Melocoton: A Program Logic for Verified Interoperability Between OCaml and C

ARMAËL GUÉNEAU*, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, France
JOHANNES HOSTERT*, Saarland University and MPI-SWS, Germany
SIMON SPIES*, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
LARS BIRKEDAL, Aarhus University, Denmark
DEREK DREYER, MPI-SWS, Germany

In recent years, there has been tremendous progress on developing program logics for verifying the correctness of programs in a rich and diverse array of languages. Thus far, however, such logics have assumed that programs are written entirely in a single programming language. In practice, this assumption rarely holds since programs are often composed of components written in different programming languages, which interact with one another via some kind of foreign function interface (FFI). In this paper, we take the first steps towards the goal of developing program logics for multi-language verification. Specifically, we present Melocoton, a multi-language program verification system for reasoning about OCaml, C, and their interactions through the OCaml FFI. Melocoton consists of the first formal semantics of (a large subset of) the OCaml FFI—previously only described in prose in the OCaml manual—as well as the first program logic to reason about the interactions of program components written in OCaml and C. Melocoton is fully mechanized in Coq on top of the Iris separation logic framework.

1 INTRODUCTION

In recent years, there has been tremendous progress on developing verification systems based on program logics (in particular, separation logics), which support the modular verification of complex programs in a rich and diverse array of languages. Notable examples include VST-Floyd [Cao et al. 2018] and RefinedC [Sammler et al. 2021] for C, RustBelt [Jung et al. 2017] and RustHornBelt [Matsushita et al. 2022] for Rust, and Cosmo [Mével et al. 2020] for multi-core OCaml—all of which are fully mechanized in Coq.

The above-cited systems all employ a common recipe:

1. Design an operational semantics for the language, which serves as the “ground truth” about program behavior.
2. Build a (Hoare-style) program logic for the language, which supports higher-level proof rules for compositionally verifying program correctness.
3. Establish soundness of the program logic by giving an interpretation of the logical judgments (e.g., Hoare triples) in terms of the operational semantics, and verifying the proof rules as lemmas about that interpretation.

*The three authors should be considered as joint first authors.
This recipe works great for verifying programs written entirely in a single language. However, in practice, programs are not typically written all in one language, but rather linked together from a patchwork of components written in different languages. For instance, they include calls to kernel and runtime primitives implemented in low-level languages such as C and assembly; leverage low-level efficient code implemented in languages like C, C++, and Rust; and reuse large, existing libraries implemented in a different language (e.g., numeric computing libraries written in C). In fact, almost all widely-used programming languages include some kind of foreign function interface (FFI) to interact with code written in other languages (often using C as an intermediary).

For these so-called “multi-language programs”, there is not yet a recipe for building (provably sound) program logics. For starters, step one (the question of how to define a semantics for multi-language programs) is still an active topic of research [Neis et al. 2015; Patterson et al. 2022; Sammler et al. 2023; Stewart et al. 2015]. Moreover, for steps two and three, there does not yet exist any program logic for reasoning about multi-language programs in which the constituent languages have (as is often the case) very different memory models. In short, the problem of building program logics for multi-language program verification is still wide open.

In this paper, we take the first steps towards filling this gap by proposing a recipe for building multi-language program logics. To keep matters concrete, we focus on a specific instance of the problem: verifying multi-language programs written in a combination of OCaml and C. OCaml has structured values (e.g., sums, pairs, lists, and references), a garbage-collected memory, and a type system providing strong guarantees about its programs. C has integer values and pointers, manually managed memory, and a type system providing only weak guarantees about its programs. Nevertheless, there exists a bridge between the two—the OCaml FFI—which exposes enough of the OCaml runtime to C to convert values, execute callbacks, and share memory.

1.1 Key Challenge: Bridging the Language Differences

In fact, in multi-language verification, one—if not the—main challenge is bridging the gap between the languages under consideration: the more the languages differ, the more the single-language reasoning principles that apply to them will differ. In this regard, OCaml and C are truly an “odd couple”: they differ in their values (i.e., abstract values in OCaml vs. concrete values in C), memory models (i.e., mutable records and arrays vs. pointers with pointer arithmetic), memory management (i.e., garbage collection vs. manual memory management), type systems (i.e., a strong, polymorphic type system vs. a weak type system), and runtime (i.e., large runtime vs. bare-bones binaries).

To illustrate these differences and how they are bridged by the OCaml FFI, let us consider a concrete example: the runtime representation of OCaml values in C. Consider the OCaml type:

```ocaml
type buf = { cap : int; used : int ref; data : bytes }
```

This type is used as part of the running example of this paper (in §2), where it exposes buffers implemented in C (i.e., raw chunks of bytes) to OCaml through the OCaml FFI. The data-field
stores the underlying bytes, the \texttt{cap}-field the capacity of the buffer, and the \texttt{used}-field how much of the buffer is currently in use (\textit{i.e.}, how many bytes are readable). The type \texttt{bytes}—as far as OCaml is concerned—is abstract (\textit{i.e.}, declared with \texttt{type bytes} without a definition), meaning it does not reveal anything about its contents to OCaml.

From the perspective of OCaml, values of type \texttt{buf} are records \{\texttt{cap}, \texttt{used}, \texttt{data}\} where the \texttt{cap}-field is an immutable integer, the \texttt{used}-field is a mutable reference, and the \texttt{data}-field stores some kind of immutable value (of an unknown shape). This is not, however, what the OCaml FFI exposes to C as the runtime representation of these values. Instead, nested values (\textit{e.g.}, records and pairs) are exposed as blocks of memory, \textit{runtime blocks}, which are nested using pointers. A value of type \texttt{buf} is a pointer to a block with three elements, one for each field of the record (as depicted in Fig. 1). The first field stores the runtime representation of the \texttt{cap}-integer; the second field stores the \texttt{used}-reference—a pointer to a block, which stores the number of used bytes; and the third field stores a pointer to a so-called “custom” block, which embeds C data into OCaml (\textit{e.g.}, raw C pointers). In this case, the custom block stores a pointer to the underlying bytes of the buffer.

Clearly, the OCaml view of \texttt{buf} values and their runtime representation in C is different. For one, in C, they reside in memory (with a concrete address where they are stored) whereas, in OCaml, they are conceptually “just values” (\textit{i.e.}, they are copied when passed around). However, that is not the only difference. The fields of \texttt{buf} are immutable (\textit{e.g.}, \texttt{cap} cannot change) and the \texttt{used}-reference is mutable. From the perspective of C, however, both records and references have the same representation, \textit{runtime blocks}. It is up to the programmer to remember that one of them is allowed to be mutated whereas the other is not. Finally, C code can embed its own data into OCaml through custom blocks (here the \texttt{data}-field) and it can \textit{access and modify} that data as needed whereas the \texttt{data}-field of the record is completely opaque to OCaml.

In \textit{program code}, the differences between OCaml and C are bridged through so-called “glue code”, code which uses the OCaml FFI to link up program parts written solely in either OCaml or C. For \textit{program verification}, the story is not so clear yet. For instance, when we reason about glue code, how do we reconcile the two very different views that OCaml and C have on values of type \texttt{buf}?

To design a multi-language program verification analogue of the single-language recipe, we have to bridge the language gap at two different levels:

The \textit{operational semantics}. At the operational semantics level, the challenge is that no existing multi-language semantics explains how to reconcile the key differences between both languages (\textit{i.e.}, interacting with the OCaml garbage collector and runtime, registering “roots”, executing callbacks, \textit{etc.}). However, there are promising starting points. Patterson et al. [2022] combine a garbage collected language and a language with manual memory management. Unfortunately, they do so using an elaboration semantics to a shared target language with garbage collection, which does not exist in the case of OCaml and C. Sammler et al. [2023] propose an approach to multi-language semantics that uses modular combinators to connect languages by composing their operational semantics. As such, their approach fits well with step one of the single-language recipe. Unfortunately, while they consider languages with significant differences (\textit{e.g.}, different calling conventions and memory models), they do not consider garbage collection or a runtime.

The \textit{program logic}. At the program logic level, the main challenge is preserving \textit{language-local reasoning}. That is, chunks of code that are solely written in C or solely written in OCaml should enjoy reasoning principles that are fine-tuned to their respective language (without being impacted by the existence of the other language). However, preserving language-local reasoning is easier said than done. For one, the two program logics will have different views on the \textit{same piece of state} (\textit{e.g.}, a value in OCaml will refer to nested runtime blocks on the C side). Changes to runtime blocks on the C side (\textit{e.g.}, updating the block storing the number of \textit{used} bytes) correspond to changes
in the state on the OCaml side (e.g., changing the used-reference in the buffer). Another, more subtle issue with preserving language-local reasoning is that code written in one language can potentially violate language-specific invariants of the other. For example, OCaml—as a functional language—makes pervasive use of immutable values. When linked with C, the C code can observe the runtime representation of those values and, in principle, mutate their contents, albeit with unknown consequences as far as the OCaml semantics is concerned.

1.2 Melocoton

In this paper, we present Melocoton, a multi-language program verification system for reasoning about OCaml, C, and their interactions through the OCaml FFI. It extends the single-language recipe for program verification to a multi-language setting as follows:

1. As a starting point, we take simplified versions of OCaml and C called λC and λML (see §2.2) and—following the single-language recipe—we define their canonical operational semantics, →C and →ML. Moreover, again following the recipe, we derive language-specific program logics for them. Since both languages have non-trivial state, we derive two separation logics, called IrisC and IrisML, in the separation logic framework Iris [Jung et al. 2018, 2015].

2. We extend the language-local program logics IrisC and IrisML with reasoning principles for “external calls”. That is, functions potentially implemented in another language (e.g., in C) are exposed to the language-local logic (e.g., to IrisML) through an interface. Conceptually, for each external call, the interface gives a precondition and a postcondition surrounding the call. (We do not add any rules to →C and →ML).

3. Finally, we develop an operational semantics for the combined language, −→ML+C, which embeds the operational semantics of λC and λML and adds reasoning rules to bridge between them. Moreover, we develop a separation logic, IrisML+C, which embeds IrisC and IrisML and extends their reasoning principles to bridge between the two logics. We then prove that IrisML+C is sound with respect to the joint semantics −→ML+C.

In Melocoton, the verification of a mixed OCaml and C program can be done almost entirely in the language-local logics IrisC and IrisML. The notion of “external calls” allows us to abstract over which language is “on the other end” of the call, and the interfaces of external calls (i.e., their pre- and postconditions) are written in the language-local logics. Hence, even for external calls, we do not have to leave the language-local logics (e.g., functions declared external in OCaml, although actually implemented in C, still have an IrisML specification). In fact, even code interacting with the OCaml FFI—on the C side—can stay in the language-local logic IrisC (see §2.4). The only purpose of the “umbrella logic” IrisML+C is to tie together language-local verifications and ensure that the assumptions of one side match up with the guarantees of the other. The key to matching up both sides is providing reasoning rules for what we call the “view reconciliation problem”: reconciling different logical views on the same shared state (see §2.5).

Contributions. The main contribution of this paper is Melocoton, a multi-language program verification system for programs written in OCaml and C. Melocoton consists of the first formal semantics of (a large subset of) the OCaml FFI—previously only described in prose in the OCaml manual [oca 2023b]—and the first program logic to reason about the interactions of OCaml and C.

Melocoton’s operational semantics (§3) is the first multi-language semantics that models interactions with an OCaml-style garbage collector. To define it, we take inspiration from Sammler et al. [2023]: we define an operational semantics that modularly embeds the semantics of λML and λC, and we use angelic and demonic non-determinism to bridge the gap between the two semantics. As mentioned above, the work of Sammler et al. does not cover the kind of language interactions that are possible through the OCaml FFI, which are what makes interoperability between OCaml and C possible.
interesting. To capture them, we use a carefully designed model of the core aspects of the OCaml runtime (e.g., runtime blocks, garbage collection, "roots", callbacks, etc.).

Melocoton’s program logic (§4) is the first separation logic (and program logic) for a multi-language setting with different memory models (including garbage collection). We use separation logic to reason about the state of OCaml and C, and we use step-indexing—inhherited from Iris—to handle the recursive and higher-order features of OCaml. To soundly combine (1) angelic nondeterminism and (2) the higher-order features of OCaml, as it turns out, we have to use a richer form of step-indexing than what is provided out-of-the-box by Iris: we have to use transfinite step-indexing and, thus, define Iris\textsubscript{ML+C} in Transfinite Iris [Spies et al. 2021] (see §4.3).

We explain the key ideas behind Melocoton (in §2) with our running example: importing a compression library from C into OCaml. Besides the compression library, we have applied Melocoton to several interesting examples (in §5): a polymorphic equality function that cannot be implemented natively in OCaml, a version of Landin’s knot [Landin 1964] that mutually recursively goes through the FFI, and an abstract data type that stores OCaml callbacks in C memory. To show that the last two examples—even though they are implemented in C—are type safe (i.e., they can be safely used from arbitrary, well-typed OCaml code), we define a standard logical relation in Iris\textsubscript{ML+C}, extended with reasoning principles for external calls. Melocoton is fully mechanized in Coq, and the Coq development can be found in the supplementary material [Anonymous 2023].

2 MELOCOTON BY EXAMPLE

We explain the key ideas underlying Melocoton using a motivating example: importing a C compression library into OCaml. We focus on the program logic level of Melocoton and follow the spirit of the multi-language recipe: first develop language-local reasoning principles, then focus on language interoperability. Concretely, we discuss the C library and an OCaml client (§2.1), we explain how to reason about them locally (§2.2), we discuss the OCaml FFI “glue code” that ties them together (§2.3), we explain how to reason about the glue code (§2.4), and, finally, we address the view reconciliation problem (§2.5). Throughout this paper, we use colors to distinguish different languages: magenta for OCaml, dark blue for C, and light blue for primitives of the OCaml FFI.

2.1 The C Library and OCaml Client

In this example, depicted in Fig. 2, we want to import a C compression library into OCaml. One can imagine that the implementation of the compression library is particularly efficient in C, or that it already exists as a separate, standalone library and we want to avoid rewriting it. In this particular example, we take inspiration from Google’s Snappy compression library [sna 2023] but, for the purposes of the example, the implementation of the actual compression algorithm is not relevant.

The C library. The function \texttt{snappy\_compress} can be used to compress an input buffer \texttt{inp} (i.e., a raw C pointer to a chunk of bytes) of length \texttt{insz} into an output buffer \texttt{outp} of capacity \texttt{outsz}. After compression, the pointer \texttt{outsz} stores the actual length of the compressed data. To make sure that the capacity of the output buffer \texttt{outp} suffices, the library provides the function \texttt{snappy\_max\_compressed\_length}, which computes an upper bound on the compressed size.

The OCaml client. The OCaml client \texttt{is\_compressible} wants to use the compression library to check whether an array of characters will shrink in size when compressed. When we implement it in OCaml, the first stumbling block that we encounter is that \texttt{is\_compressible} takes in an OCaml array of characters (i.e., a built-in array managed by the OCaml garbage collector), but the C library function \texttt{snappy\_compress} expects two char pointers (i.e., raw pointers to C buffers of bytes). The issue is the that the two types, \texttt{char array} and \texttt{char *}, are not the same—they are not even part of the same language.
To circumvent this problem, we introduce glue code: C code that uses the OCaml FFI to mediate between is_compressible and snappy_compress. For now, we delay a discussion of the C implementation of the glue code (to §2.3) and focus on its OCaml interface (Lines 3-9). The interface consists of (1) a small buffer library (loosely inspired by OCaml’s Bigarray library [oca 2023a]) and (2) wrappers of the compression functions operating on buffers. The interface declares an abstract type bytes, a record for buffers buf, and four functions to operate on buffers: buf_alloc allocates an uninitialized buffer of a given capacity (initially using 0 bytes); buf_free frees a buffer (since memory management is manual in C, we have to free allocated buffers); buf_get updates a range of the buffer using a callback (additionally increasing the number of used bytes if necessary); and buf_get reads a byte of the buffer. The compression functions are wrapped as wrap_max_len (for snappy_max_compressed_length) and wrap_compress (for snappy_compress).

The OCaml client is_compressible uses the glue code to check whether an array of characters can shrink in size as follows: it allocates an input buffer inp and an output buffer outp—large enough to store the compressed input; it then fills the input buffer with the contents of the array; it compresses the input into the output buffer; it checks whether the output size is smaller than the input size; and, finally, it frees the two buffers and returns the result. (The function is_compressible is only illustrating how one can use the external compression function from OCaml. A more realistic client would not “throw away” the compressed buffer like this.)

2.2 Language-Local Reasoning

Before we turn to the implementation of the glue code that connects the OCaml client and the C library, let us first illustrate one of the central ideas of Melocoton: preserving language-local reasoning. Even without knowing the implementation of the glue code (or the OCaml FFI), we can verify the C compression library and the OCaml client already in language-specific program logics. As mentioned in §1, we consider rather idealized versions of C and OCaml in this paper, called $\lambda_C$ and $\lambda_{OC}$.
and \(\lambda_{\text{ML}}\), which are depicted in Fig. 3. We first discuss the language \(\lambda_{\text{C}}\) and verify the compression library in Iris\(_{\text{C}}\). Then, we contrast \(\lambda_{\text{C}}\) with \(\lambda_{\text{ML}}\) and verify the client in Iris\(_{\text{ML}}\).

**The language \(\lambda_{\text{C}}\)**. The essential features of \(\lambda_{\text{C}}\) are its very simple form of values (i.e., integers \(n\) or addresses \(a\)) and its flat memory model. To be precise, memory in \(\lambda_{\text{C}}\) is a finite map from addresses to memory cells, either values or special tokens indicating that an address has been freshly allocated (\(*\)) or has been freed already (†). Memory is allocated with malloc\((n)\) and has to be manually deallocated again with free\((a, n)\). Executing malloc\((n)\) returns an address \(a\) which points to the first of \(n\) consecutive heap cells. To access heap cells other than \(a\), pointer arithmetic can be used (i.e., “\(a + i\)”). Programs \(p\), in \(\lambda_{\text{C}}\), are lists of functions (where no function is defined twice).

![Fig. 3. Syntax, state, and resources of \(\lambda_{\text{C}}\) and \(\lambda_{\text{ML}}\)](image)

**Verifying the C library**. The language \(\lambda_{\text{C}}\) gives rise to Iris\(_{\text{C}}\), a simple, language-specific separation logic, depicted in Fig. 4. In the context of our running example, we use Iris\(_{\text{C}}\) to prove correctness of the two compression library functions. Since Iris\(_{\text{C}}\) is a standard separation logic (e.g., see WRITE-C, READ-C, ALLOC-C, FREE-C, and CALL-INTERNAL), we only discuss the function specifications as the proofs themselves are routine. To verify a function in Iris\(_{\text{C}}\), one proves Hoare triples of the form “\(\{P\} \text{call fn } \overline{w} @ p, \Psi \{Q\}_C\)” where call fn \(\overline{w}\) is the function call construct of \(\lambda_{\text{C}}\), \(p\) is the surrounding program, \(P\) and \(Q\) are the pre- and postconditions, and \(\Psi\) is an “interface”. (Interfaces are only used for external calls to other languages—we will ignore them for now and come back to them shortly.) For the function snappy_max_compressed_length, we show

\[
\{n \geq 0\} \text{call snappy_max_compressed_length } [n] @ p_{\text{lib}}, \emptyset \{w. \exists m. w = m \cdot \text{maxlen}(n, m)\}_C.
\]

where \(p_{\text{lib}}\) is the surrounding compression library (and the interface is the empty interface \(\emptyset\)). Since we are not concerned with the details of the compression algorithm, we keep the upper bound (and other details of compression) abstract in the form of predicates (here “\(\text{maxlen}\)”).

For snappy_compress, we want to prove that if we pass in two buffers—one containing a sequence of integers to compress and one large enough to store the output—the compressed version of the input is stored in the output buffer. To make this intuition formal, we can use the points-to assertion \(a \rightarrow_c cl\) of Iris\(_{\text{C}}\). It conveys ownership over the memory address \(a\) and asserts that it currently

\[\text{In the Coq mechanization, we do not consider the actual Snappy compression algorithm, because its verification is besides the point of this paper. Instead, we verify a toy algorithm which can, in the worst case, double the size of the input.}\]
stores the memory cell \( cl \). For \texttt{snappy_compress}, we show

\[
\{bf(a_{in}, n_{in}, \overline{m}_{in}) \ast bf(a_{out}, n_{max}, []) \ast a_{outsz} \rightarrow_c n_{max} \ast \maxlen(n_{in}, n_{max})
\]

\[
call \texttt{snappy_compress} [a_{in}, n_{in}, a_{outsz} \rightarrow_c] \rightarrow p_{lib}, \emptyset
\]

\[
\{w', w = 0 \ast \exists \overline{m}_{out}, n_{out} \ast bf(a_{in}, n_{in}, \overline{m}_{in}) \ast bf(a_{out}, n_{max}, \overline{m}_{out}) \ast cpr(\overline{m}_{in}, \overline{m}_{out}) \ast a_{outsz} \rightarrow_c |\overline{m}_{out}|\}_{C}
\]

where \( bf(a, n, [m_0, \ldots, m_{k-1}]) \triangleq (\ast_{0 \leq i < k} (a + i) \rightarrow_c m_i) \ast (\ast_{k \leq i < n} (a + i) \rightarrow_c \emptyset) \) \( k \leq n \) asserts that \( a \) stores a buffer of capacity \( n \) with the first \( k \) cells storing the integers \( m_0, \ldots, m_{k-1} \), and we write \( a \rightarrow_c \emptyset \) whenever the contents of \( a \) are irrelevant and \( a \) has not been freed yet. In other words, we prove that if we pass in a sufficiently large output buffer \( a_{out} \), then compression is successful (i.e., return value 0), \( a_{out} \) will afterwards store the compressed version of the input \( a_{in} \) (captured by \( cpr(\overline{m}_{in}, \overline{m}_{out}) \)), and \( a_{outsz} \) will store the length of the compressed buffer.

The language \( \lambda_{ML} \). Let us now turn to \( \lambda_{ML} \). In contrast to \( \lambda_c \), the language has a very rich notion of values: integers, locations, booleans, unit, pairs, sums, foreign values "\( \emptyset \)" (discussed shortly), and closures. Moreover, the memory model of \( \lambda_{ML} \) is very different. The heap of \( \lambda_{ML} \) is a map from (abstract) locations to lists of values. Superficially, this may seem similar to the memory of \( \lambda_c \). However, there are several crucial differences: First, memory management in \( \lambda_{ML} \) relies on garbage collection, meaning memory is allocated on the heap with \texttt{alloc} \( n \) \( V \), but never has to be freed manually. Instead, conceptually, memory is garbage collected once no part of the program can access it anymore. Second, the memory of \( \lambda_{ML} \) is more complex, because it can store (lists of) arbitrary values, regardless of their shape or size (e.g., \( (n,m), \text{rec \( f \) \( x \), \( e \))}, a pair of another pair and a closure). Third, locations in \( \lambda_{ML} \) store entire lists of values. Each location \( \ell \) models a mutable array, whose length can be determined with \texttt{length} \( \ell \), whose elements can be retrieved with \texttt{\( \ell \)(\( n \))}, and whose elements can be updated with \texttt{\( \ell \)(\( n \)) \leftarrow V} \). In contrast to \( \lambda_c \), there is no "address arithmetic" (i.e., the expression \( \ell + i \) is stuck). A third, minor difference to \( \lambda_c \) is that there are no top-level function declarations in \( \lambda_{ML} \) (i.e., \( p = \emptyset \)) and, instead, in \( \lambda_{ML} \) we execute single expressions \( e \) (which internally can contain let-bindings and mutual recursion).
There are three aspects of $\lambda_{\text{ML}}$ that enable language interoperability. They are non-intrusive and minimal such that $\lambda_{\text{ML}}$ does not even (need to) know which languages it is interacting with. First, there are foreign values "$\mathbb{O}$", where $i$ is an abstract identifier. They can be used by other languages such as $\lambda_C$ to "embed" their own data into $\lambda_{\text{ML}}$. We will use this feature of OCaml to embed the underlying C buffer into OCaml in our running example (in §2.5). Foreign values are abstract: they can be passed around via function calls, but there is no language construct in $\lambda_{\text{ML}}$ to inspect their contents. Second, after executing an external function, locations in the heap of $\lambda_{\text{ML}}$ can store data that is temporarily inaccessible from $\lambda_{\text{ML}}$ (see §3.1). Whenever this is the case, the $\lambda_{\text{ML}}$ heap stores $i$ at the respective location and accesses in $\lambda_{\text{ML}}$ will get stuck. Third, and most interestingly, $\lambda_{\text{ML}}$ contains a language construct for external function calls call $\text{fn } \mathbb{V}$. Importantly, the operational semantics does not assign any meaning to it (i.e., call $\text{fn } \mathbb{V}$ is stuck). By choosing to be stuck, $\lambda_{\text{ML}}$ does not have to include any rules in its semantics for executing code of other languages (e.g., $\lambda_C$).

Verifying the OCaml client. Let us turn to the verification of is_compressible in $\text{Iris}_{\text{ML}}$. We prove that given ownership over an array of integers, the function returns an unspecified boolean, meaning $(\ell \mapsto_{\text{ML}} \mathbb{n}) \text{ is_compressible } \ell \circ \mathbb{V}, \psi_{\text{buf}} \{V', V' \in \{\text{true}, \text{false}\} \ast \ell \mapsto_{\text{ML}} \mathbb{n}\}_{\text{ML}}$. This specification is not particularly exciting, but it suffices to illustrate the interaction with the external functions implemented in C. The resources of $\text{Iris}_{\text{ML}}$ (e.g., "$\ell \mapsto_{\text{ML}} \mathbb{V}'\)" and its reasoning rules (e.g., READ-ML, WRITE-ML, and ALLOC-ML) are standard. Thus, we focus mainly on the treatment of external function calls. External calls allow us to slice the verification of a program into the language-local parts and the parts that are implemented in another language. In is_compressible, the first time that we call an external function is when we allocate the initial buffers inp and outp with buf_alloc (Line 12). Suppose (for a moment), for the sake of explanation, that buf_alloc was implemented in OCaml. In this hypothetical scenario, the standard way to proceed would be to prove (once) and afterwards apply (multiple times) the Hoare triple:

$$\{ n > 0 \} \text{buf_alloc}(n) \circ \mathbb{V}, \psi_{\text{buf}} \{V, \text{buffer}_{\text{ML}}(V, [\ell])\}_{\text{ML}}$$

where $\text{buffer}_{\text{ML}}(V, n, m) \triangleq \exists r, V'. V = (n, \ell, V') \ast \ell \mapsto_{\text{ML}} \mathbb{m} \ast \text{bytes}(V', n, m)$ asserts that $V$ is a tuple containing the buffer capacity $n$, a $\lambda_{\text{ML}}$ reference $\ell$ for the used field, and a value $V'$ for containing the underlying integers $m$. How $V'$ stores the integers $m$ is irrelevant for the verification of is_compressible, meaning we can stay at the abstraction of "$\text{bytes}(V', n, m)\)". (As we will see in §2.5, the definition of bytes uses $\lambda_{\text{ML}}$'s "foreign values" and knowledge about the OCaml FFI.) After applying the Hoare triple, we can then use the buffer that buf_alloc returns to resume the verification of is_compressible.

Of course, buf_alloc is not actually implemented in OCaml. Nevertheless, in our proof of is_compressible, we want to stay as close as possible to the language-local reasoning sketched out above. To do so, we introduce reasoning principles to "skip" external function calls by drawing a boundary at their specification. For example, in the verification of is_compressible, we want to prove the precondition $n > 0$ of buf_alloc to call it and then resume afterwards with the postcondition $\text{buffer}_{\text{ML}}(V, [\ell])$. To make this reasoning sound, we take inspiration from the work on open simulations [Hur et al. 2012] and of de Vilhena and Pottier [2021] on a program logic for effect handlers. Concretely, we parameterize Hoare triples by an interface $\Psi \in \text{Intf}(\text{Val}) \triangleright \text{FnName} \to \text{list(Val)} \to (\text{Val} \to i\text{Prop}) \to i\text{Prop}$ that maps each external call to its specification. Formally, an interface $\Psi$ is a "predicate transformer" that takes in a function name $\text{fn}$, a list of argument values $\mathbb{a}$, and a postcondition $Q$ and then produces the precondition "$\Psi \text{ fn } \mathbb{a} Q\)" required to call $\text{fn}$ with arguments $\mathbb{a}$. The way that "skipping" function calls works with predicate transformers (see CALL-EXTERNAL) is that we show that the current precondition $P$ implies $\Psi \text{ fn } \mathbb{a} Q$ where $Q$ is our desired postcondition.
value buf_alloc(value cap) {
    CAMLparam1(cap); CAMLlocal3(bk, bf, r);
    r = caml_alloc(1, 0); // allocate the 'used' reference block
    Store_field(r, 0, Val_int(0));
    bk = caml_alloc_custom(sizeof(void*)); // allocate the 'data' custom block
    Custom_contents(bk) = malloc(Int_val(cap));
    bf = caml_alloc(3, 0); // allocate the (cap, used, data) block
    Store_field(bf, 0, cap); Store_field(bf, 1, r); Store_field(bf, 2, bk);
    CAMLreturn(bf);    } } 

Fig. 5. FFI glue code for buf_alloc

For example, in the case of the buffer library, the interface
$\Psi_{buf} \triangleq \Psi_{buf_alloc} \sqcup \Psi_{buf_free} \sqcup \cdots \sqcup \Psi_{wrap_max_len}$
specifies all the library functions as a disjunction of single-function interfaces, where
$(\Psi_1 \sqcup \Psi_2)\_fn\_\phi \triangleq \Psi_1\_fn\_\phi \lor \Psi_2\_fn\_\phi$. For buf_alloc specifically, the interface is:

$\Psi_{buf_alloc}\_fn\_\phi \triangleq \exists n.\_fn = buf_{alloc} \land \_\phi = [n] \land n > 0 \ast (\forall V.\_buf_{nm}(V, n, []) \rightarrow \Phi V)$

which we, more idiomatically, write as $\Psi_{buf_alloc} \triangleq \forall n. \_buf_{alloc} [n] \_buf_{nm}(V, n, [])$ (note the angle brackets!) to give single-function interfaces their familiar “Hoare triple reading”.

With the interface $\Psi_{buf}$ for the library, the verification of is_compressible in fribols proceeds smoothly as if the functions were implemented in $\lambda_{BML}$ (using CALL-EXTERNAL).

2.3 A Primer of the OCaml FFI

Having verified the OCaml client and the C compression library (from Fig. 2), we now turn to the “glue code” that connects them. Before we can verify any glue code (in §2.4), we first have to understand the central concepts of the OCaml FFI and how they are used in our example. To explain them, we focus on the implementation of buf_alloc (in Fig. 5); the implementation of the remaining glue code functions can be found in the supplementary material [Anonymous 2023].

The representation of OCaml values. In C, all OCaml values—regardless of their type, including buf, bytes, int, and char array—are exposed by the OCaml runtime as runtime values of type value. They are either integers or pointers to runtime blocks (i.e., chunks of memory in the heap of the OCaml runtime). Integers are used to encode OCaml’s integers (i.e., int) and other simple types such as bool (using 1 for true and 0 for false). Pointers are used to encode OCaml’s structured values (e.g., pairs, data types with arguments, arrays, etc.). For example, as depicted in Fig. 1, a value of the buffer record is represented as a pointer to a runtime block with three values, one storing an integer for the capacity, one storing a reference for the used field, and another storing the underlying C buffer. To embed the C buffer—a raw C pointer which is not of type value—into OCaml, the runtime offers so-called “custom blocks”. Custom blocks are runtime blocks that embed native C data (e.g., pointers) into OCaml as foreign values.

Footnotes: 3Formally, the notation is defined as $\forall x. \langle P \_fn \_\phi, \exists y, Q \rangle \triangleq \lambda f'n' \_\phi'. f'n' = f'n \ast \exists x. \_\phi = \exists y. \_phi' = \exists x. \_\phi \ast \exists y, Q \rightarrow \phi v$. The reader may cheerfully ignore this definition and stick with the intuition of Hoare triples as specifications.

4In Melocoton, we model a simplified form of custom blocks: In OCaml, they can store arbitrary C data, which is accessed with Data_custom_val. Here, they always store a C pointer, which is accessed with Custom_contents. Moreover, in OCaml custom blocks can be given additional parameters such as comparison methods during allocation, which we omit here.
Manipulating OCaml values in C. As a function using the OCaml FFI, `buf_alloc` has to interact correctly with the OCaml runtime primitives (in light blue) and the OCaml garbage collector. We will ignore the garbage collector for now—and the primitives for working with it in Line 17 and `CAMLreturn` in Line 24—and come back to it below. Instead, we focus on how to operate on runtime values. For integers, converting between an OCaml integer (as a runtime value) and a C integer is simple: the runtime provides the primitives `Val_int` (read “integer to value”, see Line 19) and `Int_val` (read “value to integer”, see Line 21). Interacting with structured values is more subtle. Recall the runtime representation of the buffer record `{cap, used, data}` depicted in Fig. 1. To create it, `buf_alloc` proceeds as follows: it allocates a runtime block `r` of size one for the reference used using the primitive `caml_alloc` (in Line 18) and initializes it to zero using the primitive `Store_field` (in Line 19); it allocates a “custom” runtime block `bk` (in Line 20) using `caml_alloc_custom`; it allocates a C buffer of the right length and stores it in the custom block (in Line 21); it allocates a block for the `buf` record and stores the capacity (here `len`), the used reference, and the custom block in it (in Lines 22-23); and, finally, it returns the newly created runtime block (in Line 24).

Tiptoeing around the OCaml garbage collector. One fundamental difference between OCaml and C reveals itself only implicitly in the code in Fig. 2: the presence of the OCaml garbage collector (GC). That is, memory management in OCaml is based on garbage collection, meaning once a runtime allocated object (i.e., one that is represented as a runtime block) becomes unreachable (i.e., no part of the OCaml code can access it anymore), the GC can deallocate it to reduce the amount of memory consumed. This is in stark contrast to the manual memory management of C, where memory has to be explicitly allocated and eventually deallocated.

One immediate consequence of this difference is that the buffer library needs to provide a `buf_free`-function to OCaml (Line 5), because otherwise the buffer will stay in memory forever. A more burdensome and subtle consequence is that we have to make sure that the GC does not invalidate references to runtime blocks that we still want to use. That is, when we define an external function in C, the GC may execute whenever we make calls to (certain) OCaml FFI runtime functions (e.g., `caml_alloc`). To prevent it from invalidating our local references (by deallocating or moving the runtime blocks that they point to), we have to “register” our local references with the GC as so-called “roots” (using `CAMLlocal` and `CAMLparam` in Line 17). Eventually, when we no longer need them, we can unregister our roots with the GC again (using `CAMLreturn` in Line 24).

### 2.4 An Interface for the OCaml FFI

How should we verify glue code such as `buf_alloc` in Fig. 5? On the one hand, the code is written in C, so it would be natural to use the language-local logic Iris\textsubscript{C} to reason about it. On the other hand, semantically, the code is more concerned with OCaml values (and the runtime representation thereof) than with C data structures, C pointers, and C-specific features. We answer this question with our next key idea: importing a logic for the OCaml FFI into the language-local logic Iris\textsubscript{C}.

Conceptually, “the OCaml FFI” is (1) a lower-level model of OCaml’s values and heap that is exposed to C; and (2) a set of “runtime primitives” available to C to operate on this lower-level representation. Together, these two parts of the OCaml FFI form a clean abstraction over the actual, underlying C data representation (and the implementation of the garbage collector). We use them as a “middle ground” between C and OCaml in this work. Concretely, (1) we define a notion of runtime values, runtime heaps, and runtime separation logic resources; and (2) we specify the runtime primitives as external functions in Iris\textsubscript{C} using an interface $\Psi_{FFI}$. In doing so, we get access to abstract reasoning principles about the OCaml runtime while retaining the language-local reasoning principles of Iris\textsubscript{C}. 
The resources of the OCaml runtime. To define the runtime interface, we take the view that memory exposed by the OCaml FFI is an abstract heap of runtime blocks and then relate this heap to both its C and OCaml representations (i.e., map it to pointers and integers in C and map it to structured values and references in OCaml). We will describe the interface of the OCaml runtime in more detail in §4.2. For now, we just take a glimpse at its values, resources, and rules.

The values of the OCaml runtime $v \in Val := (n \in \mathbb{Z}) \mid (\gamma \in Loc)$ are either integers or abstract runtime locations $\gamma$. Runtime locations “store” runtime blocks, which we track through separation logic resources: For standard blocks, we have $\gamma \mapsto \text{blk}[t|m] \triangleright$, which says that $\gamma$ currently stores a block of values $\triangleright$. It, additionally, asserts that the tag $\triangleright$ of the block is $t$ and that the mutability of the block is $m \in \{\text{imm}, \text{mut}, \text{fresh}\}$: blocks can be immutable imm (e.g., for OCaml pairs), mutable mut (e.g., for OCaml references), or fresh fresh (i.e., it has not been decided yet whether $\gamma$ will be a mutable or immutable value). For “custom” blocks, which embed C data into OCaml, we have $\gamma \mapsto \text{cstm} w$, which says that $\gamma$ currently stores the $\lambda_c$-value $w$. Custom blocks are always mutable.

To use these abstract runtime values $v$ from $\lambda_c$, we relate them to concrete $\lambda_c$-values $w$. This correspondence is the relation $v \sim_\theta^C w$:

$$v \sim_\theta^C w \triangleq (\exists n. v = n \land w = \hat{n}) \lor (\exists \gamma. v = \gamma \land w = \theta(\gamma)) \quad \text{where } \theta \in \text{AddrMap} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Addr}$$

It takes in a finite map $\theta$ from runtime locations to $\lambda_c$-addresses, and then relates runtime integers $n$ with the $\lambda_c$-value $\hat{n}$ representing them (where $\hat{\cdot}$ translates integers to $\lambda_c$-values using the encoding also used by Val_int) and runtime locations with their $\lambda_c$-addresses. The map $\theta$ allows us to model the behavior of the garbage collector (i.e., moving and deallocating runtime blocks). It is not constant. Instead, the GC may (1) move blocks in memory, which changes their physical address in $\lambda_c$ (i.e., $a$ changes) but not their “identity” in the runtime (i.e., $\gamma$ remains unchanged) and (2) deallocate blocks that have become unreachable, which keeps the block in the abstract runtime (i.e., $\gamma$ is not removed from the runtime heap), but its physical address vanishes. To keep track of the current map $\theta$, we introduce an additional separation logic resource $\text{GC}(\theta)$, which asserts that the current map from runtime locations to $\lambda_c$ addresses is $\theta$.

The interface of the OCaml runtime. To understand how these resources are used, let us take a look at the interface for the caml_alloc runtime primitive (which is a disjunct of $\Psi_{\text{FFI}}$):

$$\Psi_{\text{alloc}} \triangleq \forall \theta, n, m. (\text{GC}(\theta) \star 0 \leq n) \text{ alloc } [n,m] \langle w. \exists \theta', y. \text{GC}(\theta') \star y \mapsto \text{blk}[m|\text{fresh}] \theta^n \land y \sim_\theta^C w \rangle$$

The interface enables us to allocate a block of (non-negative) length $n$ with tag $m$. To do so, we have to provide the GC resource $\text{GC}(\theta)$ with some address map $\theta$ initially. Since the allocation primitive caml_alloc calls the garbage collector internally, this map $\theta$ is potentially changed during the allocation, and we get back a new address map $\theta'$ after the call (and the GC resource). Moreover, for the newly allocated block, we get a fresh runtime location $\gamma$, which “stores” $n$-consecutive zeros as runtime values. To enable us to use the new block from $\lambda_c$, the return value $w$ is related by $\gamma \sim_\theta^C w$ to the freshly allocated runtime location $\gamma$ in the postcondition. (We will see more runtime primitives and how to maintain references to $\lambda_ml$ values across GC calls in §4.2.)

Using the interface $\Psi_{\text{FFI}}$, we can verify glue code in Iris_c. In particular, for buf_alloc, we prove:

$$\{ \text{GC}(\theta) \star n \sim_\theta^C w \star n \geq 0 \} \text{ call buf_alloc } [w] @ \text{pbuf}_{\text{FFI}} \{ w'. \exists \theta', y. \text{GC}(\theta') \star y \sim_\theta^C w' \land \text{buf}_{\text{RT}}(y, n)[] \}$$

where $\text{buf}_{\text{RT}}(y, n, \hat{m}) \triangleq \exists a.i. d. y \mapsto \text{blk}[0|\text{imm}] [n. y_i. y_d] \star y_0 \mapsto \text{blk}[0|\text{mut}] [0] \star y_1 \mapsto \text{cstm} a \ast \text{bf}(a, n, \hat{m})$ encodes the runtime representation of a buffer, as illustrated in Fig. 1—albeit without tags. It contains ownership of (1) the “buffer record” in runtime representation $y \mapsto \text{blk}[0|\text{imm}] [n. y_i. y_d]$. This is the record that stores the buffer size $n$ and its start address $y$.

---

5Tags are how OCaml distinguishes between different constructors of a datatype. In our examples, they are typically 0.
We have sketched how to verify a mixed OCaml-and-C program from the OCaml side using Iris separation logic, which means they intentionally do not model C. Concretely, the rule Iris depicted in Fig. 6. They allow us to use view reconciliation principles to enforce that, at any given point, there is only a single view—either through the lense of OCaml or of the runtime—on the underlying state (e.g., in wrap_compress), the new value becomes also observable on the other side (e.g., in is_compressible).

The fact that there are two ways of describing the same physical state means we have to make sure that they stay “in sync” (e.g., $y_u \leftrightarrow_{\text{blk}[0]\text{[mut]} [0]$ and $\ell \leftrightarrow_{\text{ML}} [1]$ would be inconsistent). Here, we can reap the benefits of working in separation logic. Using the notion of exclusive ownership, we can enforce that, at any given point, there is only a single view—either through the lens of OCaml or of the runtime—on the underlying state (i.e., $y_u \leftrightarrow_{\text{blk}[0]\text{[mut]} [0]$ and $\ell \leftrightarrow_{\text{ML}} [0]$ are mutually exclusive). We can transition between the two perspectives using the view reconciliation rules, depicted in Fig. 6. They allow us to use Iris resources when we are verifying C glue code in Iris.

Concretely, the rule ML-TO-FFI allows us to turn $\ell \leftrightarrow_{\text{ML}} \bar{V}$ (whenever we own the GC resource $GC(\ell)$ into the runtime block $y \leftrightarrow_{\text{blk}[0]\text{[mut]} \bar{\delta}$ using Iris’s resource updates [Jung et al. 2018, §5.4]. The runtime block location $y$ is tied to the $\lambda_{ML}$ location $\ell$ through a new assertion $\ell \sim_{\text{ML}} y$, which makes sure that $y$ always (uniquely) corresponds to $\ell$. To relate the values of the runtime block $\delta$ and the reference $\bar{V}$, the relation $\sim_{\text{ML}}$ is lifted to values as $\bar{V} \sim_{\text{ML}} \bar{\delta}$ (e.g., by relating $false \sim_{\text{ML}} 0$). The rule FFI-TO-ML reverses ML-TO-FFI: we can take ownership of a runtime block and turn it (back) into ownership of a $\lambda_{ML}$ reference. The assertion $\ell \sim_{\text{ML}} y$ appears after the update, because we can use this rule also to expose blocks that were freshly created in C (e.g., in buf_alloc) to OCaml.

With the view reconciliation rules in hand, we can finally tie the knot. We will discuss the formal details of connecting Iris and Iris in §4. For now, let us use buf_alloc to illustrate the key steps for our running example. Concretely, we need to match up the two buffer representations in the postconditions of buf_alloc, bufferML $(V, n, \bar{m})$ and bufRT $(y, n, \bar{m})$. To do so, we prove the following rules:

\[
\begin{align*}
\text{GC}(\ell) \cdot \ell \leftrightarrow_{\text{ML}} \bar{V} \equiv \exists \bar{\delta}, y. \ \text{GC}(\ell) \cdot y \leftrightarrow_{\text{blk}[0]\text{[mut]} \bar{\delta} \cdot \ell \sim_{\text{ML}} y \cdot \bar{V} \sim_{\text{ML}} \bar{\delta} & \quad \text{(ML-TO-FFI)} \\
\text{GC}(\ell) \cdot y \leftrightarrow_{\text{blk}[0]\text{[mut]} \bar{\delta} \cdot \bar{V} \sim_{\text{ML}} \bar{\delta} \equiv \exists \ell. \ \text{GC}(\ell) \cdot \ell \leftrightarrow_{\text{ML}} \bar{V} \leftrightarrow_{\text{ML}} y & \quad \text{(FFI-TO-ML)}
\end{align*}
\]
following view reconciliation rule:

\[ GC(\theta) * \text{buf}_\text{RT}(\gamma, n, \tilde{m}) \Rightarrow \exists V. GC(\theta) * \text{buffer}_\text{ML}(V, n, \tilde{m}) * V \sim_{\text{ML}} \gamma \]

which we can use to turn \text{buf}_\text{RT} into \text{buffer}_\text{ML}. The proof of this rule consists of three parts: First, we turn the runtime block \( \gamma_u \mapsto \text{blk}[\theta[\text{int}]] [n] \) inside \text{buf}_\text{RT} into the mutable used-reference \( \ell \mapsto_{\text{ML}} [n] \) inside \text{buffer}_\text{ML} (using FFI-TO-ML). Second, we turn the custom block \( \gamma_d \mapsto_{\text{cstim}} a * \text{bf}(a, n, \tilde{m}) \) inside \text{buf}_\text{RT} into bytes \( (V', n, \tilde{m}) \) inside \text{buffer}_\text{ML}. This is the place where we define bytes, which was treated axiomatically while verifying \text{is_compressible}. Concretely, we define \( \text{bytes}(V', n, \tilde{m}) \triangleq \exists \gamma_d, \ell, a. V' = (\ell * (D) \sim_{\text{ML}} \gamma_d * \gamma_d \mapsto_{\text{cstim}} a * \text{bf}(a, n, \tilde{m})) \), which means \( V' \) is a foreign value \( D \) that corresponds to the custom runtime block \( \gamma_d \), which stores the address \( a \) of the underlying bytes. (We obtain the foreign identifier \( \ell \) using additional rules given in the supplementary material [Anonymous 2023].) Third, we prove that the runtime block \( \gamma \mapsto \text{blk}[\theta[\text{imm}]] [n, \gamma_u, \gamma_d] \) represents the \( \lambda_{\text{ML}} \) buffer \( V \triangleq (n, \ell, D) \), meaning \( V \sim_{\text{ML}} \gamma \).

With the above view reconciliation rule and its reverse (i.e., turning \text{buffer}_\text{ML} into \text{buf}_\text{RT}), we can verify all the glue code of the buffer library (see [Anonymous 2023]). We can then finally connect all the puzzle pieces (as described in §4) to conclude that \text{is_compressible} is correct, which in particular means (1) the glue code correctly interacts with the FFI and maintains all the language invariants of \( \lambda_{\text{ML}} \), and (2) that the client \text{is_compressible} uses the function \text{snappy_compress} correctly without triggering any unsafe behavior in \( \lambda_C \) (e.g., no out-of-bounds accesses).

3 OPERATIONAL SEMANTICS

Zooming out, let us return to the multi-language recipe from §1. In this section, we focus on the operational semantics side of the recipe. As a starting point, we take the canonical small-step operational semantics for the languages \( \lambda_C \) and \( \lambda_{\text{ML}} \) (defined in the supplementary material [Anonymous 2023]). We write \( (e, \sigma) \rightarrow_{\text{ML}} (e', \sigma') \) for a step in \( \lambda_{\text{ML}} \) and \( p; (c, \sigma) \rightarrow_{\text{C}} (c', \sigma') \) for a step in \( \lambda_C \) (where \( p \) is the surrounding \( \lambda_C \)-program). Since these semantics operate on different values and have different memory models, they cannot be simply "plugged together." Thus, the big question is how can we connect the two semantics to a multi-language semantics \( \rightarrow_{\lambda_{\text{ML+C}}} \)?

Structure of the semantics. To define the semantics \( \rightarrow_{\lambda_{\text{ML+C}}} \), we take inspiration from Sammler et al. [2023]: we define it as a composition of smaller building blocks. In our setting, each building block is a language \( \lambda \) with an associated notion of expressions \( e \in \text{Expr} \), values \( v \in \text{Val} \), state \( \sigma \in \text{State} \), functions \( f \in \text{Func} \), programs \( p \in \text{FnName} \mapsto \text{Func} \), and a small-step operational semantics \( \rightarrow \). Two languages \( \lambda_A \) and \( \lambda_B \) interact through external function calls \( \text{call fn } \overline{v} \).

Our basic building blocks are, as one would expect, the semantics \( \lambda_C \) and \( \lambda_{\text{ML}} \). To connect two languages \( \lambda_A \) and \( \lambda_B \) through their external calls, we introduce a linking combinator \( p_A \oplus p_B \) with its associated language \( \lambda_{A\oplus B} \) (and semantics \( \rightarrow_{A\oplus B} \)). The linking combinator only composes languages with the same values and memory model. Thus, we cannot directly link a \( \lambda_{\text{ML}} \) expression \( e \) and a \( \lambda_C \) program \( p \), since their languages differ in their values and memory model (see Fig. 3). To bridge the gap between them, we introduce a wrapping combinator \( "[\cdot]_{\text{FFI}} \) that embeds the language \( \lambda_{\text{ML}} \) into its own language \( \lambda_{\text{ML}}^\text{FFI} \) (with semantics \( \rightarrow_{\lambda_{\text{ML}}^\text{FFI}} \)). The wrapper takes a \( \lambda_{\text{ML}} \)-expression \( e \) and produces a program \( [e]_{\text{FFI}} \) with \( \lambda_C \)-values and the \( \lambda_C \)-memory model. However, the program \( [e]_{\text{FFI}} \) is not a \( \lambda_C \)-program; it is a program in the language \( \lambda_{\text{ML}}^\text{FFI} \) which uses \( \lambda_C \)-values and the \( \lambda_C \) memory model, but has a very different notion of expressions and semantics (see §3.1).

Putting everything together, we obtain the combined language \( \lambda_{\text{ML+C}} \triangleq \lambda_{\text{ML}}^\text{FFI} \oplus \lambda_C \) with its operational semantics \( \rightarrow_{\lambda_{\text{ML+C}}} \triangleq \rightarrow_{\lambda_{\text{ML}}^\text{FFI} \oplus \lambda_C} \) and its programs \([e]_{\text{FFI}} \oplus p \) where \( e \) is a \( \lambda_{\text{ML}} \)-expression and \( p \) a \( \lambda_C \)-program. In the semantics \( \rightarrow_{\lambda_{\text{ML+C}}} \), external calls of \( e \) are made in terms of \( \lambda_{\text{ML}} \)'s values and memory model. They get translated by the wrapper to external calls of \( \lambda_C \), which are then resolved.
to the actual function implementations within \( p \) through linking. In the other direction, \( p \) can make external calls to FFI functions, which will be resolved by the wrapper in its operational semantics.

**Bridging language barriers with angelic non-determinism.** The wrapper \([\cdot]\)FFI bridges the gap between \( \lambda_{\text{ML}} \) and \( \lambda_{\text{C}} \) at the level of the operational semantics. Doing so is not straightforward for a number of reasons (see also §3.1). In particular, when going from \( \lambda_{\text{C}} \) to \( \lambda_{\text{ML}} \) (e.g., when \( \lambda_{\text{C}} \) invokes a \( \lambda_{\text{ML}} \)-callback or returns to \( \lambda_{\text{ML}} \)), the wrapper needs to translate values from their low-level runtime representation to their high-level \( \lambda_{\text{ML}} \) representation. The issue is that this translation is, unfortunately, not unique! Different \( \lambda_{\text{ML}} \)-values \( V \) can have the same runtime representation \( \varnothing \) (e.g., integers and booleans are both runtime integers; pairs and arrays are both runtime blocks, etc.). When we go from \( \lambda_{\text{C}} \) to \( \lambda_{\text{ML}} \) in the wrapper, we have to choose “the right” high-level representation. Which one is the right one is only known to the programmer who wrote the glue code.\(^6\)

To make “the right” choice, we follow the lead of Song et al. [2023] and Sammler et al. [2023]: we use **angelic non-determinism** (together with the usual demonic non-determinism). Instead of forcing the wrapper to make the right choice (as would be the case for demonic non-determinism), angelic non-determinism rules out all the wrong choices. Concretely, we use multirelations [Martin et al. 2007; Rewitzky 2003], written \( p; s \rightarrow S \), as steps in the operational semantics. Here, \( s \) is a configuration of the operational semantics (i.e., a pair of an expression \( e \) and the state \( \sigma \)) and \( S \) a set of configurations from which to continue the execution after the step. We interpret the choice of the set \( S \) as **angelic non-determinism**, and the choice of the configuration \( s' \in S \) to resume from as **demonic non-determinism**.

The operational semantics \( \rightarrow_{\text{C}} \) and \( \rightarrow_{\text{ML}} \) use—as usual—only demonic non-determinism. Their semantics can be lifted in a generic way to multirelations \( \rightarrow_{\text{C}} \) and \( \rightarrow_{\text{ML}} \) (note the double arrow!). We define the semantics of the wrapper \([\cdot]\)FFI and linking combinator \( "\oplus" \) in terms of multirelations directly. The definition can be found in the supplementary material [Anonymous 2023]. Instead of belaboring it here, we focus (in §3.1) on the most interesting part: how the wrapper deals with the **view reconciliation problem** (see §2.5) at the level of the operational semantics.

### 3.1 View Reconciliation in the Wrapper Semantics

To explain view reconciliation in the wrapper, we need to understand, at a high level, how the wrapper is set up: The purpose of the wrapper is to produce a program \([\cdot]\)FFI that (1) faithfully executes the code of \( e \) in the semantics of \( \lambda_{\text{ML}} \) and (2) provides a model of the runtime primitives such that they can be called from \( \lambda_{\text{C}} \). The program \([e]_{\text{FFI}} \) does not have to be a **syntactic** \( \lambda_{\text{C}} \)-program. Instead, since the linking combinator “\( \oplus \)” connects languages through external calls \( \text{call fn } \bar{v} \), it only needs to provide “functions” that can be called using \( \lambda_{\text{C}} \)-values in the \( \lambda_{\text{C}} \)-memory model. We use this freedom in the semantics \( \rightarrow_{[\text{ML}]_{\text{FFI}}} \) by implementing the “functions” of \([e]_{\text{FFI}} \) (i.e., the primitives of the FFI and a **main** function that triggers the execution of \( e \)) as operations on an internal notion of **wrapper state**.

**The wrapper state.** The wrapper state \( p \in \text{State} \) (in Fig. 7) has two execution modes: \( \text{ML} \) and \( \text{C} \). While it is executing the wrapped expression \( e \), its state is \( \text{ML} \) \( ((\zeta, \chi, \sigma, \text{rm}), \sigma) \) where \( \sigma \) is the current \( \lambda_{\text{ML}} \)-state and the remaining state is part of the runtime state (discussed shortly). The wrapper transitions to the \( \text{C} \) execution mode whenever we are at the boundary between \( \lambda_{\text{C}} \) and \( \lambda_{\text{ML}} \) (e.g., \( \lambda_{\text{C}} \)-function, or we are executing a FFI primitive called from \( \lambda_{\text{C}} \)). In the \( \text{C} \) execution mode, the state of the wrapper is \( \text{C} \) \( ((\zeta, \chi, \theta, \text{rs}), \sigma) \) where \( \sigma \) is the current \( \lambda_{\text{C}} \)-state.

---

\(^6\)We do not use types to guide this choice as our wrapper—like the OCaml runtime—does not track type information and even with type information the choice would be unclear for functions with polymorphic types.
The first constraint, requires the address map \( \theta \) to be transitively closed: for any runtime location \( y \) in \( \theta \), any reachable location \( y' \) must also be part of the map (where \( \text{locs}(b) \) is the set of runtime locations contained in \( b \)). The second constraint ensures that \( \theta \) contains at least all the “registered roots”. Roots, in general, are how the OCaml GC keeps track of which runtime blocks to keep around. In \( \lambda_{\text{ML+C}} \), it suffices to track those roots that have been explicitly registered with the GC through an FFI primitive (explained below). How the other GC roots—which we do not model in \( \lambda_{\text{ML+C}} \)—are determined is left as an implementation choice to the GC so long as it does not deallocate reachable blocks (GC1). In \( \lambda_{\text{ML+C}} \), registered roots are \( \lambda_{C} \)-addresses that store (the \( \lambda_{C} \)-addresses) of runtime roots and the remaining state is again part of the runtime state. In this mode, the \( \lambda_{\text{ML}} \)-heap \( \sigma \) has been dissolved completely into the runtime state.

The runtime state (i.e., \( \zeta, \chi, \theta, \sigma, rm, \) and \( rs \)) is how the wrapper reconciles the views of \( \lambda_{C} \) and \( \lambda_{\text{ML}} \).

It is an abstract model of the runtime heap of blocks (alluded to in §2.4) and the GC roots, together with additional state to relate them to their \( \lambda_{C} \) and \( \lambda_{\text{ML}} \) representation. The map \( \zeta \) stores the current heap of runtime blocks. It is the central piece of state of the runtime: many primitives only operate at the level of the block heap (e.g., allocating new blocks, accessing or modifying existing ones). A block can be either a standard block \( \text{Val}(t, \nu) \) (with tag \( t \), storing values \( \nu \)), a "custom" block \( \text{Custom}(w) \) (storing a \( \lambda_{C} \) value \( w \)), or a "closure" block \( \text{Closure}(\text{rec } f x. e) \) representing a \( \lambda_{\text{ML}} \) function.

The map \( \theta \) maps runtime locations to their \( \lambda_{C} \)-addresses (see also §2.4) if they are materialized in the \( \lambda_{C} \)-memory (i.e., they have not been deallocated by the GC). The map \( \chi \) maps runtime locations to their \( \lambda_{\text{ML}} \) counterpart. A runtime location \( y \) is mapped (1) to a \( \lambda_{\text{ML}} \)-location \( \ell \) if \( y \) is a standard block backing an array, (2) to a "foreign value identifier" \( i \) if \( y \) is a custom block, or (3) to a special token \( * \) if \( y \) is freshly allocated (from \( \lambda_{C} \)) or a block that backs a pure \( \lambda_{\text{ML}} \) value (e.g., pairs or closures). A runtime location \( y \) can be in all three maps at the same time: it can be in \( \chi \) to relate it to a corresponding \( \lambda_{\text{ML}} \)-reference \( \ell \), in \( \zeta \) to track the runtime values \( \nu \) that are stored in \( y \), and in \( \theta \) to assign it an address \( a \) in the \( \lambda_{C} \)-heap. The map \( rm \) and the set \( rs \) are used to keep track of GC "roots" and the memory \( \sigma \) tracks the "residual" \( \lambda_{C} \)-memory (explained below).

The garbage collector and registered roots. In \( \lambda_{\text{ML+C}} \), we do not fix a specific garbage collector implementation. Instead, the semantics of \( \lambda_{\text{ML+C}} \) should be sound with respect to all reasonable garbage collectors. Thus, to model garbage collection in \( \lambda_{\text{ML+C}} \), we use non-determinism. The way it works is that the address map \( \theta \) is only part of the state while we are at the boundary to \( \lambda_{C} \) (i.e., in the mode \( C \)). It is picked anew whenever we move from an execution in \( \lambda_{\text{ML}} \) to an external function in \( \lambda_{C} \) (and for certain runtime primitives such as \text{caml_alloc}). This choice over \( \theta \) is made demonically (i.e., if we verify a program, we have to reason about all possible choices for \( \theta \)). To avoid “over-eager” deallocation, the choice is subject to two constraints that ensure that reachable locations remain alive:

\[
\closed(\theta, \zeta) \triangleq \forall y \in \text{dom } \theta. (b \in \text{Block}) \Rightarrow \exists y' \in \text{locs}(b), y' \in \text{dom } \theta \quad \text{(GC1)}
\]

\[
\roots(\theta, rs) \triangleq \forall a \ y, \text{rm}(a) = y \Rightarrow y \in \text{dom } \theta \quad \text{(GC2)}
\]

Fig. 7. Runtime state of the wrapper.
blocks \( y \), given by \( \theta(y) \). They are tracked in the roots map \( rm \) (and in the set \( rs \) when in C).\(^7\) Their intended behavior is that (1) the block \( y \) is not deallocated by the GC and (2) whenever the runtime moves the block \( y \) in memory, it will also update its address stored in the root. The first part is addressed by (GC2). The second part happens whenever we move from ML to C: The ML wrapper state contains a \( \lambda_C \)-memory \( \sigma \) where all the registered roots have been removed, the "residual" \( \lambda_C \)-memory. When we return to \( \lambda_C \), the wrapper uses the freshly picked address map \( \theta \) to "add in" the \( \lambda_C \)-representation of the roots.

**Angellic non-determinism.** As mentioned above, the angellic non-determinism comes into play when we move from \( \lambda_C \) to \( \lambda_{ML} \). Concretely, when we transition from C ((\( \zeta, \chi, \theta, rs \), \( \sigma \)) to ML ((\( \zeta, \chi, \sigma, rm \), \( \sigma \)), we need to choose \( \lambda_{ML} \)-values \( V \) for runtime values \( v \) in their block representation. However, the block representation is not unique, since, for example, true and 1 have the same runtime representation 1. To pick "the right" value, the semantics uses angellic non-determinism.

There is a second use of angellic non-determinism in the semantics: in transitioning from \( \lambda_C \) to \( \lambda_{ML} \), the wrapper angellicly chooses a subset of the heap \( \sigma \), possibly empty, that it temporarily disables (i.e., it marks the contents with \( \checkmark \)). This use of angellic non-determinism simplifies reasoning about the semantics: Locations that are not accessed on the \( \lambda_{ML} \)-side can be disabled, which avoids (unnecessarily) committing to a concrete \( \lambda_{ML} \)-value when transitioning from \( \lambda_C \) to \( \lambda_{ML} \). Instead, by staying uncommitted, the choice of the value is deferred to a later point (e.g., to the next external call from \( \lambda_{ML} \)). In contrast, locations that are subsequently accessed on the \( \lambda_{ML} \)-side have to remain enabled in this choice. They cannot be disabled, because an access to \( \checkmark \) would be undefined behavior.

The runtime provides the following primitives, a large subset of the OCaml FFI [oca 2023b], operating on the runtime state: The primitive alloc (for caml_alloc) allocates a new runtime block \( y \) by extending \( \zeta \) (with the block) and \( \chi \) (with \( * \)), "calls the GC" on \( \theta \), and then extends the new map with the \( \lambda_C \) address. Similarly, alloc_custom (for caml_alloc_custom) extends these maps with a new custom block. The primitives Field (for Field) and Store_field (for Store_field) access and update a field of a block in \( \zeta \). The primitives read_custom (for reading from Custom_contents) and write_custom (for writing to Custom_contents) access and update the contents of a custom block in \( \zeta \). The primitive isblock (for Is_block) checks whether a runtime value is a block, length (for Wosize_val) reads the length of the block, and read_tag (for Tag_val) reads the tag of the block. The primitives registerroot and unregisterroot can be used for registering and unregistering roots in the map \( rm \). These primitives model what the runtime does when the macros CAMLparam, CAMLlocal, and CAMLreturn are executed. The primitive callback can be used to trigger the execution of a callback \( \text{rec } f \ x \ . \ e \) in \( \lambda_{ML} \) (see also §4.1).

## 4 PROGRAM LOGIC

As we have seen in §2, reasoning about programs is done at the level of the language-local logics Iris\(\_C \) and Iris\(\_{ML} \) in Melocoton. However, eventually, we have to ensure that this reasoning is sound (i.e., that assumptions of one side are properly connected to proofs in the other). For this purpose, we introduce the "umbrella logic" Iris\(\_{ML+C} \), which embeds Iris\(\_C \) and Iris\(\_{ML} \). We discuss how Iris\(\_{ML+C} \) composes proofs (in §4.1), how it justifies the interface \( \Psi_{FFI} \) (in §4.2), and how it is modelled (in §4.3).

### 4.1 Composing Proofs in Iris\(\_{ML+C} \)

In Iris\(\_{ML+C} \), we primarily reason at the level of interfaces \( \Psi, \Pi \in \text{Intf}(Val) \). We write \( \Psi \models \Pi : \Pi \) to mean that the program \( \Pi \) implements interface \( \Pi \) under the "assumption" of interface \( \Psi \). We

\(^7\)The distinction between the roots map \( rm \) and set \( rs \) arises, because during the \( \lambda_{ML} \)-execution we know which runtime value is rooted and has to be materialized again when moving to \( \lambda_C \) whereas during the execution of \( \lambda_C \)-code the value that is stored in a root can be changed.
\[ \forall fn \circ P. \{ \Psi \circ fn \circ P \} \text{ call } fn \circ P @ p, \Pi \{ \psi'. P\psi' \} \quad \Psi \circ fn \text{ is False for all } fn \notin \text{dom } p \]

\[ \Pi \models p : \Psi \]

\[ \text{INTFIMPLEMENT} \]

\[ \forall fn \circ P. \{ \Psi \circ fn \circ P \} \text{ call } fn \circ P @ p, \Pi \{ \psi'. P\psi' \} \quad \Psi \circ fn \text{ is False for all } fn \notin \text{dom } p \]

\[ \Pi \models p : \Psi \]

\[ \text{INTFCONSEQ} \]

\[ \Psi \subseteq \Psi' \quad \Psi \models p : \Pi \quad \Pi' \subseteq \Pi \]

\[ \Psi' \models p : \Pi' \]

\[ \text{LINK} \]

\[ \Pi \models_{\text{ML}+C} p_1 : \Psi \quad \Psi \models_{\text{ML}+C} p_2 : \Pi \quad \text{dom}(p_1) \# \text{dom}(p_2) \]

\[ \emptyset \models_{\text{ML}+C} p_1 \oplus p_2 : \Psi \cup \Pi \]

\[ \text{EMBEDML} \]

\[ \{ \text{True} \} e @ \emptyset, \Pi \{ V. \exists n. V = n \ast \phi(n) \}_{\text{ML}} \quad \Pi \text{ is } \perp \text{ for primitives} \]

\[ \{ \Pi \}_{\text{FFI}} \models_{\text{ML}+C} \{ e \}_{\text{FFI}} : \Psi_{\text{FFI}} \cup \text{MAIN}(\phi) \]

\[ \begin{array}{l}
  \text{EMBEDC} \\
  \Psi \models_{\text{C}} p : \Pi \\
  \Psi \models_{\text{ML}+C} p : \Pi
\end{array} \]

Fig. 8. The interface rules of Iris_{\text{ML}+C}

We can prove (also in Iris_{\text{C}} and Iris_{\text{ML}}) that \( p \) implements an interface by showing that its functions satisfy the specification given by \( \Psi \) against the interface \( \Pi \) (INTFIMPLEMENT). As one would expect, there is also an analogue to the Hoare rule of consequence (INTFCONSEQ) where \( \Psi \subseteq \Psi' \Leftrightarrow \forall fn \circ P. \Psi \circ fn \circ P \rightarrow \Psi' \circ fn \circ P \). Using interfaces, we can succinctly state how proofs in the language-local logics (e.g., from §2) can be connected through Iris_{\text{ML}+C}:

**THEOREM 4.1 (CONNECTING Iris_{\text{C}} AND Iris_{\text{ML}}).** Let \( e \) be a \( \lambda_{\text{ML}} \)-expression, \( p \) a \( \lambda_{\text{C}} \)-program (where \( \text{dom } p \) does not contain FFI primitives), \( \Pi \) a \( \lambda_{\text{ML}} \)-interface, and \( \phi \) a pure predicate on integers. If

1. \( \{ \text{True} \} e @ \emptyset, \Pi \{ V. \exists n. V = n \ast \phi(n) \}_{\text{ML}} \text{ in Iris}_{\text{ML}} \) and
2. \( \Psi_{\text{FFI}} \models_{\text{C}} p : \{ \Pi \}_{\text{FFI}} \text{ in Iris}_{\text{C}}, \)

then we have \( \emptyset \models_{\text{ML}+C} \{ e \}_{\text{FFI}} \oplus p : \text{MAIN}(\phi) \text{ in Iris}_{\text{ML}+C}. \)

Let us break this theorem down. It allows linking a \( \lambda_{\text{ML}} \)-expression \( e \) with a \( \lambda_{\text{C}} \)-program \( p \). For \( e \), we need to prove a language-local triple (in Iris_{\text{ML}}) with a pure postcondition on integers \( \phi \) (e.g., True for a proof of safety). We can do so against an arbitrary \( \lambda_{\text{ML}} \)-interface \( \Pi \). For \( p \), we need to prove that its functions implement the interface \( \Pi \) as we have claimed they do. Since \( \Pi \) is an interface over \( \lambda_{\text{ML}} \)-values, but we want to use it in Iris_{\text{C}}, we convert it into an interface on \( \lambda_{\text{C}} \)-values using the interface wrapper [\cdot]|_{\text{FFI}} (discussed below). To use runtime primitives, \( p \) is proven against the runtime interface \( \Psi_{\text{FFI}} \). Here, \( \Psi_{\text{FFI}} \) gets the \( \lambda_{\text{ML}} \)-interface \( \Pi \) as an additional argument (which we omit in §2 for simplicity) to be able to execute \( \lambda_{\text{ML}} \)-callbacks from \( \lambda_{\text{C}} \) (see §4.2). Under these conditions, the theorem allows us to deduce that the combined program \( [e]_{\text{FFI}} \oplus p \) implements the interface \( \text{MAIN}(\phi) \triangleq \langle \text{atInit} \rangle \text{main } [ ] \langle w. \exists n. w = n \land \phi(n) \rangle \). Here, atInit is a resource that signals program start and main is a “primitive” of the wrapper that will trigger the execution of \( e \).

The proof of Theorem 4.1 is straightforward using the key properties of Iris_{\text{ML}+C} (in Fig. 8). We can embed proofs into Iris_{\text{ML}+C} from Iris_{\text{C}} (EMBEDC) and Iris_{\text{ML}} (EMBEDML). To embed an Iris_{\text{ML}} proof, it suffices to prove a Hoare triple about a \( \lambda_{\text{ML}} \)-expression \( e \) against the interface \( \Pi \). We then obtain that the wrapped program \( [e]_{\text{FFI}} \) implements the runtime interface \( \Psi_{\text{FFI}} \) and the main interface \( \text{MAIN}(\phi) \) —assuming the wrapped interface \( [\Pi]_{\text{FFI}} \). After embedding proofs into Iris_{\text{ML}+C}, we can then "link" them together (LINK): if one side assumes \( \Pi \) and implements \( \Psi \) and the other assumes \( \Psi \) and implements \( \Pi \), then they cancel out (i.e., the remaining assumption is \( \emptyset \)) and the linked program implements \( \Psi \cup \Pi \). In the case of Theorem 4.1, the wrapped program \( [e]_{\text{FFI}} \) assumes \( [\Pi]_{\text{FFI}} \) and implements \( \Psi_{\text{FFI}} \) whereas \( p \) assumes \( \Psi_{\text{FFI}} \) and implements \( [\Pi]_{\text{FFI}} \). Thus, they cancel out.
While its formal definition is quite a mouthful, its high-level intuition is comparatively simple: The
\([\text{the} \, (\ell \Rightarrow \gamma) \rightarrow \gamma \Rightarrow^\theta \gamma \, ]\) gives us access to the GC resource
\(\rightarrow \gamma \Rightarrow^\theta \gamma \Rightarrow \gamma \) to \(\gamma \). Since the resources \(a \mapsto \gamma \Rightarrow \gamma \) of \(\gamma \) is a root for the runtime block \(\gamma \), we obtain a new resource "\(a \Rightarrow \gamma \)" as a root with the runtime. If we do so, we obtain a new resource "\(a \Rightarrow \gamma \)" as a root with the runtime block \(\gamma \). The resource is stable across calls to the GC (i.e., it does not depend on \(\theta \)) and thus we can use it to access \(\gamma \) even after triggering the GC (e.g., through \(\text{alloc} \)). The rule \(\text{ExecCallback} \) allows executing a callback \(\text{exec} \, f \, \cdot \, \gamma \) in the runtime. It uses a special points-to assertion "\(\Rightarrow \gamma \)" for closures. The rule says that if we call \(f \) with the \(\lambda_C \)-representation \(w \) of \(V \) under precondition \(P \), then we get the \(\lambda_C \)-representation \(w' \) of the value \(V' \) that results from executing \(f \) and the postcondition \(Q(\gamma) \). The "\(\Rightarrow \gamma \)" modality in this rule is Iris’s modality for \(\text{step-indexing} \); we will discuss the effects of step-indexing on Iris\(_{ML+C} \) shortly (in §4.3).

**4.2 The Runtime Interface**

One thing that we have not discussed yet is who is on the other side of the runtime primitives in the interface Iris\(_{ML+C} \). Operationally, these primitives are given semantics by \(\cdots \Rightarrow \text{ML+C} \). At the program logic level, we prove their specifications in Iris\(_{ML+C} \) (against the underlying semantics \(\cdots \Rightarrow \text{ML+C} \)). A key selection of these primitives is depicted in Fig. 9. The rule \(\text{AllocCustom} \) allows allocating a custom block; it is analogous to the interface Iris\(_{alloc} \) (from §2.4). The rule \(\text{RegisterRoot} \) allows registering a \(\lambda_C \)-address \(a \) as a root with the runtime. If we do so, we obtain a new resource "\(a \Rightarrow \gamma \)" as a root points-to, which asserts that \(a \) is registered as a root for the runtime block \(\gamma \). The resource is stable across calls to the GC (i.e., it does not depend on \(\theta \)) and thus we can use it to access \(\gamma \) even after triggering the GC (e.g., through \(\text{alloc} \)). The rule \(\text{ExecCallback} \) allows executing a callback \(\text{exec} \, f \, \cdot \, \gamma \) in the runtime. It uses a special points-to assertion "\(\Rightarrow \gamma \)" for closures. The rule says that if we call \(f \) with the \(\lambda_C \)-representation \(w \) of \(V \) under precondition \(P \), then we get the \(\lambda_C \)-representation \(w' \) of the value \(V' \) that results from executing \(f \) and the postcondition \(Q(\gamma) \). The "\(\Rightarrow \gamma \)" modality in this rule is Iris’s modality for \(\text{step-indexing} \); we will discuss the effects of step-indexing on Iris\(_{ML+C} \) shortly (in §4.3).

**View Reconciliation.** Proving these rules is reasonably straightforward against the underlying, combined semantics \(\cdots \Rightarrow \text{ML+C} \). The main challenge, also at the level of the program logic, is view reconciliation. Concretely, we have to soundly connect the possibly overlapping resources \(a \Rightarrow c \) of Iris\(_{C} \), \(\ell \Rightarrow \text{ML} \) of Iris\(_{ML} \), and \(\gamma \Rightarrow \text{blk}[r|m] \) of the runtime. We focus on \(\ell \Rightarrow \text{ML} \) and \(\gamma \Rightarrow \text{blk}[r|m] \) of the runtime, since the resources \(a \Rightarrow c \) and \(\gamma \Rightarrow \text{blk}[r|m] \) never actually overlap by the design of the semantics \(\cdots \Rightarrow \text{ML+C} \) (but there are some intricacies surrounding garbage collection and roots). The resources \(\ell \Rightarrow \text{ML} \) and \(\gamma \Rightarrow \text{blk}[r|m] \) very much overlap: if we mutate a runtime block in \(\lambda_C \) through the FFI, then this change will be observable in \(\lambda_C \) after an external call.

We solve this problem using the central idea underlying separation logic: \textit{mutually exclusive ownership}. Concretely, we maintain as an invariant (not in the Iris sense) that, conceptually, the runtime blocks backing $\ell \mapsto_{\text{ML}} V$ and $y \mapsto_{\text{blk}[t \mid ml]} \delta$ must always be disjoint. We maintain this invariant in the model of the GC resource:

\[
GC(\emptyset) \triangleq \exists \xi_{\text{phys}}, \sigma_{\text{virt}}, \xi_{ml}. \SI_{\text{ML}}(\sigma_{\text{virt}}) \ast \left[ \left[ \delta(\xi_{\text{phys}}) \right] \ast \xi_{\text{phys}} = \xi_{\text{virt}} \cup \xi_{ml} \ast \text{repr}(\sigma_{\text{virt}}, \xi_{ml}) \ast \cdots \right].
\]

The GC resource is available while we are verifying glue code. The physical view that the operational semantics has at this point on the $\lambda_{\text{ML}}$-state is the runtime heap $\xi_{\text{phys}}$ (the map $\xi$ in execution mode $C$ in Fig. 7). Logically, we split this heap into two disjoint parts, $\xi_{\text{virt}}$ and $\xi_{ml}$. The part $\xi_{\text{virt}}$ backs up the resource $y \mapsto_{\text{blk}[t \mid ml]} \delta$ (through $\delta(\xi_{\text{virt}})$) and the part $\xi_{ml}$ backs up the resource $\ell \mapsto_{\text{ML}} V$.

Of course, we cannot directly back $\ell \mapsto_{\text{ML}} V$ with a runtime heap, since “$\mapsto_{\text{ML}}$” is a resource from Iris which—as usual—is backed by a $\lambda_{\text{ML}}$-heap (through $\SI_{\text{ML}}(\cdot)$, the “state interpretation” in the terminology of Iris). However, what we can do is maintain a separate, virtual $\lambda_{\text{ML}}$-heap $\sigma_{\text{virt}}$ (not present in the underlying physical memory) which (1) backs the resource $\mapsto_{\text{ML}}$ in the form of $\SI_{\text{ML}}(\sigma_{\text{virt}})$ and (2) is faithfully represented by the runtime heap $\xi_{\text{virt}}$ (denoted $\text{repr}(\sigma_{\text{virt}}, \xi_{ml})$).

When we are verifying glue code in Iris$_{C}$, the way the double vision rules (see Fig. 6) work is by moving locations between $\sigma_{\text{virt}}$ and $\xi_{\text{virt}}$ as needed. Concretely, ML-TO-FFI removes $\ell$ from $\sigma_{\text{virt}}$ and adds a corresponding runtime identifier $y$ (previously stored in $\xi_{ml}$) into $\xi_{\text{virt}}$. FFI-TO-ML does the opposite. When we start verifying a glue code function, the original $\lambda_{\text{ML}}$-heap is turned into $\sigma_{\text{virt}}$, and then we can gradually convert to and from the runtime representation as needed.

### 4.3 The Model of Iris$_{ML+}$

We have discussed how to verify programs $[e]_{\text{FFI}} \oplus p$ in Iris$_{ML+}$ (see Theorem 4.1). What is still missing is what we obtain from verifying a program in Iris$_{ML+}$. The answer is 

\begin{theorem}[Adequacy of Iris$_{ML+}$]
Let $p$ be a $\lambda_{ML+}$-program and $\phi$ a pure predicate on integers. If $\emptyset \models p : \text{MAIN}(\phi)$, then $p; (\text{call main } [], \emptyset) \rightarrow^*_{\text{ML+}} \{(\bar{n}, h) \mid \phi(h)\}$.
\end{theorem}

Intuitively, this theorem says that call main $[]$ (which will trigger the execution of $e$) can only terminate in integers satisfying $\phi$ and diverge. In particular, call main $[]$ is safe to execute. Formally, we define the program executions on multi-relations coinductively as

\[
p; s \rightarrow^* S \triangleq_{\text{coind}} s \in S \lor \exists S'. \ p; s \rightarrow S' \land \forall s' \in S'. \ p; s' \rightarrow^* S
\]

At first glance, this definition may seem odd, because it looks like we are proving the existence of an execution. However, this thought is misleading. In a semantics with demonic and angelic non-determinism, if we want to prove something for all possible executions of a program, we have to resolve angelic choices and accept demonic choices. In our multi-relations $p; s \rightarrow S$, the outer choice over the set of states $S$ is angelic and, thus, when we reason about all executions, we need to resolve this choice (hence “$\exists S’$”). The inner choice over the next state $s \in S$ is demonic and, hence, when we reason about all executions, we need to accept this choice (hence “$\forall s' \in S'$”).

**Angelic non-determinism and step-indexing.** The proof of Theorem 4.2 is conceptually not too far from a standard Iris adequacy proof. There is, however, one aspect that warrants discussion: step-indexing and its interactions with angelic non-determinism. The way that step-indexing shows up has to do with the model of Hoare triples in Iris$_{ML+}$, so let us discuss this model first.

As usual for Iris, we define Hoare triples in terms of a \textit{weakest precondition}, which in our case takes in the surrounding program $p$ and interface $\Psi$. A simplified version is depicted in Fig. 10. There are three things to note about this definition. First, it has three (instead of two) cases, since it contains an additional case for external calls (WP2). This case allows us to “skip” over external
We have applied Melocoton to several interesting case studies. We start with two examples that

\[ WP1 \] \[ \text{wp} (e, v) \in p, \Psi \{ \Phi \} \triangleq \forall \sigma. \text{SI}(\sigma) \Rightarrow \text{wp'} (e, \sigma) \in p, \Psi \{ \Phi \} \]

\[ WP2 \] \[ \text{wp'} (K[\text{fn } \overline{\Phi} \text{ at } v], \sigma) \in p, \Psi \{ \Phi \} \triangleq \text{SI}(\sigma) * \Phi(v) \]

\[ WP3 \] \[ \exists S. \: \text{wp'} (e, \sigma) \in p, \Psi \{ \Phi \} \triangleq (e \text{ not a value nor a call}) \]

\[ \exists \forall S. \: \text{wp'} (e, \sigma) \in p, \Psi \{ \Phi \} \triangleq (e \text{ not a value nor a call}) \]

Fig. 10. Simplified weakest-precondition wp e @ p, \{ \Phi \} of Iris_{\text{ML+C}}

function calls in the way explained in §2.2. Second, the definition is step-indexed—as usual in
Iris—which reveals itself through the modality “⇒” that appears in the cases (WP2) and (WP3).

Step-indexing enables all kinds of recursive reasoning in Iris and, hence, Iris_{\text{ML+C}}. We make use
of it in our examples (in §5) and when reasoning about callbacks (see ExecCallback). Third, the
weakest precondition uses multi-relations \( p; z \rightarrow \rightarrow s \) instead of “ordinary” relations \( p; s \rightarrow \rightarrow s' \)
in (WP3). This is only the case for Iris_{\text{ML+C}}; the logics Iris_{\text{C}} and Iris_{\text{ML}} use weakest preconditions with
ordinary relations as usual. We use multi-relations here to deal with the semantics \( \rightarrow \rightarrow \).

The interaction of step-indexing and angelic non-determinism is interesting: To prove Theorem 4.2, we have to take the angelic choices over \( S \) in (WP3) inside of Iris_{\text{ML+C}} and replay them “at
the meta level” to resolve the angelic choices in \( p; s \rightarrow \rightarrow s \). (Note that the order of the choices in
the two definitions is exactly the same: angelic, then demonic, then repeat.) However, in traditional
step-indexed logics such as Iris, the meaning of existential quantification is weaker than what one
would intuitively expect [Spies et al. 2021]. In particular, it is not always the case that an existential
quantifier in the logic such as “\( \exists S \)” in (WP3) implies an existential quantifier outside the logic such
as “\( \exists S' \)” in \( p; s \rightarrow \rightarrow s \). To “replay” the angelic choices outside of Iris_{\text{ML+C}}, we have to use a stronger
form of step-indexing than the one that Iris provides out-of-the-box. To do so, we build Melocoton
atop of Transfinite Iris [Spies et al. 2021], a transfinitely step-indexed version of Iris that allows us
to do the kind of “witness extraction” from existential quantifiers that we need in this proof.

5 CASE STUDIES

We have applied Melocoton to several interesting case studies. We start with two examples that
illustrate how Melocoton can be used for program verification. Afterwards, in §5.1, we show how
Melocoton can, additionally, be used to prove type safety of external functions implemented in C.
invalidation). Thus, OCaml code that uses the buffer library must carefully respect its specification. The function \texttt{is\_compressible}, as we prove by verifying it, satisfies these requirements.

**Polymorphic equality.** As another example, we have verified a polymorphic equality function that (deeply) compares the runtime representation of two OCaml values. What makes this function interesting is that it is not possible to implement it natively in OCaml (or in our case \(\lambda_{\text{ML}}\)), because there is no way to determine the \textit{shape of values} (e.g., whether they are sums or pairs). For example, a \(\lambda_{\text{ML}}\) function designed to compare pairs would get stuck when we attempt to pass it a sum value.

As such, this example demonstrates that the FFI can be used to add new functionality to \(\lambda_{\text{ML}}\).

### 5.1 Type Safe Interfaces

Besides program verification, we can additionally use Melocoton to prove type safety of external functions (and their clients). To this end, we equip \(\lambda_{\text{ML}}\) with a \textit{logical relation}. The logical relation (and its associated type system) are standard constructions from the literature [Timany et al. 2022, Figures 2 and 5], which we extend with support for external functions implemented in \(\lambda_C\): First, we add a new context \(\Sigma\), which assigns a type \(\text{fn} : A_1 \to \cdots \to A_n \to B\) to every external function \(\text{fn}\) in it. Then, we interpret the contexts using the interfaces \(\Psi\) of Iris\(_{\text{ML}}\) such that each external function \(\text{fn} \in \Sigma\) is assigned (the semantic interpretation of) its type as a specification in \(\Psi\). Finally, we use Iris\(_C\) to \textit{validate} the assumed types of external functions, which means we prove in Iris\(_C\) that the \(\lambda_C\)-implementation satisfies the interpretation of the given type. Besides proving type safety of individual external functions \(\text{fn}\), we can additionally use the logical relation to prove type safety of OCaml clients that wrap them in a safe abstraction (e.g., \texttt{is\_compressible}).

**Landin’s knot.** Our next example, a simple modification of Landin’s knot [Landin 1964], illustrates that Melocoton and its logical relation are powerful enough to reason about higher-order functions, callbacks to \(\lambda_{\text{ML}}\), and mutual recursion \textit{through the FFI} (and the heap). Our version of Landin’s knot implements recursion through backpatching by combining \(\lambda_{\text{ML}}\) and \(\lambda_C\) code:

```ml
let knot (f : ('a -> 'b) -> ('a -> 'b)) = value callk(value l, value x) {
  let l = ref (fun _ -> assert false) in
  1 := (fun x -> f (fun y -> callk l y) x); return caml_callback(f, x);
  (fun x -> callk l x)
}
```

We proved functional correctness of \texttt{knot} (i.e., it is a recursion combinator), and that it is semantically safe (i.e., in the logical relation) at type \(\forall\alpha\beta. ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)\).

**Event listeners.** As our last example, we have verified a small library of “event listeners”. It demonstrates a tricky use of higher-order state in \(\lambda_C\) (i.e., memory storing closures), which we handle using step-indexing. The \(\lambda_{\text{ML}}\)-interface for this library is below. It captures a programming pattern commonly found in event-based GUI libraries to mediate between “event consumers” (clients of the GUI library) and “event producers” (the backend of the GUI). Consumers are handed an abstract value (of type ‘a listener) to which they can attach callbacks (using \texttt{listen}) to react to future events. When an event happens, the backend can notify (using \texttt{notify}) a listener which triggers the consumer’s callback. The library is implemented in \(\lambda_C\) and exposed using the FFI.

```ml
type 'a listener
external create : unit -> 'a listener = "listener\_create"
external listen : ('a -> unit) -> 'a listener -> unit = "listener\_listen"
external notify : 'a -> 'a listener -> unit = "listener\_notify"
```

The most interesting thing about this library is that the implementation of \texttt{listen} stores an arbitrary \(\lambda_{\text{ML}}\)-callback in a mutable data structure managed on the \(\lambda_C\)-side. The callback, by the nature of event propagation, is incapable of being in a structural state without a \texttt{listen} event. Therefore, \(\lambda_{\text{ML}}\) functions must be executed synchronously on the OCaml side, meaning we can safely use the buffer library in our OCaml clients.

of OCaml’s type system, can capture the listener to which it is attached. This makes proving type safety of the library tricky, because we can easily run into circularity issues when naively attempting to define an interpretation of a listener (e.g., a listener is safe if its callbacks are safe, and callbacks are safe if the listeners they capture are safe). We resolve these circularity issues as usual in these relations, by using step-indexing. Concretely, analogous to our standard interpretation of reference types [Timany et al. 2022, Figures 5], we make use of Iris’s impredicative invariants [Jung et al. 2018, §7.1], which internally are modelled using step-indexing. As mentioned in §4.3, for the semantics \( \rightarrow_{\text{ML+C}} \), this is only sound because we use a transfinitely step-indexed version of Iris. Ultimately, we prove the functions \( (\text{create}, \text{listen}, \text{notify}) \) safe at type:

\[
\forall \alpha. \exists \text{listener}. (\text{unit} \rightarrow \text{listener}) \times ((\alpha \rightarrow \text{unit}) \rightarrow \text{listener} \rightarrow \text{unit}) \times (\alpha \rightarrow \text{listener} \rightarrow \text{unit}).
\]

6 RELATED WORK

Melocoton is, to our knowledge, the first program logic for programs spanning multiple languages with different memory models. (The only other multi-language program logic that we are aware of is Iris-Wasm discussed below.) Here, we also compare with work that tackles the related problem of compiler verification in a multi-language setting (but is not based on a program logic).

**Iris-Wasm.** Iris-Wasm [Rao et al. 2023] provides an Iris-based program logic for reasoning about WebAssembly and its interaction with its host language, which in Iris-Wasm is a tiny subset of JavaScript. The memory model of the host is a minor extension of WebAssembly’s memory model and hence, there is no view reconciliation challenge. In contrast, Melocoton shows how to scale verification to more complex FFIs that require integrating program logics with very different memory models and how to deal with the problems that arise such as view reconciliation (§2.5).

**Cito.** Cito [Pit-Claudel et al. 2020; Wang et al. 2014] is a C-like language with a verified compiler formalized in Coq, which supports linking with functions from other languages via axiomatic specifications built into its operational semantics. While these axiomatic specifications follow the style of open simulations [Hur et al. 2012] like the interfaces \( \Psi \) presented in this paper, they are not phrased using a program logic, but stated using abstract data types and pure pre- and postconditions. Cito uses program logics to reason about individual languages, but it does not tackle the problem of building a multi-language program logic like Iris\(_{\text{ML+C}}\).

**Cogent.** Cheung et al. [2022] show how to extend the compiler correctness theorem of Cogent [O’Connor et al. 2016, 2021] with manually verified external C functions. Cheung et al. require the C code to uphold the invariants guaranteed by Cogent’s linear type system and focus on extending the correctness proof of the Cogent compiler using these invariants. In contrast, we assume correctness of the OCaml compiler and runtime and instead focus on building a program logic for verifying OCaml-and-C programs that maintain the (more complex) OCaml runtime invariants.

**Semantic soundness for language interoperability.** Patterson et al. [2022] prove type safety of the interaction between (among others) a MiniML-style garbage collected language and an L3-style language with manual memory management. Instead of Melocoton’s source-level reasoning, they compile both languages to a common target language and build logical relations that relate source-level types with target-level terms. This approach shifts reasoning to the target language. For example, they relate values from different languages using target-level conversion functions instead of Melocoton’s source-level relations like \( V \sim_{\text{ML}} v \) and \( v \sim_{\text{C}} w \).

**Verified compilers.** Mates et al. [2019] (building on the work of Patterson et al. [2017]) verify a compiler from a stateful language with closures to one without closures in a syntactic multi-language [Matthews and Findler 2007; Perconti and Ahmed 2014] using a logical relation for
contextual equivalence. Hur and Dreyer [2011] verify a one-pass compiler from an ML-like language to an assembly-like language, via a cross-language logical relation where garbage collection is axiomatized. Their approach supports linking with manually verified assembly-level code, but only if that code can be proven behaviorally equivalent to some ML-level module. Compositional CompCert [Stewart et al. 2015] and the line of work it inspired [Gu et al. 2015; Koenig and Shao 2021; Song et al. 2020] show how to extend the CompCert compiler with cross-language linking. The composition of languages based on external calls in Melocoton is inspired by Compositional CompCert’s interaction semantics, but extends it with DimSum-style wrappers [Sammler et al. 2023] to handle linking of languages with different memory models. DimSum [Sammler et al. 2023] provides a “decentralized” approach for reasoning about multi-language programs via combinators for linking and language translation, which inspired the definition of the operational semantics in Melocoton (see §3). DimSum only considers the interaction of relatively low-level languages, none of which have such a rich FFI and runtime as OCaml.

Formal reasoning about the OCaml FFI. The model of the OCaml FFI in this paper is based on the informal description given in the OCaml manual [oca 2023b]. We model the core features of the FFI, but omit some more advanced features like direct access in C to runtime memory, exceptions, and multithreading. Furthermore, we model the primitives for registering roots with a more elementary API than the one provided by the OCaml FFI through CAMLlocal, CAMLparam, and CAMLreturn. Alternative APIs for rooting have also been proposed by Munch-Maccagnoni and Scherer [2022], and we believe that these APIs would be easy to specify on top of our primitives. Furr and Foster [2005] build a type system for C glue code that uses the OCaml FFI to detect common misuses of the FFI. This type system is proven sound using a formal model of the C-side of the FFI. However, because their focus is on finding bugs in C glue code, they do not model the OCaml side of the FFI and do not target verification of mixed OCaml and C code like Melocoton. This also means that they do not handle advanced features such as callbacks and closures.

REFERENCES

1188 Guillaume Munch-Maccagnoni and Gabriel Scherer. 2022. Boxroot, fast movable GC roots for a better FFI. In ML Family Workshop. Benoît Montagu, Ljubljana, Slovenia. https://hal.inria.fr/hal-03910313


