\ell \mapsto_{\mathrm{ML}} \vec{v}$ which is a resource that indicates that ℓ points to the array $\vec{v}$. After the execution, the resource is consumed and the expression returns the unit value $\langle\rangle$ and a new resource $\ell \mapsto_{\mathrm{ML}} \vec{v}[i := v]$, which changes the element i pointed by ℓ. The Alloc rule creates a resource, so we just need to prove that the size n is nonnegative. The rule provides a location ℓ which points to a list of values $\ell \mapsto_{\mathrm{ML}} \vec{v}$. The postcondition gives some properties on the list such as its size is equal to n and all the values inside $\vec{v}$ are equal to v.

3.2 Adding exceptions to λ_{ML}

Figure 3 describes $\lambda_{\mathsf{ML+exn}}$, our extension of λ_{ML} with exceptions. We model the behaviour of OCaml's exceptions in $\lambda_{\mathsf{ML+exn}}$. We start by adding the expression $\mathsf{raise}\ e$ and how to propagate them, then we add the expression $\mathsf{try}\ e\ r$ to be able to catch them.

$$e \in Expr \mathrel{+}= \mathsf{raise}\ e \mid \mathsf{try}\ e\ e$$
$$E \in Ectx_item \mathrel{+}= \mathsf{raise}\ \square \mid \mathsf{try}\ \square\ e$$

$$\textsc{TryVS}\quad \frac{}{(\mathsf{try}\ v\ e, \sigma) \rightarrowtail (v, \sigma)} \qquad \textsc{TryRS}\quad \frac{}{(\mathsf{try}\ \mathsf{raise}\ v\ e, \sigma) \rightarrowtail (e\ v; \sigma\})}$$

$$\textsc{RaiseS}\quad \frac{E \neq \mathsf{try}\ \square\ e}{(K \circ E[\mathsf{raise}\ v], \sigma) \rightarrow_{\mathsf{ML}} (K[\mathsf{raise}\ v]; \sigma\})}$$

Fig. 3. Exception in $\lambda_{\mathsf{ML+exn}}$

$$\textsc{Raise}\quad \frac{E \neq \mathsf{try}\ \square\ e \qquad \{P\}\ \mathsf{raise}\ v\ \{\phi\}_{\mathrm{ML}}}{\{P\}\ E[\mathsf{raise}\ v]\ \{\phi\}_{\mathrm{ML}}} \qquad \textsc{Try}\quad \frac{\forall v.\ \{E\ v\}\ r\ v\ \{\phi\}_{\mathrm{ML}} \qquad \{P\}\ e\ \{\mathtt{OVal}\ v \Rightarrow \phi\ v \mid \mathtt{OExn}\ v \Rightarrow E\ v\}_{\mathrm{ML}}}{\{P\}\ \mathsf{try}\ e\ r\ \{\phi\}_{\mathrm{ML}}}$$

Fig. 4. New rules in $\mathbf{Iris}_{\mathsf{ML+exn}}$

First, we add an expression raise which cuts across all current expressions of the current λ_{ML}. To reduce under the exception we add the context item raise $\square$, because thanks to the rule Prim_step, the square will eventually be reduced to a value. The reduction rule RaiseS allow exception to skip a small context E. To prevent a raise v passing though a try, we need to ensure that the context item E is not a try $\square$ e. This rule is incorporated into the primary reduction ($\rightarrow_{\mathsf{ML}}$) because the rule encapsulates multiple reductions. RaiseS must be included within a large context K, enabling its application to all expressions with raise v.

Finally, we add the expression try e e used to block exceptions and retrieve their values. The first argument e represents the main program, so we want to reduce into a value on an exception, so we use the Prim_step rule again, by adding a context item try $\square$ e. Thanks to the rule TryVS, if the expression is reduced to a value v, then v can pass though the try. Otherwise, the head reduction rule TryRS catches a raise v and applies the value v to the exception handler e

We have adapted the method used in Didier Rémy's course for large context semantics [Rémy 2015]. There are a few differences in the course, there is no notion of small context and there is only one type of reduction. We therefore had to adapt the rules to make them compatible with the existing ones.

3.3 Adapting of Melocoton hoar triples logic reasoning rules with exceptions

Now that the exceptions have been added to λ_{ML}, we can adapt $\mathbf{Iris}_{\mathsf{ML}}$ logic to be able to reason about the new expressions by extending $\mathbf{Iris}_{\mathsf{ML}}$ into $\mathbf{Iris}_{\mathsf{ML+exn}}$, shown in Figure 4.

$\mathbf{Iris}_{\mathsf{ML+exn}}$ contains two new rules Raise and Try. The rules are more complex, because exceptions manipulate the execution flow, which is difficult to represent in triple form. The rule Raise, passes a raise v through a small context E. To do this, we must show that E is not a try $\square$ r, then we can continue the proof without the small context E. The rule Try verifies the expression e within a try. After the verification, if e create a value v, then $\phi\ v$ holds, otherwise $r\ v$ must be verified. To make this distinction, we introduce a new notation: $\mathtt{OVal}\ v \Rightarrow V \mid \mathtt{OExn}\ v \Rightarrow E$. With this notation

Fig. 5. Melocoton structure

the formula $V\ v$ holds if the expression returns a value v. Otherwise, if the expression raise an exception with a value v, the formula $E\ v$ holds. To transfer the "logical result" of executing e, we use a formula E and the value is transferred to the triple using a for-all. We must now check that the result of $r\ v$ verifies the postcondition Φ.

Thanks to the Iris framework modularity, after instantiating **Iris**$_{ML}$, we obtain the adequacy of **Iris**$_{ML}$(Theorem 3.1). This is important, because if we succeed to prove a triple, then adequacy tells us that the program does not crash and the result verifies the postcondition, we say that the e is safe with respect to ϕ). We prove that each rule is correct with respect operational semantic, by adding the **Iris**$_{ML+exn}$rules, the theorem remains true without modification.

THEOREM 3.1 (ADEQUACY OF **IRIS**$_{ML}$). *Let e a λ_{ML+exn}expression, σ a state and ϕ : $Val \rightarrow Prop$*

$$\{\texttt{True}\}\ e\ \{\phi\}_{ML} \Rightarrow (\forall v, \sigma', (e, \sigma) \rightarrow^*_{ML} (v, \sigma') \Rightarrow \phi\ v)$$

4 BACKGROUND: THE STRUCTURE OF MELOCOTON

Before making any change, it is important to understand certain parts of Melocoton unchanged, such as the link between two language, and in particular how the control flow is modeled. As shown in Figure 5, Melocoton is composed of a mechanism for linking two languages describe in the first section, and a wrapper around $\lambda_{ML}$to link it with $\lambda_C$describe in the second section. Finally, we present the top block, which creates the program logic for this combination of languages.

4.1 Linking two abstract languages with the same value definition

In practice, when we compile a program using FFI, we go through a process called linking. Linking combines binary files possibly produced from different languages together to create an executable. Before linking two languages in Melocoton, we need to make a general definition of the notion of language. A language is a structure that depends on a notion of value *Val*, but in must also instantiate other definitions:

- Expressions *Expr*, with two functions:
  - `of_val` : $Val \rightarrow Expr$, transform a value into an expression (total function)
  - `to_val` : $Expr \rightarrow Val + \bot$, transform an expression into a value (partial function)
- Context K, with basic operations: composition $\circ$ and filling a context with an expression $K[e]$ (produces an expression)
- State, which doesn't really interest us here
- Functions *Fun*, with two functions:
  - `of_call` : $Fun \times \vec{Val} \rightarrow Expr$, transform a function call into an expression
  - `is_call` : $Expr \rightarrow (Fun \times \vec{Val} \times K) + \bot$, isolate a function call into a context ($e = K[fn\ \vec{v}]$)
  - `apply_call` : $Fun \times \vec{Val} \rightarrow Expr + \bot$, applies the arguments to the function.
- $(e, \sigma) \rightarrow^p (e', \sigma')$, symbolizing the operational semantics parameterized by a program $p \triangleq fn \rightharpoonup Fun$

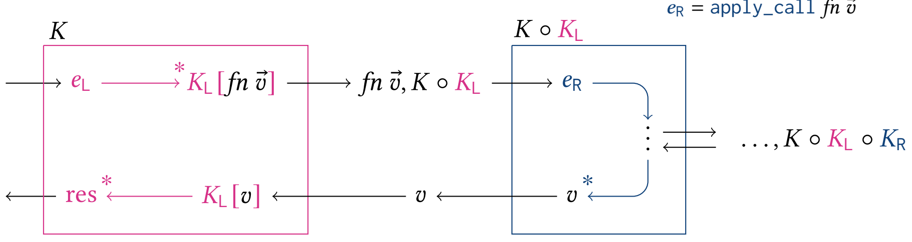

Fig. 6. Linking semantics, between two languages (pink and blue) symbolized by boxes of two different colors. Black arrows represent internal steps of the linking language for communicating languages

Melocoton defines a linking operator: "$- \oplus -$", which takes two instantiated languages with the **same** value definition as parameters, to create a new language noted: $\lambda_L \oplus \lambda_R$. In $\lambda_L \oplus \lambda_R$, λ_L (resp. λ_R) can make external calls if the function is defined in the λ_R (resp. λ_L) program. To detect whether a function call is external, linking semantics has a rule that determines whether execution is blocked during a function call, and if so, linking hands over to the other language. The transition from one language to another contains quite few steps in the operational semantics, illustrated in Figure 6 (The memory state changes, but we won't explain this part, because in has no influence on the control flow). The semantics of linking save the state of the calling language in a context, the semantics of linking can retrieve the context using the function `is_call`. Finally, the linker transform the function call into an expression of the called language using the function `apply_call`. Once the function call is complete, when the function `to_val` return a value v. The linker retrieve the context and replace the hole with the result v, then the linking semantics continue execution. We note, that only values can be transferred from one language to another, and the control flow is totally linear.

4.2 Bridging the gap between two languages with different values (and state)

The gap between λ_{ML} and λ_C. The C language is modeled with λ_C. It has a completely different definition of values than λ_{ML} presented just below:

$$w \in Val := (n \in \mathbb{Z}) \mid (a \in Addr)$$

The λ_C values, are either integers, or an address that points to a memory block that contains an address. These values are completely different than those of λ_{ML} which are more structured. So we can't link directly these two languages using the linking language presented in the previous section.

Bridging the gap. To bridge the gap between λ_C and λ_{ML} languages, Melocoton introduces the wrapping combinator "$[-]_{FFI}$". This combinator allows us to obtain a language $\lambda_{[ML]_{FFI}}$ and a semantics $\rightarrow_{[ML]_{FFI}}$ described in Figure 7. This semantic allows you to execute λ_{ML} expression with the λ_{ML} semantic $\rightarrow_{ML}$ (purple block in the Figure 7), execute FFI primitives called from C code, and translate λ_{ML} and λ_C values. The semantics of the wrapper are special in that they are non-deterministic, meaning that they point an expression to a set of possible executions. To bridge the gap between the two languages, Melocoton need to be able to translate a λ_{ML} value into a λ_C value. To do this, the wrapper contains a relation:

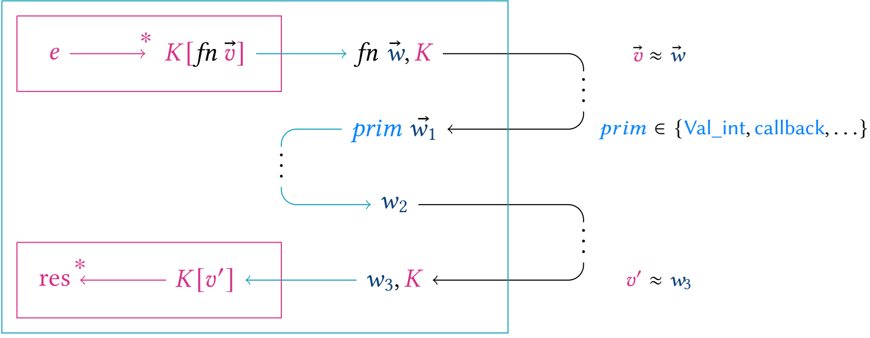

Fig. 7. $\lambda_{[\mathsf{ML}]_{\mathrm{FFI}}}$ semantics presentation, the pink boxes contain the λ_{ML} wrapped code, encompassed by a blue box representing the wrapper.

$v \approx w$ $\qquad$ v and w represent the same value in the λ_{ML} memory model[2]

Memory in the wrapper. The reader may have noticed that the relation: $\approx$ does not depend on memory, but translating an address to an λ_{ML} value requires access to memory, we decided to ignore it to simplify the explanations.

Wrapper semantics. In this language λ_{ML} expression $[e]_{\mathrm{FFI}}$ can be execute thanks to the rule WStep (shown in Figure 7), in order to apply the rule, we need to prove that the expression is not blocked. The execution of an expression can be blocked when it represents an external function call, we transform this function call into an internal expression of $\lambda_{[\mathsf{ML}]_{\mathrm{FFI}}}$ that communicates the function name, save context and arguments translated into value of λ_{C} thanks to the rule WCall shown in Figure 8. Once the function call is complete, the RetS rule retrieves the result, which is translated into an λ_{ML} value for insertion into the saved context.

FFI primitives. The FFI also provides primitives that allow λ_{C} expressions to communicate with λ_{ML} expressions. To model them, Melocoton use the wrapper, which retrieves a function call with arguments which the wrapper transform into a primitive call. Most primitives don't execute λ_{ML} expression and return a λ_{C} value directly. Their semantics are modeled using a separate relation $\rightsquigarrow$ that relates an expression (primitive call) to a λ_{C} value set, this relation is used in the wrapper's global semantics in the rule WPrim. The only primitive that calls an λ_{ML} expression has a semantics directly modeled on the wrapper's semantics $\rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}}$ is callback, which reduce the primitive call to a λ_{ML} expression in the rule WCallback. Once the execution of the function is complete, the value must be returned to the linker using the WRet rule. So we now have a language $\lambda_{[\mathsf{ML}]_{\mathrm{FFI}}} \oplus \lambda_{\mathsf{C}}$ and a semantics that model the execution of an OCaml and C program ho can communicate thanks to the FFI.

4.3 Multi-language program logic

Communication between the two logics. Now that the semantics of λ_{ML} and λ_{C} can communicate, Melocoton link **Iris**$_{\mathsf{ML}}$ and **Iris**$_{\mathsf{C}}$ to have a new program logic called **Iris**$_{\mathsf{ML+C}}$, which allows you to prove multi-language specification between OCaml and C shown in Figure 5. To link the two logics, we add specifications Ψ for external functions inside the triple. This specifications are executed

[2] In this document we use $\approx$ as a simplification, because the details do not matter with exception In Melocoton/Coq, it corresponds to a combination of the inductive `is_val` and `repr_lval` defines in the `interop/basic.v` file. But in triple, this notation corresponds to the combination of `repr_lval` and to the different $\mapsto$ **Iris**$_{\mathsf{ML+C}}$ resources.

WStep
$$\frac{e \notin Val \qquad \exists e', e \rightarrow_{\mathsf{ML}} e'}{e \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} \{e' \mid e' \in Expr, e \rightarrow_{\mathsf{ML}} e'\}}$$

WCall
$$\frac{}{K[\mathsf{call}\ fn\ \vec{v}] \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} \{fn\ \vec{w}, K \mid \vec{w} \in Val^*, \vec{v} \approx^* \vec{w}\}}$$

WVal
$$\frac{\mathtt{to_val}\ e = v}{e \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} \{w \mid w \in Val, v \approx w\}}$$

WRet
$$\frac{v \approx w}{w, K \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} \{K[v]\}}$$

WPrim
$$\frac{prim\ \vec{w} \rightsquigarrow X}{prim\ \vec{w} \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} X}$$

WCallback
$$\frac{w \approx \mathsf{rec}\ f\ x.\ e \qquad w' \approx v}{\mathsf{callback}\ [w, w'] \rightarrow_{[\mathsf{ML}]_{\mathrm{FFI}}} \{(\mathsf{rec}\ f\ x.\ e)\ v\}}$$

Fig. 8. A subset of the wrapper semantics, without state

when a function is not in the program p, thanks to the Call-external rule. Otherwise, the function is retrieved from the p program and executed with the Call-internal rule. The rules are detailed just below; the program p and protocols Ψ are ignored in the following, as we're going to write them as a triple to make them easier to write.

Call-internal
$$\frac{p(fn) = f \qquad \{P\}\ f(\vec{v})\ @\ p, \Psi\ \{v.Q\ v\}}{\{P\}\ \mathtt{call}\ fn\ \vec{v}\ @\ p, \Psi\ \{v.Q\ v\}}$$

Call-external
$$\frac{p \notin \mathrm{dom}(p) \qquad P -\!\!* \Psi\ fn\ \vec{v}\ Q}{\{P\}\ \mathtt{call}\ fn\ \vec{v}\ @\ p, \Psi\ \{v.Q\ v\}}$$

Using FFI primitives in $\textbf{Iris}_{\mathsf{ML+C}}$. When executing C code in the FFI, we can manipulate objects from the λ_{ML} world and execute FFI primitives, so there are specific features to $\textbf{Iris}_{\mathsf{ML+C}}$. To illustrate some of this logic, we'll explain the ExecCallback rule shown below, which has been simplified for the explanation[3].

ExecCallback
$$\frac{\{P\}\ (\mathsf{rec}\ f\ x.\ e)\ v\ \{Q\}}{\{w \approx \mathsf{rec}\ f\ x.\ e * w' \approx v * P\}\ \mathsf{callback}\ w\ w' \{r.\ \exists r.\ r \approx r * Q(r)\}}$$

To begin, the primitive callback takes two arguments w and w'. The first must corresponds to an λ_{ML} closure, for which we use $\approx$ notation in the precondition, then the second argument can correspond to any λ_{ML} value that is applied to the closure. Next, the rule execute the λ_{ML} expression $(\mathsf{rec}\ f\ x.\ e)\ v$, to transmit precondition, we use the predicate P, which is transformed into the post-condition Q once execution is complete. The primitive returns an expression $\lambda_{\mathsf{C}} r$ which correspond to an expression $\lambda_{\mathsf{ML}}\ r$ which satisfies the Q postcondition.

The logic $\textbf{Iris}_{\mathsf{ML+C}}$ is based on the semantics of $\lambda_{[\mathsf{ML}]_{\mathrm{FFI}}} \oplus \lambda_{\mathsf{C}}$ language, of in the end we have an adequacy theorem similar to tho Theorem 3.1, for the multi-language logic.

5 ADDING EXCEPTIONS TO MELOCOTON

Now that we have covered the basics of Melocoton, we can start modifiynig it to add exceptions. As mentioned in the intoduticon, this is not a trivial addition, espacially as Melocoton's code base is already very large and complex. Even adding small notions can be very complex, so weneed to be methodical when implementing them in Melocoton. To start adding exception, we had to find a way to pragate them through the linker, which required modifying about 1000 lines of Coq to set

[3]We hide Iris resources that talk about memory and GC management, which makes the rules more complicated.

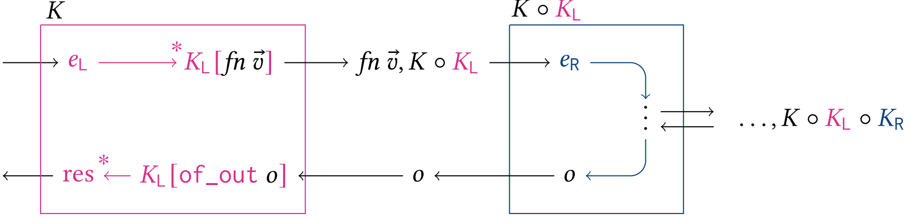

Fig. 9. Linking semantics with outcome

up the system without yet adding the exceptions. Then we had to define the new primitves in the wrapper and define reasoning rules for them, which makes about 1500 lines of Coq.

Before, we stat modifying the means of communication between languages we need to define what can be sent to model the raising of an exception. To do this, we create the notion of generic outcome parameterized by a value definition:

$$o \in Out(Val) \coloneqq v_{\texttt{val}} \qquad \text{with } v \in Val$$
$$\mid v_{\texttt{exn}} \qquad \text{with } v \in Val$$

We instantiate this definition, with the value of λ_{ML}: $Out(Val)$, abstract outcome are noted like this: o_{ML}. We do the same thing for the $\lambda_C$values which gives: o_C in $Out(Val)$. Now, we adapt the linking and wrapper languages to connect the outcome between $\lambda_{ML}$and λ_C. In the first subsection, we present how we chose to adapt the linking language with the outcome. Next, we show how the wrapper has been modified to handle outcomes. Finally, we add the two primitives: callback_exn and raise to the wrapper.

5.1 Transferring exception between two linked languages

As we saw in Section 4.1, the linking lanugage only transmitted values at the end of a function. Now we want to be able to propagate en exception in the other language. We have a notion of generalized outcomes, we can integrate it into the language interface. The interface is always parameterized by a value Val, which is used to instantiate an outcome within the language. New, functions are needed to link outcomes and expression:

- `of_out` : $Out(Val) \longrightarrow Expr$, transform a outcome into an expression (total function)
- `to_out` : $Expr \longrightarrow Out(Val) + \bot$, transform an expression into a value (partial function)

This new function defines both functions:

$$\texttt{of_val}(v) \triangleq \texttt{of_out}(v_{\texttt{val}}) \qquad \texttt{to_val}(e) \triangleq \begin{cases} v & \text{if } \texttt{to_out}(e) = v_{\texttt{val}} \\ \bot & \text{otherwise} \end{cases}$$

In our new version of linking, return values are transformed into outcomes, as shown in the Figure 9. A function is completed when the function: `to_out`, returns the output outcome o which is passed to the other language and transformed into an expression e thanks to the function `of_out`, to fill the context.

Note that the `of_out` function is a total function, which creates a problem in relation to λ_C, because it has no exception. To do this, we extend the language by λ_{C+exn}, by adding the raise w expression, which represents the exception w propagating through a C program, cannot be caught in C. This is an implementation choice, as we could have made the `of_out` function partial and, when the function returns nothing, cut the context save in the internal state of the linking language

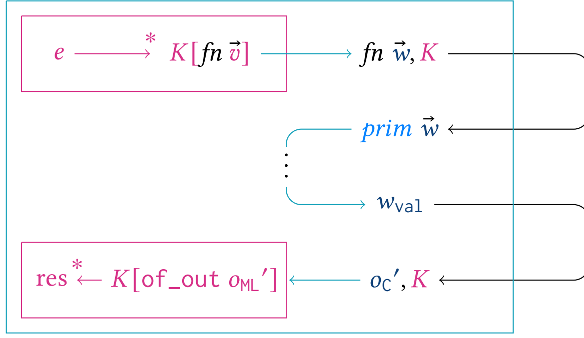

Fig. 10. modified wrapper smantics, omitting memory changes for clarity

$$\textsc{WVal} \rightarrow \textsc{WOut}\quad \frac{\texttt{to_out}\ e = o_{\text{ML}}}{e \rightarrow_{[\text{ML}]_{\text{FFI}}} \{w \mid w \in Out(Val), v \approx_o w\}}$$

$$\textsc{WRet}\quad \frac{o_{\text{ML}} \approx_o o_{\text{C}}}{o_{\text{C}}, K \rightarrow_{[\text{ML}]_{\text{FFI}}} \{K[\texttt{of_out}\ o_{\text{ML}}]\}}$$

$$\textsc{WPrim}\quad \frac{prim\ \vec{w} \rightsquigarrow X}{prim\ \vec{w} \rightarrow_{[\text{ML}]_{\text{FFI}}} X}$$

Fig. 11. modified wrapper semantics, without state

to translate it into the other language, this is not the mots general implementation choice, since a language theat doesn't catch an exception won't release its stack frame. Thanks to the chosen implementation, the exception passes all expressions thanks to a similar rule in RaiseS (semantic λ_{ML}) add to $\lambda_{\text{C+exn}}$.

5.2 Repairing the wrapper for transferring exception

The linking language transmits outcomes, we need to adapt the wrapper. Figure 10 shows similar modifications to the linker, including of_outcome, which propagates exceptions through the wrapper. Then you also need to be able to, it must be possible to relate o_{ML} and o_{C} outcomes, to do that we add the relation "$\approx_o$ based on the relation "$\approx$":

$$\textsc{OVal}\quad \frac{v \approx w}{v_{\texttt{val}} \approx_o w_{\texttt{val}}} \qquad \textsc{OExn}\quad \frac{v \approx w}{v_{\texttt{exn}} \approx_o w_{\texttt{exn}}}$$

To transfer outcomes, you need to modify three semantic rules, shown in the Figure 11. First of all, the WVal rule is transformed into WOut, because an expression returns outcomes instead of values. To determine whether an expression is terminated, the to_out function must return an outcome o_{ML}. This outcome is transformed into an outcome o_{C} of $\lambda_{\text{C+exn}}$which is then retrieved by the linker. Then, at the end of the execution of a $\lambda_{\text{C+exn}}$expression, we transform the recovered outcome into a $\lambda_{\text{ML+exn}}$outcome, which is transformed into an expression using the of_out function to fill the context. Finally, the WPrim rule doesn't change visually, but the $\rightsquigarrow$ relation returns a set X of $\lambda_{\text{C+exn}}$outcomes instead of a set of $\lambda_{\text{C+exn}}$values. We had to modify all the $\rightsquigarrow$ rules to replace the values w by $w_{\texttt{val}}$. Figure 10 shows the new path taken by the wrapper semantics. And finally, WRet can now be used to retrieve the boolean after execution of an external function. The λ_{ML}expression created by the callback primitive must also contain a boolean set to false, so as not to change its semantics.

WStep
$$\frac{e \notin Val \qquad \exists e', e \to_{\mathsf{ML}} e'}{e_b \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{e'_b \mid e' \in Expr, e \to_{\mathsf{ML}} e'\}}$$

WCall
$$\frac{}{K[\mathsf{call}\ fn\ \vec{v}]_b \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{fn\ \vec{w}, (K_b) \mid \vec{w} \in Val^*, \vec{v} \approx^* \vec{w}\}}$$

WRet
$$\frac{v \approx w}{w, (K_b) \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{K[v]_b\}}$$

WCallback
$$\frac{w \approx \mathsf{rec}\ f\ x.\ e \qquad w' \approx v}{\mathsf{callback}\ [w, w'] \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{((\mathsf{rec}\ f\ x.\ e)\ v)_{\mathtt{false}}\}}$$

WOut
$$\frac{\mathsf{to_out}\ e = o_{\mathsf{ML}}}{e_{\mathtt{false}} \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{o_{\mathsf{C}} \mid o_{\mathsf{C}} \in Out(Val), o_{\mathsf{ML}} \approx_o o_{\mathsf{C}}\}}$$

WOutC
$$\frac{\mathsf{to_out}\ e = o_{\mathsf{ML}}}{e_{\mathtt{true}} \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{\overline{o_{\mathsf{C}}} \mid o_{\mathsf{C}} \in Out(Val), o_{\mathsf{ML}} \approx_o o_{\mathsf{C}}\}}$$

RetCV
$$\frac{}{\overline{w_{\mathtt{val}}}, \sigma \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{a_{\mathtt{val}}, \{a := 0; a + 1 := w; \sigma\} \mid a \in Addr, \{a, a + 1\} \cap dom\ \sigma = \emptyset\}}$$

RetCR
$$\frac{}{\overline{w_{\mathtt{exn}}}, \sigma \to_{[\mathsf{ML}]_{\mathrm{FFI}}} \{a_{\mathtt{val}}, \{a := 1; a + 1 := w; \sigma\} \mid a \in Addr, \{a, a + 1\} \cap dom\ \sigma = \emptyset\}}$$

Fig. 12. new interop semantics rules, for rules that will write to the memory state σ of λ_{C}

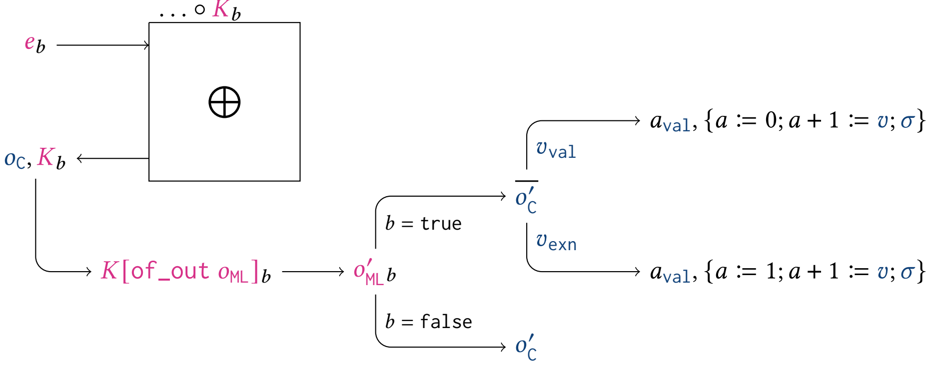

Fig. 13. catch exception on the wrapper

5.3 Modeling FFI exception primitives in the wrapper

This section discusses one of the larger contributions made in this paper, as it allows to model exception primitives, but to do this, it was necessary to make modifications to manipulate the control flow of the wrapper, which required doing very technical proofs in Melocoton.

Catching outcomes in the wrapper. Before creating the primitives, we need to extend the wrapper to catch the outcomes and transform them into structure (detailed in section 2.1). To do this, we add a boolean to the λ_{ML} expressions e_b to determine whether the outcome should be caught up. So as not to lose this boolean, we also add it to the context K_b which is passed to the linker to retrieve it after an external call. In doing this, we need to adapt several rules detailed in Figure 12.The WStep rule transmits the boolean at each step of the λ_{ML} semantics. Then WCall creates the context

RaiseP
$$\frac{}{\text{raise } w \rightsquigarrow \{w_{\text{exn}}\}}$$

ExecRaise
$$\{\top\} \text{ raise } w \ \{w_{\text{exn}}. \top\}$$

WCallbackExn
$$\frac{w \approx \text{rec } f\ x.\ e \qquad w' \approx v}{\text{callback_exn } [w, w'] \rightarrow_{[\text{ML}]_{\text{FFI}}} \{((\text{rec } f\ x.\ e)\ v)_{\text{true}}\}}$$

ExecCallbackExn
$$\frac{\{P\}\ (\text{rec } f\ x.\ e)\ v\ \{Q\}}{\{P * w \approx \text{rec } f\ x.\ e * w' \approx v\}\ \ \text{callback_exn } w\ w'\ \ \{a_C.\ \exists r.\ a_C \mapsto_C^* [n; r] * r \approx r * (n = 0 \Rightarrow Q(r_{\text{val}})) * (n = 1 \Rightarrow Q(r_{\text{exn}}))\}}$$

Fig. 14. adding callback_exn and raise in the wrapper semantics $\rightarrow_{[\text{ML}]_{\text{FFI}}}$ and **Iris**$_{\text{ML+C}}$. ExecCallback is simplified.

with the boolean function call expression from λ_{ML}. Then, when a λ_{ML} expression is terminated and the boolean is false, we do the same as before with WOut. Otherwise, if the boolean is true, we transform the outcome into a new wrapper expression $\overline{o_C}$ (WOutC rule), which symbolizing a caught-up outcome. Now that we've got a caught outcome, we can return an address that points to a boolean and the next box to the value contained in the outcome. The RetCV and RetCR rules distinguish whether the outcome is a value or an expression, allowing you to set the boolean to true or false.

Implementing the two exception primitives. Now that everything is in place, we can model the two primitives raise and callback_exn. We start with the simplest primitives: raise, it allows you to throw an exception withe a value that passed as arguments. For semantic, we just add the rule RaiseP, to the relation $\rightsquigarrow$, the rule return a singleton containing an exception outcome w_{exn}, with the parameter of the primitives as its value. The ExecRaise logical rule is just as simple, as it only returns an outcome exception like the RaiseP semantic rule. For the second **callback_exn** primitive, we start by adding the PCallbakExn rule, which is the same as PCallback, except that the boolean is set to true to catch the returned outcomes. Then, for the reasoning rule, we add ExecCallbackExn which is practically the same as ExecCallback, except that the postcondition takes as parameter a λ_C address pointing to an integer n and the next address pointing to a value r. The resource $\mapsto_C^*$ expresses that an address points to a memory bloc of λ_C. As seen earlier, the integer n determines whether the return value r represents an exception, so there are two implication that depending on whether n is equal to on (resp. zero) the $Q(r_{\text{exn}})$ (resp. $Q(r_{\text{val}})$) must be verified. Proving the ExecCallvackExn rule is difficult, because it includes many complex notions of Melocoton, such as writing in memory at an address chose non-deterministically, translation of λ_C values into λ_{ML} values, and above all this primitive puts into action all the language of Melocoton.

6 CASE STUDIES

To test the modeling, we created a completely artificial example that utilizes all the additions to Melocoton, allowing us to verify if the correctness proof is not hindered by overly weak rules. The example code (found in `theories/examples/raise.v`) resembles the pattern shown in Section 2.1 but is simplified by removing the loop and file handling to make the proof more manageable. This example has enabled us to test whether our model can check multilingual programs with exceptions.

7 CONCLUSION FUTURE WORK

To conclude, we have successfully implemented and proven in Coq all the contributions presented in this paper. We have divided the contributions into several Git branches, some of which have already been integrated into Melocoton. These include PR 23: adding outcome without exceptions in the linker, PR 25: refactoring the relation $\approx$ to prepare for adding outcomes in the wrapper, and PR 27: adding outcomes in the wrapper.

There are still many tasks ahead, such as proving more complex examples to fully integrate the addition of exceptions into Melocoton (merge branch exception). During his internship, Gurvan added local variables and stack frames to Melocoton. We will need to find a way to combine our work to create a version of Melocoton that includes stack control with exceptions.

Acknowledgments: I would like to thank Armaël for his excellent supervision and invaluable advice throughout the internship.

REFERENCES

2024. The OCaml manual – Chapter 22: Interfacing C with OCaml. https://ocaml.org/manual/5.2/intfc.html

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer* 17 (2015), 709–727.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A software analysis perspective. In *International conference on software engineering and formal methods*. Springer, 233–247.

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods*. Springer, 90–105.

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (oct 2023), 29 pages. https://doi.org/10.1145/3622823

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Johannes Hostert. 2023. *Logical Foundations Of Language Interoperability Between OCaml And C*. Ph. D. Dissertation. Saarland University.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Didier Rémy. 2015. Type Systems for Programming Languages. (2015). https://gallium.inria.fr/~remy/mpri/cours-mpri.pdf Course notes, available electronically.

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. (2002), 55–74. https://doi.org/10.1109/LICS.2002.1029817