

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.

Authors’ addresses: Armaël Guéneau, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France, armael.gueneau@inria.fr; Johannes Hostert, Saarland University and MPI-SWS, Saarland Informatics Campus, Germany, jhostert@mpi-sws.org; Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

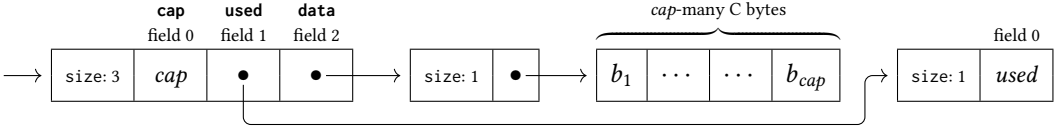


Fig. 1. C memory layout of the `buf`-record `{cap, used, data}` as exposed by the OCaml FFI

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),¹ 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:

```
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

¹OCaml’s values are abstract in the sense that their semantics is independent of their machine-level representation. In contrast, C’s values are concrete in the sense that they have concrete representation as a sequence of bytes.

99 stores the underlying bytes, the `cap`-field the capacity of the buffer, and the `used`-field how much of
 100 the buffer is currently in use (*i.e.*, how many bytes are readable). The type `bytes`—as far as OCaml
 101 is concerned—is abstract (*i.e.*, declared with `type bytes` without a definition), meaning it does not
 102 reveal anything about its contents to OCaml.

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

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

122 In *program code*, the differences between OCaml and C are bridged through so-called “glue code”,
 123 code which uses the OCaml FFI to link up program parts written solely in either OCaml or C. For
 124 *program verification*, the story is not so clear yet. For instance, when we reason about glue code,
 125 how do we reconcile the two very different views that OCaml and C have on values of type `buf`?
 126 To design a multi-language program verification analogue of the single-language recipe, we have
 127 to bridge the language gap at two different levels:

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

140 **The program logic.** At the program logic level, the main challenge is preserving *language-local*
 141 *reasoning*. That is, chunks of code that are solely written in C or solely written in OCaml should
 142 enjoy reasoning principles that are fine-tuned to their respective language (without being impacted
 143 by the existence of the other language). However, preserving language-local reasoning is easier
 144 said than done. For one, the two program logics will have different views *on the same piece of state*
 145 (*e.g.*, a value in OCaml will refer to nested runtime blocks on the C side). Changes to runtime blocks
 146 on the C side (*e.g.*, updating the block storing the number of *used* bytes) correspond to changes
 147

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 λ_C and λ_{ML} (see §2.2) and—following the single-language recipe—we define their canonical operational semantics, \rightarrow_C and \rightarrow_{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 **Iris_C** and **Iris_{ML}**, in the separation logic framework Iris [Jung et al. 2018, 2015].
- (2) We extend the language-local program logics **Iris_C** and **Iris_{ML}** 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 **Iris_{ML}**) 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 \rightarrow_C and \rightarrow_{ML}).
- (3) Finally, we develop an operational semantics for the combined language, \rightarrow_{ML+C} , which embeds the operational semantics of λ_C and λ_{ML} and adds reasoning rules to bridge between them. Moreover, we develop a separation logic, **Iris_{ML+C}**, which embeds **Iris_C** and **Iris_{ML}** and extends their reasoning principles to bridge between the two logics. We then prove that **Iris_{ML+C}** is sound with respect to the joint semantics \rightarrow_{ML+C} .

In Melocoton, the verification of a mixed OCaml and C program can be done *almost entirely* in the language-local logics **Iris_C** and **Iris_{ML}**. 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 **Iris_{ML}** specification). In fact, even code interacting with the OCaml FFI—on the C side—can stay in the language-local logic **Iris_C** (see §2.4). The only purpose of the “umbrella logic” **Iris_{ML+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 λ_{ML} and λ_C , 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

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—inherited from Iris—to handle the recursive and higher-order features of OCaml. To soundly combine (1) angelic non-determinism 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 $\mathbf{Iris}_{\text{ML+C}}$ 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 $\mathbf{Iris}_{\text{ML+C}}$, 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 `snappy_compress` can be used to compress an input buffer `inp` (i.e., a raw C pointer to a chunk of bytes) of length `insz` into an output buffer `outp` of capacity `outsz`. After compression, the pointer `outsz` stores the actual length of the compressed data. To make sure that the capacity of the output buffer `outp` suffices, the library provides the function `snappy_max_compressed_length`, which computes an upper bound on the compressed size.

The OCaml client. The OCaml client `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 `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 `snappy_compress` expects two char pointers (i.e., raw pointers to C buffers of bytes). The issue is that the two types, `char array` and `char *`, are not the same—they are not even part of the same language.

```

246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294

```

C library:

```

1 int snappy_compress(unsigned char *inp, int insz, unsigned char *outp, int *outsz);
2 size_t snappy_max_compressed_length(size_t source_length);

```

OCaml client:

```

3 type bytes and buf = { cap : int; used: int ref; data: bytes }
4 external buf_alloc   : int -> buf           = "buf_alloc"
5 external buf_free    : buf -> unit          = "buf_free"
6 external buf_get     : buf -> int -> char   = "buf_get"
7 external buf_upd     : int -> int -> (int -> char) -> buf -> unit = "buf_upd"
8 external wrap_compress : buf -> buf -> bool = "wrap_compress"
9 external wrap_max_len  : int -> int         = "wrap_max_len"
10 let is_compressible (xs: char array) =
11   let len = Array.length xs in if len = 0 then false else
12   let (inp, outp) = (buf_alloc len, buf_alloc (wrap_max_len len)) in
13   buf_upd 0 (len - 1) (fun i -> Array.get xs i) inp;
14   let _ = wrap_compress inp outp in let shrank = !(outp.used) < !(inp.used) in
15   buf_free inp; buf_free outp; shrank

```

Fig. 2. Using a C compression function from OCaml.

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_upd` 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 λ_C

295	λ_C	
296		$w \in Val ::= (n \in \mathbb{Z}) \mid (a \in Addr)$
297		$c \in Expr ::= w \mid x \mid \ominus c \mid \otimes c \mid \text{malloc}(c) \mid \text{free}(c, c) \mid *c \mid *c \leftarrow c \mid \text{while}(c) c \mid \text{call } fn \vec{c} \mid \dots$
298		$p \in Prog ::= \emptyset \mid p, fn(\vec{x}) := c$
299		$\sigma \in State \triangleq Addr \xrightarrow{\text{fin}} Cell$
300		$where\ cl \in Cell \triangleq \{\star, \dagger\} \uplus Val$
301	λ_{ML}	
302		$V \in Val ::= (n \in \mathbb{Z}) \mid (\ell \in Loc) \mid \text{true} \mid \text{false} \mid \langle \rangle \mid \langle V, V \rangle \mid \text{inl } V \mid \text{inr } V \mid \text{rec } f x. e \mid \textcircled{1}$
303		$e \in Expr ::= V \mid x \mid e e \mid \ominus e \mid e \otimes e \mid \text{alloc } e e \mid e.(e) \mid e.(e) \leftarrow e \mid \text{length } e \mid \text{call } fn \vec{e} \mid \dots$
304		$\sigma \in State \triangleq Loc \xrightarrow{\text{fin}} \text{list}(Val) \uplus \{\dagger\}$
305		$p \in Prog ::= \emptyset$

Fig. 3. Syntax, state, and resources of λ_C and λ_{ML}

and λ_{ML} , which are depicted in Fig. 3. We first discuss the language λ_C and verify the compression library in **Iris_C**. Then, we contrast λ_C with λ_{ML} and verify the client in **Iris_{ML}**.

The language λ_C . The essential features of λ_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 λ_C is a finite map from addresses to *memory cells*, either values or special tokens indicating that an address has been freshly allocated (\star) or has been freed already (\dagger). 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 λ_C , are lists of functions (where no function is defined twice).

Verifying the C library. The language λ_C gives rise to **Iris_C**, a simple, language-specific separation logic, depicted in Fig. 4. In the context of our running example, we use **Iris_C** to prove correctness of the two compression library functions. Since **Iris_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_C**, one proves Hoare triples of the form “ $\{P\} \text{call } fn \vec{w} @ p, \Psi \{Q\}_C$ ” where **call** $fn \vec{w}$ is the function call construct of λ_C , p is the surrounding program, P and Q are the pre- and postconditions, and Ψ 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

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

where p_{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 “maxlen”).²

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* $a \mapsto_c cl$ of **Iris_C**. It conveys *ownership* over the memory address a and asserts that it currently

²In 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.

344	Iris_C	
345	READ-C	WRITE-C
346	$\{a \mapsto_C w\} * a \{w'. w = w' * a \mapsto_C w\}_C$	$\{a \mapsto_C _ \} * a \leftarrow w' \{ _ . a \mapsto_C w' \}_C$
347	Iris_{ML}	
348	ALLOC-C	FREE-C
349	$\{n \geq 0\} \text{malloc}(n) \{a. *_{0 \leq i < n} (a + i) \mapsto_C \star\}_C$	$\{*_{0 \leq i < n} (a + i) \mapsto_C _ \} \text{free}(a, n) \{ _ . \text{True} \}_C$
350	Iris_{ML}	
351	READ-ML	WRITE-ML
352	$\{\ell \mapsto_{ML} \vec{V}\} \ell.(i) \{V'. V' = \vec{V}[i] * \ell \mapsto_{ML} \vec{V}\}_{ML}$	$\{\ell \mapsto_{ML} \vec{V}\} \ell.(i) \leftarrow V' \{ _ . \ell \mapsto_{ML} \vec{V}[i := V'] \}_{ML}$
353	ALLOC-ML	
354	$\{n \geq 0\} \text{alloc } n V \{\ell. \exists \vec{V}. \ell \mapsto_{ML} \vec{V} * \vec{V} = n * \forall i. \vec{V}[i] = V\}_{ML}$	
355	Iris_C and Iris_{ML}	
356	CALL-INTERNAL	CALL-EXTERNAL
357	$\frac{p(fn) = f \quad \{P\} f(\vec{v}) @ p, \Psi \{v. Q v\}}{\{P\} \text{call } fn \vec{v} @ p, \Psi \{v. Q v\}}$	$\frac{fn \notin \text{dom}(p) \quad P \multimap \Psi fn \vec{v} Q}{\{P\} \text{call } fn \vec{v} @ p, \Psi \{v. Q v\}}$
358		
359		
360		

Fig. 4. A selection of the reasoning rules of **Iris_C** and **Iris_{ML}**. The program p and the interface Ψ are omitted in rules that do not mention them.

stores the memory cell cl . For **snappy_compress**, we show

$$\begin{aligned} & \{ \text{bf}(a_{in}, n_{in}, \vec{m}_{in}) * \text{bf}(a_{out}, n_{max}, []) * a_{outsz} \mapsto_C n_{max} * \text{maxlen}(n_{in}, n_{max}) \} \\ & \text{call snappy_compress } [a_{in}, n_{in}, a_{out}, a_{outsz}] @ p_{lib}, \emptyset \\ & \{ w'. w' = 0 * \exists \vec{m}_{out}, n_{out}. \text{bf}(a_{in}, n_{in}, \vec{m}_{in}) * \text{bf}(a_{out}, n_{max}, \vec{m}_{out}) * \text{cpr}(\vec{m}_{in}, \vec{m}_{out}) * a_{outsz} \mapsto_C |\vec{m}_{out}| \}_C \end{aligned}$$

where $\text{bf}(a, n, [m_0, \dots, m_{k-1}]) \triangleq (*_{0 \leq i < k} (a + i) \mapsto_C m_i) * (*_{k \leq i < n} (a + i) \mapsto_C _) * k \leq n$ asserts that a stores a buffer of capacity n with the first k cells storing the integers m_0, \dots, m_{k-1} , and we write $a \mapsto_C _ \triangleq \exists cl \in \text{Val} \uplus \{\star\}. a \mapsto cl$ 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 $\text{cpr}(\vec{m}_{in}, \vec{m}_{out})$), and a_{outsz} will store the length of the compressed buffer.

The language λ_{ML} . Let us now turn to λ_{ML} . In contrast to λ_C , the language has a very rich notion of values: integers, locations, booleans, unit, pairs, sums, foreign values “ \textcircled{i} ” (discussed shortly), and *closures*. Moreover, the memory model of λ_{ML} is very different. The heap of λ_{ML} is a map from (abstract) locations to lists of values. Superficially, this may seem similar to the memory of λ_C . However, there are several crucial differences: First, memory management in λ_{ML} relies on garbage collection, meaning memory is allocated on the heap with **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 λ_{ML} is more complex, because it can store (lists of) arbitrary values, regardless of their shape or size (e.g., $\langle \langle n, m \rangle, \text{rec } f x. e \rangle$, a pair of another pair and a closure). Third, locations in λ_{ML} store *entire lists of values*. Each location ℓ models a mutable array, whose length can be determined with **length** ℓ , whose elements can be retrieved with $\ell.(n)$, and whose elements can be updated with $\ell.(n) \leftarrow V$. In contrast to λ_C , there is no “address arithmetic” (i.e., the expression $\ell + i$ is stuck). A third, minor difference to λ_C is that there are no top-level function declarations in λ_{ML} (i.e., $p = \emptyset$) and, instead, in λ_{ML} we execute single expressions e (which internally can contain let-bindings and mutual recursion).

393 There are three aspects of λ_{ML} that enable language interoperability. They are non-intrusive and
 394 minimal such that λ_{ML} does not even (need to) know which languages it is interacting with. First,
 395 there are foreign values “ \textcircled{i} ”, where i is an abstract identifier. They can be used by other languages
 396 such as λ_{C} to “embed” their own data into λ_{ML} . We will use this feature of OCaml to embed the
 397 underlying C buffer into OCaml in our running example (in §2.5). Foreign values are abstract:
 398 they can be passed around via function calls, but there is no language construct in λ_{ML} to inspect
 399 their contents. Second, after executing an external function, locations in the heap of λ_{ML} can store
 400 data that is temporarily inaccessible from λ_{ML} (see §3.1). Whenever this is the case, the λ_{ML} heap
 401 stores \textcircled{f} at the respective location and accesses in λ_{ML} will get stuck. Third, and most interestingly,
 402 λ_{ML} contains a language construct for *external function calls* `call fn \vec{V}` . Importantly, the operational
 403 semantics *does not assign any meaning* to it (i.e., `call fn \vec{V}` is stuck). By choosing to be stuck, λ_{ML}
 404 does not have to include any rules in its semantics for executing code of other languages (e.g., λ_{C}).
 405

406 **Verifying the OCaml client.** Let us turn to the verification of `is_compressible` in `IrisML`. We
 407 prove that given ownership over an array of integers, the function returns an unspecified boolean,
 408 meaning $\{\ell \mapsto_{\text{ML}} \vec{n}\} \text{is_compressible } \ell @ \emptyset, \Psi_{\text{buf}}\{V'. V' \in \{\text{true}, \text{false}\} * \ell \mapsto_{\text{ML}} \vec{n}\}_{\text{ML}}$. This spec-
 409 ification is not particularly exciting, but it suffices to illustrate the interaction with the external
 410 functions implemented in C. The resources of `IrisML` (e.g., “ $\ell \mapsto_{\text{ML}} \vec{V}$ ”) and its reasoning rules
 411 (e.g., `READ-ML`, `WRITE-ML`, and `ALLOC-ML`) are standard. Thus, we focus mainly on the treatment
 412 of *external function calls*. External calls allow us to slice the verification of a program into the
 413 language-local parts and the parts that are implemented in another language. In `is_compressible`,
 414 the first time that we call an external function is when we allocate the initial buffers `inp` and `outp`
 415 with `buf_alloc` (Line 12). Suppose (for a moment), for the sake of explanation, that `buf_alloc`
 416 was implemented *in OCaml*. In this hypothetical scenario, the standard way to proceed would be to
 417 prove (once) and afterwards apply (multiple times) the Hoare triple:
 418

$$419 \{n > 0\} \text{buf_alloc}(n) @ \emptyset, \Psi_{\text{buf}}\{V. \text{buffer}_{\text{ML}}(V, n, [])\}_{\text{ML}}$$

420 where $\text{buffer}_{\text{ML}}(V, n, \vec{m}) \triangleq \exists \ell, V'. V = \langle n, \ell, V' \rangle * \ell \mapsto_{\text{ML}} [|\vec{m}|] * \text{bytes}(V', n, \vec{m})$ asserts that V is
 421 a tuple containing the buffer capacity n , a λ_{ML} reference ℓ for the used field, and a value V'
 422 containing the underlying integers \vec{m} . How V' stores the integers \vec{m} is irrelevant for the verification
 423 of `is_compressible`, meaning we can stay at the abstraction of “`bytes(V', n, \vec{m})`”. (As we will see
 424 in §2.5, the definition of `bytes` uses λ_{ML} ’s “foreign values” and knowledge about the OCaml FFI.)
 425 After applying the Hoare triple, we can then use the buffer that `buf_alloc` returns to resume the
 426 verification of `is_compressible`.

427 Of course, `buf_alloc` is *not actually* implemented in OCaml. Nevertheless, in our proof of
 428 `is_compressible`, we want to stay as close as possible to the language-local reasoning sketched
 429 out above. To do so, we introduce reasoning principles to “skip” external function calls by drawing
 430 a boundary at their specification. For example, in the verification of `is_compressible`, we want
 431 to prove the precondition $n > 0$ of `buf_alloc` to call it and then resume afterwards with the
 432 postcondition $\text{buffer}_{\text{ML}}(V, n, [])$. To make this reasoning sound, we take inspiration from the work
 433 on open simulations [Hur et al. 2012] and of de Vilhena and Pottier [2021] on a program logic
 434 for effect handlers. Concretely, we parameterize Hoare triples by an *interface* $\Psi \in \text{Intf}(\text{Val}) \triangleq$
 435 $\text{FnName} \rightarrow \text{list}(\text{Val}) \rightarrow (\text{Val} \rightarrow \text{iProp}) \rightarrow \text{iProp}$ that maps each external call to its specification.
 436 Formally, an interface Ψ is a “predicate transformer” that takes in a function name fn , a list of
 437 argument values \vec{v} , and a postcondition Q and then produces the *precondition* “ $\Psi \text{ fn } \vec{v} Q$ ” required to
 438 call fn with arguments \vec{v} . The way that “skipping” function calls works with predicate transformers
 439 (see `CALL-EXTERNAL`) is that we show that the current precondition P implies $\Psi \text{ fn } \vec{v} Q$ where Q is
 440 our desired postcondition.
 441

```

442 16 value buf_alloc(value cap) {
443 17   CAMLparam1(cap); CAMLlocal3(bk, bf, r);
444 18   r = caml_alloc(1, 0); // allocate the `used` reference block
445 19   Store_field(r, 0, Val_int(0));
446 20   bk = caml_alloc_custom(sizeof(void*)); // allocate the `data` custom block
447 21   Custom_contents(bk) = malloc(Int_val(cap));
448 22   bf = caml_alloc(3, 0); // allocate the `{cap, used, data}` block
449 23   Store_field(bf, 0, cap); Store_field(bf, 1, r); Store_field(bf, 2, bk);
450 24   CAMLreturn(bf);
451 25 }

```

Fig. 5. FFI glue code for `buf_alloc`

For example, in the case of the buffer library, the interface $\Psi_{buf} \triangleq \Psi_{buf_alloc} \sqcup \Psi_{buf_free} \sqcup \dots \sqcup \Psi_{wrap_max_len}$ specifies all the library functions as a disjunction of single-function interfaces, where $(\Psi_1 \sqcup \Psi_2) fn \vec{v} \Phi \triangleq \Psi_1 fn \vec{v} \Phi \vee \Psi_2 fn \vec{v} \Phi$. For `buf_alloc` specifically, the interface is:

$$\Psi_{buf_alloc} fn \vec{V} \Phi \triangleq \exists n. fn = buf_alloc \wedge \vec{V} = [n] \wedge n > 0 * (\forall V. buffer_{ML}(V, n, []) \multimap \Phi V)$$

which we, more idiomatically, write as $\Psi_{buf_alloc} \triangleq \forall n. \langle n > 0 \rangle buf_alloc [n] \langle V. buffer_{ML}(V, n, []) \rangle$ (note the angle brackets!) to give single-function interfaces their familiar “Hoare triple reading”.³

With the interface Ψ_{buf} for the library in hand, the verification of `is_compressible` in `IrisML` proceeds smoothly *as if* the functions were implemented in λ_{ML} (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.⁴

³Formally, the notation is defined as $\forall \vec{x}. \langle P \rangle fn \vec{v} \langle v. \exists \vec{y}. Q \rangle \triangleq \lambda fn' \vec{v}' \Phi. fn' = fn * \exists \vec{x}. \vec{v}' = \vec{v} * P * (\forall v, \vec{y}. Q \multimap \Phi v)$. 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.

491 **Manipulating OCaml values in C.** As a function using the OCaml FFI, `buf_alloc` has to
 492 interact correctly with the OCaml runtime primitives (in **light blue**) and the OCaml garbage
 493 collector. We will ignore the garbage collector for now—and the primitives for working with it
 494 in **Line 17** and **CAMLreturn** in **Line 24**—and come back to it below. Instead, we focus on how to
 495 operate on runtime values. For integers, converting between an OCaml integer (as a runtime value)
 496 and a C integer is simple: the runtime provides the primitives `Val_int` (read “integer to value”, see
 497 **Line 19**) and `Int_val` (read “value to integer”, see **Line 21**). Interacting with structured values is
 498 more subtle. Recall the runtime representation of the buffer record `{cap, used, data}` depicted
 499 in **Fig. 1**. To create it, `buf_alloc` proceeds as follows: it allocates a runtime block `r` of size one
 500 for the reference used using the primitive `caml_alloc` (in **Line 18**) and initializes it to zero using
 501 the primitive `Store_field` (in **Line 19**); it allocates a “custom” runtime block `bk` (in **Line 20**) using
 502 `caml_alloc_custom`; it allocates a C buffer of the right length and stores it in the custom block
 503 (in **Line 21**); it allocates a block for the `buf` record and stores the capacity (here `len`), the used
 504 reference, and the custom block in it (in **Lines 22-23**); and, finally, it returns the newly created
 505 runtime block (in **Line 24**).

506
 507 **Tiptoeing around the OCaml garbage collector.** One fundamental difference between OCaml
 508 and C reveals itself only implicitly in the code in **Fig. 2**: the presence of the OCaml garbage collector
 509 (GC). That is, memory management in OCaml is based on garbage collection, meaning once a
 510 runtime allocated object (*i.e.*, one that is represented as a runtime block) becomes unreachable (*i.e.*,
 511 no part of the OCaml code can access it anymore), the GC can deallocate it to reduce the amount
 512 of memory consumed. This is in stark contrast to the manual memory management of C, where
 513 memory has to be explicitly allocated and eventually deallocated.

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

523 2.4 An Interface for the OCaml FFI

524
 525 How should we verify glue code such as `buf_alloc` in **Fig. 5**? On the one hand, the code is written
 526 in C, so it would be natural to use the language-local logic \mathbf{Iris}_C to reason about it. On the other
 527 hand, semantically, the code is more concerned with OCaml values (and the runtime representation
 528 thereof) than with C data structures, C pointers, and C-specific features. We answer this question
 529 with our next key idea: importing *a logic for the OCaml FFI* into the language-local logic \mathbf{Iris}_C .

530 Conceptually, “the OCaml FFI” is (1) a lower-level model of OCaml’s values and heap that is
 531 exposed to C; and (2) a set of “runtime primitives” available to C to operate on this lower-level
 532 representation. Together, these two parts of the OCaml FFI form a clean abstraction over the
 533 actual, underlying C data representation (and the implementation of the garbage collector). We
 534 use them as a “middle ground” between C and OCaml in this work. Concretely, (1) we define a
 535 notion of *runtime values*, *runtime heaps*, and *runtime separation logic resources*; and (2) we specify
 536 the runtime primitives as external functions in \mathbf{Iris}_C using an interface Ψ_{FFI} . In doing so, we get
 537 access to abstract reasoning principles about the OCaml runtime while retaining the language-local
 538 reasoning principles of \mathbf{Iris}_C .

539

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 $v \in \text{Val} ::= (n \in \mathbb{Z}) \mid (\gamma \in \text{Loc})$ are either integers or abstract runtime locations γ . Runtime locations “store” runtime blocks, which we track through separation logic resources: For standard blocks, we have $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$, which says that γ currently stores a block of values \vec{v} . It, additionally, asserts that the tag⁵ 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 γ 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 γ currently stores the λ_C -value w . Custom blocks are always mutable.

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

$$v \sim_C^\theta w \triangleq (\exists n. v = n \wedge w = \hat{n}) \vee (\exists \gamma. v = \gamma \wedge w = \theta(\gamma)) \quad \text{where } \theta \in \text{AddrMap} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Addr}$$

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

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 Ψ_{FFI}):

$$\Psi_{\text{alloc}} \triangleq \forall \theta, n, m. \langle \text{GC}(\theta) * 0 \leq n \rangle \text{alloc } [n, m] \langle w. \exists \theta', \gamma. \text{GC}(\theta') * \gamma \mapsto_{\text{blk}[m|\text{fresh}]} 0^n * \gamma \sim_C^{\theta'} w \rangle$$

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 θ initially. Since the allocation primitive `caml_alloc` calls the garbage collector internally, this map θ is potentially changed during the allocation, and we get back a new address map θ' after the call (and the GC resource). Moreover, for the newly allocated block, we get a fresh runtime location γ , which “stores” n -consecutive zeros as *runtime values*. To enable us to use the new block from λ_C , the return value w is related by $\gamma \sim_C^{\theta'} w$ to the freshly allocated runtime location γ in the postcondition. (We will see more runtime primitives and how to maintain references to λ_{ML} values across GC calls in §4.2.)

Using the interface Ψ_{FFI} , we can verify glue code in `Iris_C`. In particular, for `buf_alloc`, we prove:

$$\left\{ \text{GC}(\theta) * n \sim_C^\theta w * n \geq 0 \right\} \text{call buf_alloc } [w] @ p_{\text{lib}}, \Psi_{\text{FFI}} \left\{ w'. \exists \theta', \gamma. \text{GC}(\theta') * \gamma \sim_C^{\theta'} w' * \text{buf}_{\text{RT}}(\gamma, n, []) \right\}_C$$

where $\text{buf}_{\text{RT}}(\gamma, n, \vec{m}) \triangleq \exists \gamma_u, \gamma_d, a. \gamma \mapsto_{\text{blk}[0|\text{imm}]} [n, \gamma_u, \gamma_d] * \gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [0] * \gamma_d \mapsto_{\text{cstm}} a * \text{bf}(a, n, \vec{m})$ 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 $\gamma \mapsto_{\text{blk}[0|\text{imm}]} [n, \gamma_u, \gamma_d]$

⁵Tags are how OCaml distinguishes between different constructors of a datatype. In our examples, they are typically 0.

$$\begin{aligned}
& GC(\theta) * \ell \mapsto_{\text{ML}} \vec{V} \Rightarrow \exists \vec{v}, \gamma. GC(\theta) * \gamma \mapsto_{\text{blk}[0|\text{mut}]} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \quad (\text{ML-TO-FFI}) \\
& GC(\theta) * \gamma \mapsto_{\text{blk}[0|\text{mut}]} \vec{v} * \vec{V} \sim_{\text{ML}} \vec{v} \Rightarrow \exists \ell. GC(\theta) * \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \quad (\text{FFI-TO-ML})
\end{aligned}$$

Fig. 6. The view reconciliation rules

(with capacity n), (2) the reference for the **used** field $\gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [0]$, (3) the “custom block” $\gamma_a \mapsto_{\text{cstm}} a$ underneath **data**, which stores the address of the underlying λ_C buffer, and (4) the underlying buffer $\text{bf}(a, n, \vec{m})$.

2.5 View Reconciliation

We have sketched how to verify a mixed OCaml-and-C program from the OCaml side using Iris_{ML} and from the C side using Iris_{C} (including glue code using the OCaml FFI via Ψ_{FFI}). The last remaining piece of the puzzle is connecting the different parts, which brings us to the *view reconciliation problem*. Take `buf_alloc` again for example. We have discussed how to *assume a specification about it* in Iris_{ML} (in §2.2) and how to *prove a specification for it* in Iris_{C} (in §2.4). However, so far, the two specifications do not match up—the Iris_{C} -specification uses the resources for the runtime and the Iris_{ML} -specification the resources for λ_{ML} .

There is a fundamental challenge here. The two logics, Iris_{C} (extended with Ψ_{FFI}) and Iris_{ML} , are about entirely different languages, and yet they express views on *the same underlying physical state*. More specifically, they are language-local logics following the single-language recipe (from §1), which means they intentionally *do not model* how values and memory of their language are represented in other languages. Yet, whenever code on one side changes the underlying physical state, it becomes observable in the other. For example, after `buf_alloc` allocates the block r for the **used**-reference (obtaining $\gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [0]$), the block becomes observable in OCaml as a reference (in the form of $\ell \mapsto_{\text{ML}} [0]$) when `buf_alloc` returns. Subsequently, whenever the OCaml or C side mutate it (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., $\gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [0]$ and $\ell \mapsto_{\text{ML}} [1]$ 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., $\gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [0]$ and $\ell \mapsto_{\text{ML}} [0]$ are mutually exclusive). We can transition between the two perspectives using the *view reconciliation principles*, depicted in Fig. 6. They allow us to use Iris_{ML} resources when we are verifying C glue code in Iris_{C} . Concretely, the rule **ML-TO-FFI** allows us to turn $\ell \mapsto_{\text{ML}} \vec{V}$ (whenever we own the GC resource $GC(\theta)$) into the runtime block $\gamma \mapsto_{\text{blk}[0|\text{mut}]} \vec{v}$ using Iris’s resource updates [Jung et al. 2018, §5.4]. The runtime block location γ is tied to the λ_{ML} location ℓ through a new assertion $\ell \sim_{\text{ML}} \gamma$, which makes sure that γ always (uniquely) corresponds to ℓ . To relate the values of the runtime block \vec{v} and the reference \vec{V} , the relation “ \sim_{ML} ” is lifted to values as $\vec{V} \sim_{\text{ML}} \vec{v}$ (e.g., by relating `false` \sim_{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 λ_{ML} reference. The assertion $\ell \sim_{\text{ML}} \gamma$ 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_{C} and Iris_{ML} 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`, $\text{buffer}_{\text{ML}}(V, n, \vec{m})$ and $\text{buf}_{\text{RT}}(\gamma, n, \vec{m})$. To do so, we prove the

638 following view reconciliation rule:

$$639 \quad GC(\theta) * \text{buf}_{\text{RT}}(\gamma, n, \vec{m}) \Rightarrow \exists V. GC(\theta) * \text{buffer}_{\text{ML}}(V, n, \vec{m}) * V \sim_{\text{ML}} \gamma$$

640
641 which we can use to turn buf_{RT} into $\text{buffer}_{\text{ML}}$. The proof of this rule consists of three parts: First, we
642 turn the runtime block $\gamma_u \mapsto_{\text{blk}[0|\text{mut}]} [n]$ inside buf_{RT} into the mutable **used**-reference $\ell \mapsto_{\text{ML}} [n]$
643 inside $\text{buffer}_{\text{ML}}$ (using **FFI-TO-ML**). Second, we turn the custom block $\gamma_d \mapsto_{\text{cstm}} a * \text{bf}(a, n, \vec{m})$ inside
644 buf_{RT} into **bytes**(V', n, \vec{m}) inside $\text{buffer}_{\text{ML}}$. This is the place where we define **bytes**, which was
645 treated axiomatically while verifying **is_compressible**. Concretely, we define $\text{bytes}(V', n, \vec{m}) \triangleq$
646 $\exists \gamma_d, \iota, a. V' = \textcircled{L} * \textcircled{L} \sim_{\text{ML}} \gamma_d * \gamma_d \mapsto_{\text{cstm}} a * \text{bf}(a, n, \vec{m})$, which means V' is a foreign value \textcircled{L} that
647 corresponds to the custom runtime block γ_d , which stores the address a of the underlying bytes. (We
648 obtain the foreign identifier ι using additional rules given in the supplementary material [**Anonymous**
649 **2023**].) Third, we prove that the runtime block $\gamma \mapsto_{\text{blk}[0|\text{imm}]} [n, \gamma_u, \gamma_d]$ represents the λ_{ML}
650 buffer $V \triangleq \langle n, \ell, \textcircled{L} \rangle$, meaning $V \sim_{\text{ML}} \gamma$.

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

657 3 OPERATIONAL SEMANTICS

659 Zooming out, let us return to the multi-language recipe from §1. In this section, we focus on *the*
660 *operational semantics* side of the recipe. As a starting point, we take the canonical small-step opera-
661 tional semantics for the languages λ_{C} and λ_{ML} (defined in the supplementary material [**Anonymous**
662 **2023**]). We write $(e, \sigma) \rightarrow_{\text{ML}} (e', \sigma')$ for a step in λ_{ML} and $p; (c, \sigma) \rightarrow_{\text{C}} (c', \sigma')$ for a step in λ_{C} (where
663 p is the surrounding λ_{C} -program). Since these semantics operate on different values and have
664 different memory models, they cannot be simply “plugged together.” Thus, the big question is how
665 can we connect the two semantics to a multi-language semantics “ $\rightarrow_{\text{ML+C}}$ ”?

666
667 **Structure of the semantics.** To define the semantics $\rightarrow_{\text{ML+C}}$, we take inspiration from **Sammler**
668 **et al.** [2023]: we define it as a composition of smaller building blocks. In our setting, each building
669 block is a *language* λ with an associated notion of expressions $e \in \text{Expr}$, values $v \in \text{Val}$, state
670 $\sigma \in \text{State}$, functions $f \in \text{Func}$, programs $p \in \text{FnName} \xrightarrow{\text{fin}} \text{Func}$, and a small-step operational
671 semantics \rightarrow . Two languages λ_A and λ_B interact through external function calls `call fn \vec{v}` .

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

682 Putting everything together, we obtain the combined language $\lambda_{\text{ML+C}} \triangleq \lambda_{[\text{ML}]_{\text{FFI}} \oplus \text{C}}$ with its opera-
683 tional semantics $\rightarrow_{\text{ML+C}} \triangleq \rightarrow_{[\text{ML}]_{\text{FFI}} \oplus \text{C}}$ and its programs $[e]_{\text{FFI}} \oplus p$ where e is a λ_{ML} -expression and
684 p a λ_{C} -program. In the semantics $\rightarrow_{\text{ML+C}}$, external calls of e are made in terms of λ_{ML} ’s values and
685 memory model. They get translated by the wrapper to external calls of λ_{C} , which are then resolved
686

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 $[\cdot]_{\text{FFI}}$ bridges the gap between λ_{ML} and λ_{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 λ_{C} to λ_{ML} (e.g., when λ_{C} invokes a λ_{ML} -callback or returns to λ_{ML}), the wrapper needs to translate values from their low-level runtime representation to their high-level λ_{ML} representation. The issue is that this translation is, unfortunately, not unique! Different λ_{ML} -values V can have the same runtime representation v (e.g., integers and booleans are both runtime integers; pairs and arrays are both runtime blocks, etc.). When we go from λ_{C} to λ_{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 $p; s \twoheadrightarrow S$, 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 σ) 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_{C} and \rightarrow_{ML} use—as usual—only demonic non-determinism. Their semantics can be lifted in a generic way to multirelations $\twoheadrightarrow_{\text{C}}$ and $\twoheadrightarrow_{\text{ML}}$ (note the double arrow!). We define the semantics of the wrapper “ $[\cdot]_{\text{FFI}}$ ” 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 $[e]_{\text{FFI}}$ that (1) faithfully executes the code of e in the semantics of λ_{ML} and (2) provides a model of the runtime primitives such that they can be called from λ_{C} . The program $[e]_{\text{FFI}}$ does not have to be a *syntactic* λ_{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 λ_{C} -values in the λ_{C} -memory model. We use this freedom in the semantics $\twoheadrightarrow_{[\text{ML}]_{\text{FFI}}}$ 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 $\rho \in \text{State}$ (in Fig. 7) has two execution modes: **ML** and **C**. While it is executing the wrapped expression e , its state is **ML** $((\zeta, \chi, \sigma, rm), \sigma)$ where σ is the current λ_{ML} -state and the remaining state is part of the *runtime state* (discussed shortly). The wrapper transitions to the **C** execution mode whenever we are *at the boundary* between λ_{C} and λ_{ML} (e.g., we have called an external λ_{C} -function, or we are executing a FFI primitive called from λ_{C}). In the **C** execution mode, the state of the wrapper is **C** $((\zeta, \chi, \theta, rs), \sigma)$ where σ is the current λ_{C} -state

⁶We 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.

$$\begin{aligned}
736 \quad \zeta \in \text{BlockStore} &\triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Block} \quad b \in \text{Block} ::= \text{Vals}(t, \vec{v}) \mid \text{Custom}(w) \mid \text{Closure}(\text{rec } f x. e) \\
737 \\
738 \quad \chi \in \text{LocMap} &\triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Loc} \uplus \text{ForeignId} \uplus \{\bullet\} \quad \theta \in \text{AddrMap} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Addr} \\
739 \quad rm \in \text{rootMap} &\triangleq \text{Addr} \xrightarrow{\text{fin}} \text{Val} \quad rs \in \text{rootSet} \triangleq \text{set}(\text{Addr}) \\
740 \quad \rho \in \text{State} &::= \text{ML}((\zeta, \chi, \sigma, rm), \sigma) \mid \text{C}((\zeta, \chi, \theta, rs), \sigma) \\
741 \\
742 \\
743 \\
744
\end{aligned}$$

Fig. 7. Runtime state of the wrapper.

and the remaining state is again part of the runtime state. In this mode, the λ_{ML} -heap σ 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 λ_{C} and λ_{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 λ_{C} and λ_{ML} representation. The map ζ 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 $\text{Vals}(t, \vec{v})$ (with tag t , storing values \vec{v}), a “custom” block $\text{Custom}(w)$ (storing a λ_{C} value w), or a “closure” block $\text{Closure}(\text{rec } f x. e)$ representing a λ_{ML} function.

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

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

$$773 \quad \text{closed}(\theta, \zeta) \triangleq \forall (\gamma \in \text{dom } \theta)(b \in \text{Block}). \zeta(\gamma) = b \Rightarrow \forall \gamma' \in \text{locs}(b). \gamma' \in \text{dom } \theta \quad (\text{GC1})$$

$$774 \quad \text{roots}(\theta, rs) \triangleq \forall a \gamma. rm(a) = \gamma \Rightarrow \gamma \in \text{dom } \theta \quad (\text{GC2})$$

The first constraint, requires the address map θ to be transitively closed: for any runtime location γ in θ , any reachable location γ' 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 θ 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 $\lambda_{\text{ML}+\text{C}}$, 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 $\lambda_{\text{ML}+\text{C}}$ —are determined is left as an implementation choice to the GC so long as it does not deallocate reachable blocks (GC1). In $\lambda_{\text{ML}+\text{C}}$, registered roots are λ_{C} -addresses that store (the λ_{C} -addresses) of runtime

785 blocks γ , given by $\theta(\gamma)$. They are tracked in the roots map rm (and in the set rs when in C).⁷ Their
 786 intended behavior is that (1) the block γ is not deallocated by the GC and (2) whenever the runtime
 787 moves the block γ in memory, it will also update its address stored in the root. The first part is
 788 addressed by (GC2). The second part happens whenever we move from ML to C: The ML wrapper
 789 state contains a λ_C -memory σ where all the registered roots have been removed, the “residual”
 790 λ_C -memory. When we return to λ_C , the wrapper uses the freshly picked address map θ to “add in”
 791 the λ_C -representation of the roots.

792 **Angelic non-determinism.** As mentioned above, the *angelic* non-determinism comes into
 793 play when we move from λ_C to λ_{ML} . Concretely, when we transition from $C((\zeta, \chi, \theta, rs), \sigma)$ to
 794 $ML((\zeta, \chi, \sigma, rm), \sigma)$, we need to choose λ_{ML} -values V for runtime values v in their block representa-
 795 tion. However, the block representation is not unique, since, for example, `true` and `1` have the same
 796 runtime representation `1`. To pick “the right” value, the semantics uses angelic non-determinism.

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

805 **The runtime primitives.** The runtime provides the following primitives, a large subset of the
 806 OCaml FFI [oca 2023b], operating on the runtime state: The primitive `alloc` (for `caml_alloc`) allo-
 807 cates a new runtime block γ by extending ζ (with the block) and χ (with \bullet), “calls the GC” on θ , and
 808 then extends the new map with the λ_C address. Similarly, `alloc_custom` (for `caml_alloc_custom`)
 809 extends these maps with a new custom block. The primitives `Field` (for `Field`) and `Store_field`
 810 (for `Store_field`) access and update a field of a block in ζ . The primitives `read_custom` (for reading
 811 from `Custom_contents`) and `write_custom` (for writing to `Custom_contents`) access and update
 812 the contents of a custom block in ζ . The primitive `isblock` (for `Is_block`) checks whether a runtime
 813 value is a block, `length` (for `Wosize_val`) reads the length of the block, and `read_tag` (for `Tag_val`)
 814 reads the tag of the block. The primitives `registerroot` and `unregisterroot` can be used for registering
 815 and unregistering roots in the map rm . These primitives model what the runtime does when the
 816 macros `CAMLparam`, `CAMLlocal`, and `CAMLreturn` are executed. The primitive `callback` can be used
 817 to trigger the execution of a callback `rec f x. e` in λ_{ML} (see also §4.1).

819 4 PROGRAM LOGIC

820 As we have seen in §2, reasoning about programs is done at the level of the language-local logics **Iris_C**
 821 and **Iris_{ML}** in Melocoton. However, eventually, we have to ensure that this reasoning is sound (*i.e.*,
 822 that assumptions of one side are properly connected to proofs in the other). For this purpose, we
 823 introduce the “umbrella logic” **Iris_{ML+C}**, which embeds **Iris_C** and **Iris_{ML}**. We discuss how **Iris_{ML+C}**
 824 composes proofs (in §4.1), how it justifies the interface Ψ_{FFI} (in §4.2), and how it is modelled (in §4.3).

826 4.1 Composing Proofs in **Iris_{ML+C}**

827 In **Iris_{ML+C}**, we primarily reason at the level of interfaces $\Psi, \Pi \in \text{Intf}(Val)$. We write $\Psi \models p : \Pi$
 828 to mean that the program p implements interface Π under the “assumption” of interface Ψ . We

829 ⁷The distinction between the roots map rm and set rs arises, because during the λ_{ML} -execution we know *which* runtime
 830 value is rooted and has to be materialized again when moving to λ_C whereas during the execution of λ_C -code the value that
 831 is stored in a root can be changed.
 832
 833

834	$\text{INTFIMPLEMENT} \quad \forall fn \vec{v} P. \{ \Psi \quad fn \vec{v} P \} \text{ call } fn \vec{v} @ p, \Pi \{ v'. Pv' \} \quad \Psi \text{ fn is False for all } fn \notin \text{dom } p$	
835	$\Pi \models p : \Psi$	
836		
837		
838	$\text{INTFCONSEQ} \quad \Psi \sqsubseteq \Psi' \quad \Psi \models p : \Pi \quad \Pi' \sqsubseteq \Pi$	$\text{LINK} \quad \Pi \models_{\text{ML+C}} p_1 : \Psi \quad \Psi \models_{\text{ML+C}} p_2 : \Pi \quad \text{dom}(p_1) \# \text{dom}(p_2)$
839	$\Psi' \models p : \Pi'$	$\emptyset \models_{\text{ML+C}} p_1 \oplus p_2 : \Psi \sqcup \Pi$
840		
841		
842	$\text{EMBEDML} \quad \{ \text{True} \} e @ \emptyset, \Pi \{ V. \exists n. V = n * \phi(n) \}_{\text{ML}} \quad \Pi \text{ is } \perp \text{ for primitives}$	$\text{EMBEDC} \quad \Psi \models_c p : \Pi$
843	$[\Pi]_{\text{FFI}} \models_{\text{ML+C}} [e]_{\text{FFI}} : \Psi_{\text{FFI}}^{\Pi} \sqcup \text{MAIN}(\phi)$	$\Psi \models_{\text{ML+C}} p : \Pi$
844		
845		
846		
847		
848		
849		

Fig. 8. The interface rules of $\text{Iris}_{\text{ML+C}}$

can prove (also in Iris_{C} and Iris_{ML}) that p implements an interface by showing that its functions satisfy the specification given by Ψ against the interface Π (**INTFIMPLEMENT**). As one would expect, there is also an analogue to the Hoare rule of consequence (**INTFCONSEQ**) where $\Psi \sqsubseteq \Psi' \triangleq \forall fn \vec{v} P. \Psi \quad fn \vec{v} P \multimap \Psi' \quad fn \vec{v} P$. Using interfaces, we can succinctly state how proofs in the language-local logics (e.g., from §2) can be connected through $\text{Iris}_{\text{ML+C}}$:

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

- (1) $\{ \text{True} \} e @ \emptyset, \Pi \{ V. \exists n. V = n * \phi(n) \}_{\text{ML}}$ in Iris_{ML} and
- (2) $\Psi_{\text{FFI}}^{\Pi} \models_c p : [\Pi]_{\text{FFI}}$ in Iris_{C} ,

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

Let us break this theorem down. It allows linking a λ_{ML} -expression e with a λ_{C} -program p . For e , we need to prove a language-local triple (in Iris_{ML}) with a pure postcondition on integers ϕ (e.g., True for a proof of safety). We can do so against an arbitrary λ_{ML} -interface Π . For p , we need to prove that its functions implement the interface Π as we have claimed they do. Since Π is an interface over λ_{ML} -values, but we want to use it in Iris_{C} , we convert it into an interface on λ_{C} -values using the interface wrapper $[\cdot]_{\text{FFI}}$ (discussed below). To use runtime primitives, p is proven against the runtime interface Ψ_{FFI}^{Π} . Here, Ψ_{FFI}^{Π} gets the λ_{ML} -interface Π as an additional argument (which we omit in §2 for simplicity) to be able to execute λ_{ML} -callbacks from λ_{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 $\text{MAIN}(\phi) \triangleq \langle \text{atNit} \text{ main } [] \langle w. \exists n. w = \hat{n} \wedge \phi(n) \rangle \rangle$. Here, atNit 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 $\text{Iris}_{\text{ML+C}}$ (in Fig. 8). We can embed proofs into $\text{Iris}_{\text{ML+C}}$ from Iris_{C} (**EMBEDC**) and Iris_{ML} (**EMBEDML**). To embed an Iris_{ML} proof, it suffices to prove a Hoare triple about a λ_{ML} -expression e against the interface Π . We then obtain that the wrapped program $[e]_{\text{FFI}}$ implements the runtime interface Ψ_{FFI}^{Π} and the main interface $\text{MAIN}(\phi)$ —assuming the wrapped interface $[\Pi]_{\text{FFI}}$. After embedding proofs into $\text{Iris}_{\text{ML+C}}$, we can then “link” them together (**LINK**): if one side assumes Π and implements Ψ and the other assumes Ψ and implements Π , then they cancel out (i.e., the remaining assumption is \emptyset) and the linked program implements $\Psi \sqcup \Pi$. In the case of **Theorem 4.1**, the wrapped program $[e]_{\text{FFI}}$ assumes $[\Pi]_{\text{FFI}}$ and implements Ψ_{FFI}^{Π} whereas p assumes Ψ_{FFI}^{Π} and implements $[\Pi]_{\text{FFI}}$. Thus, they cancel out.

883	$\{GC(\theta)\} \text{ call alloc_custom } [] \{w. \exists \theta', v', w'. GC(\theta') * \gamma \mapsto_{\text{cstm}} w' * \gamma \sim_C^{\theta'} w\}$	(ALLOCCUSTOM)
884		
885	$\{GC(\theta) * \gamma \sim_C^{\theta} w * a \mapsto_C w\} \text{ call registerroot } [a] \{GC(\theta) * a \mapsto_{\text{root}} \gamma\}$	(REGISTERROOT)
886		
887	$\frac{\{P\} (\text{rec } f x. e') V @ \emptyset, \Pi \{V'. Q(V')\}_{\text{ML}}}{\{GC(\theta) * \gamma \mapsto_{\text{clos}} \text{rec } f x. e' * \triangleright P * \gamma \sim_C^{\theta} w_f * V \sim_{\text{ML}} v * v \sim_C^{\theta} w\} \text{ call callback } [w_f, w] @ [e]_{\text{FFI}}, [\Pi]_{\text{FFI}} \{w. \exists \theta' V' v' w'. GC(\theta') * Q(V') * V' \sim_{\text{ML}} v' * v' \sim_C^{\theta'} w'\}}$	(EXECCALLBACK)
888		
889		
890		
891		
892		
893		
894		
895		
896		
897		
898		
899		
900		
901		
902		
903		
904		
905		
906		
907		
908		
909		
910		
911		
912		
913		
914		
915		
916		
917		
918		
919		
920		
921		
922		
923		
924		
925		
926		
927		
928		
929		
930		
931		

Fig. 9. $\text{Iris}_{\text{ML+C}}$ Hoare triples for select runtime primitives.

What allows us to use the λ_{ML} -interface Π as a λ_{C} -interface in this proof is the wrapper $[\Pi]_{\text{FFI}}$:

$$[\Pi]_{\text{FFI}} \text{ fn } \vec{w} P \triangleq \exists \theta \vec{V} \vec{v} Q. GC(\theta) * \vec{V} \sim_{\text{ML}} \vec{v} * \vec{v} \sim_C^{\theta} \vec{w} * \Pi \text{ fn } \vec{V} Q * (\forall \theta' V' v' w'. Q(V') * GC(\theta') * V' \sim_{\text{ML}} v' * v' \sim_C^{\theta'} w' \multimap P(w'))$$

While its formal definition is quite a mouthful, its high-level intuition is comparatively simple: The wrapper $[\Pi]_{\text{FFI}}$ relates the λ_{C} -values to λ_{ML} -values through the runtime representation. Concretely, it (1) gives us access to the GC resource $GC(\theta)$ when entering the function on the λ_{C} -side (and demands it back with an updated map θ' when exiting the function), (2) connects the λ_{C} -arguments \vec{w} to λ_{ML} -arguments \vec{V} through their runtime representation \vec{v} , and (3) connects the λ_{ML} -return value V' through its runtime representation v' to the λ_{C} -return value w' .

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 Ψ_{FFI} . Operationally, these primitives are given semantics by $\multimap_{\text{ML+C}}$. At the program logic level, we prove their specifications in $\text{Iris}_{\text{ML+C}}$ (against the underlying semantics $\multimap_{\text{ML+C}}$). A key selection of these primitives is depicted in Fig. 9. The rule **ALLOCCUSTOM** allows allocating a custom block; it is analogous to the interface Ψ_{alloc} (from §2.4). The rule **REGISTERROOT** allows registering a λ_{C} -address a as a root with the runtime. If we do so, we obtain a new resource “ $a \mapsto_{\text{root}} \gamma$ ”, a root points-to, which asserts that a is registered as a root for the runtime block γ . The resource is stable across calls to the GC (*i.e.*, it does not depend on θ) and thus we can use it to access γ even after triggering the GC (*e.g.*, through **alloc**). The rule **EXECCALLBACK** allows executing a callback $\text{rec } f x. e'$ in the runtime. It uses a special points-to assertion “ \mapsto_{clos} ” for closures. The rule says that if we call f (with the λ_{C} -representation w of) V under precondition P , then we get (the λ_{C} -representation w' of) the value V' that results from executing f and the postcondition $Q(V')$. The “ \triangleright ” modality in this rule is Iris’s modality for *step-indexing*; we will discuss the effects of step-indexing on $\text{Iris}_{\text{ML+C}}$ shortly (in §4.3).

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

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

$$\text{GC}(\theta) \triangleq \exists \zeta_{\text{phys}}, \sigma_{\text{virt}}, \zeta_{\text{virt}}, \zeta_{\text{ml}}. \text{SI}_{\text{ML}}(\sigma_{\text{virt}}) * [\bullet(\zeta_{\text{virt}})] * \zeta_{\text{phys}} = \zeta_{\text{virt}} \uplus \zeta_{\text{ml}} * \text{repr}(\sigma_{\text{virt}}, \zeta_{\text{ml}}) * \dots$$

The GC resource is available while we are verifying glue code. The physical view that the operational semantics has at this point on the λ_{ML} -state is the runtime heap ζ_{phys} (the map ζ in execution mode **C** in Fig. 7). Logically, we split this heap into *two disjoint parts*, ζ_{virt} and ζ_{ml} . The part ζ_{virt} backs up the resource $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$ (through $[\bullet(\zeta_{\text{virt}})]$) and the part ζ_{ml} backs up the resource $\ell \mapsto_{\text{ML}} \vec{V}$. Of course, we cannot directly back $\ell \mapsto_{\text{ML}} \vec{V}$ with a runtime heap, since “ \mapsto_{ML} ” is a resource from **Iris**_{ML} which—as usual—is backed by a λ_{ML} -heap (through $\text{SI}_{\text{ML}}(\cdot)$, the “*state interpretation*” in the terminology of Iris). However, what we can do is maintain a separate, virtual λ_{ML} -heap σ_{virt} (not present in the underlying physical memory) which (1) backs the resource \mapsto_{ML} in the form of $\text{SI}_{\text{ML}}(\sigma_{\text{virt}})$ and (2) is faithfully represented by the runtime heap ζ_{virt} (denoted $\text{repr}(\sigma_{\text{virt}}, \zeta_{\text{ml}})$).

When we are verifying glue code in **Iris**_C, the way the double vision rules (see Fig. 6) work is by moving locations between σ_{virt} and ζ_{virt} as needed. Concretely, **ML-TO-FFI** removes ℓ from σ_{virt} and adds a corresponding runtime identifier γ (previously stored in ζ_{ml}) into ζ_{virt} . **FFI-TO-ML** does the opposite. When we start verifying a glue code function, the original λ_{ML} -heap is turned into σ_{virt} , and then we can gradually convert to and from the runtime representation as needed.

4.3 The Model of **Iris**_{ML+C}

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

THEOREM 4.2 (ADEQUACY OF **Iris_{ML+C}).** *Let p be a $\lambda_{\text{ML+C}}$ -program and ϕ a pure predicate on integers. If $\emptyset \models p : \text{MAIN}(\phi)$, then $p; (\text{call main } [], \emptyset) \longrightarrow_{\text{ML+C}}^* \{(\hat{n}, h) \mid \phi(n)\}$.*

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

$$p; s \longrightarrow^* S \triangleq_{\text{COIND}} s \in S \vee \exists S'. p; s \longrightarrow S' \wedge \forall s' \in S'. p; s' \longrightarrow^* S$$

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 $p; s \longrightarrow S$, 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+C}, so let us discuss this model first.

As usual for Iris, we define Hoare triples in terms of a *weakest precondition*, which in our case takes in the surrounding program p and interface Ψ . 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

$$\begin{array}{ll}
981 & \text{wp } e @ p, \Psi\{\Phi\} \quad \triangleq \quad \forall \sigma. \text{SI}(\sigma) \multimap \text{wp}'(e, \sigma) @ p, \Psi\{\Phi\} \\
982 & \text{(WP1) } \text{wp}'(v, \sigma) @ p, \Psi\{\Phi\} \quad \triangleq \quad \text{SI}(\sigma) * \Phi(v) \\
983 & \text{(WP2) } \text{wp}'(K[\text{call } fn \vec{v}], \sigma) @ p, \Psi\{\Phi\} \quad \triangleq \\
984 & \quad \text{SI}(\sigma) * fn \notin \text{dom}(p) * \exists \Phi'. \Psi \text{ fn } \vec{v} \Phi' * \triangleright \forall v'. \Phi'(v') \multimap \text{wp } K[v'] @ p, \Psi\{\Phi\} \\
985 & \text{(WP3) } \text{wp}'(e, \sigma) @ p, \Psi\{\Phi\} \quad \triangleq \quad \text{(e not a value nor a call)} \\
986 & \quad \exists S. p; (e, \sigma) \longrightarrow_{\text{ML+C}} S * \forall e' \sigma'. (e', \sigma') \in S \multimap \triangleright \multimap \text{SI}(\sigma') * \text{wp } e' @ p, \Psi\{\Phi\} \\
987 & \\
988 & \\
989 & \\
990 &
\end{array}$$

Fig. 10. Simplified weakest-precondition $\text{wp } e @ p, \Psi\{\Phi\}$ of $\mathbf{Iris}_{\text{ML+C}}$

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 “ \triangleright ” that appears in the cases (WP2) and (WP3). Step-indexing enables all kinds of recursive reasoning in Iris and, hence, $\mathbf{Iris}_{\text{ML+C}}$. We make use of it in our examples (in §5) and when reasoning about callbacks (see EXEC_CALLBACK). Third, the weakest precondition uses multi-relations $p; s \longrightarrow S$ instead of “ordinary” relations $p; s \rightarrow s'$ in (WP3). This is only the case for $\mathbf{Iris}_{\text{ML+C}}$; the logics \mathbf{Iris}_{C} and $\mathbf{Iris}_{\text{ML}}$ use weakest preconditions with ordinary relations as usual. We use multi-relations here to deal with the semantics $\longrightarrow_{\text{ML+C}}$.

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* $\mathbf{Iris}_{\text{ML+C}}$ and replay them “at the meta level” to resolve the angelic choices in $p; s \longrightarrow^* S$. (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 $p; s \longrightarrow^* S$. To “replay” the angelic choices outside of $\mathbf{Iris}_{\text{ML+C}}$, 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.

Buffer library & compression. In §2, we have primarily focused on the function `buf_alloc` as a running example. Most of the other external functions (in Fig. 2) are straightforward to verify. The most interesting one is `buf_upd`, because it uses callbacks. We prove the following specification:

$$\frac{\forall k. \{P(k)\} F(k) @ \emptyset, \Psi_{\text{buf}}\{u. u = f(k) * P(k+1)\}_{\text{ML}}}{\langle 0 \leq i \leq j < n * i \leq |\vec{m}| * \text{buffer}_{\text{ML}}(V, n, \vec{m}) * P(i) \rangle \text{buf_upd } [i, j, F, V]} \\
\langle V'. V' = \langle \rangle * \text{buffer}_{\text{ML}}(V, n, \vec{m}[i := f(i), \dots, j := f(j)]) * P(j+1) \rangle$$

where the predicate P describes the resources used by the callback F depending on the current index k and f is a mathematical function describing the integers that the callback computes. The specification asserts that `buf_upd` modifies the contents of the buffer between indices i and j according to f . Using this specification, we verify `is_compressible` by picking $P(_) \triangleq \ell \mapsto_{\text{ML}} \vec{n}$ (for the input λ_{ML} -array of characters) and $f(k) \triangleq \vec{n}[k]$ (for the k -th element of the array). Unlike well-typed functions in OCaml, the function `buf_upd` is not safe in general: if it is not used according to its specification, then it can exhibit undefined behavior (e.g., out-of-bounds accesses and iterator

invalidation). Thus, OCaml code that uses the buffer library must carefully respect its specification. The function `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 λ_{ML}), because there is no way to determine the *shape of values* (e.g., whether they are sums or pairs). For example, a λ_{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 λ_{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 λ_{ML} with a *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 λ_{C} : First, we add a new context Σ , which assigns a type $fn : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ to every external function fn in it. Then, we interpret the contexts using the interfaces Ψ of `IrisML` such that each external function $fn \in \Sigma$ is assigned (the semantic interpretation of) its type as a specification in Ψ . Finally, we use `IrisC` to *validate* the assumed types of external functions, which means we prove in `IrisC` that the λ_{C} -implementation satisfies the interpretation of the given type. Besides proving type safety of individual external functions fn , we can additionally use the logical relation to prove type safety of OCaml clients that wrap them in a safe abstraction (e.g., `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 λ_{ML} , and mutual recursion *through the FFI* (and the heap). Our version of Landin’s knot implements recursion through backpatching by combining λ_{ML} and λ_{C} code:

```
let knot (f : ('a -> 'b) -> ('a -> 'b)) =           value callk(value l, value x) {
  let l = ref (fun _ -> assert false) in           value f = Field(l, 0);
  l := (fun x -> f (fun y -> callk l y) x);         return caml_callback(f, x);
  (fun x -> callk l x)                             }
```

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

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 λ_{C} (i.e., memory storing closures), which we handle using step-indexing. The λ_{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 `listen`) to react to future events. When an event happens, the backend can notify (using `notify`) a listener which triggers the consumer’s callback. The library is implemented in λ_{C} and exposed using the FFI.

```
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 `listen` stores an arbitrary λ_{ML} -callback in a mutable data structure managed on the λ_{C} -side. The callback, by the nature

1079 of OCaml's type system, can capture the listener to which it is attached. This makes proving
 1080 type safety of the library tricky, because we can easily run into circularity issues when naively
 1081 attempting to define an interpretation of 'a listener' (e.g., a listener is safe if its callbacks are
 1082 safe, and callbacks are safe if the listeners they capture are safe). We resolve these circularity
 1083 issues as usual in these relations, by using *step-indexing*. Concretely, analogous to our standard
 1084 interpretation of reference types [Timany et al. 2022, Figures 5], we make use of Iris's impred-
 1085 icative invariants [Jung et al. 2018, §7.1], which internally are modelled using step-indexing. As
 1086 mentioned in §4.3, for the semantics $\rightarrow_{\text{ML+C}}$, this is only sound because we use a transfinitely step-
 1087 indexed version of Iris. Ultimately, we prove the functions (**create**, **listen**, **notify**) safe at type:
 1088 $\forall \alpha. \exists \text{listener}. (\text{unit} \rightarrow \text{listener}) \times ((\alpha \rightarrow \text{unit}) \rightarrow \text{listener} \rightarrow \text{unit}) \times (\alpha \rightarrow \text{listener} \rightarrow \text{unit})$.

1090 6 RELATED WORK

1091 Melocoton is, to our knowledge, the first program logic for programs spanning multiple languages
 1092 with different memory models. (The only other multi-language program logic that we are aware of
 1093 is Iris-Wasm discussed below.) Here, we also compare with work that tackles the related problem
 1094 of compiler verification in a multi-language setting (but is not based on a program logic).

1095 **Iris-Wasm.** Iris-Wasm [Rao et al. 2023] provides an Iris-based program logic for reasoning
 1096 about WebAssembly and its interaction with its host language, which in Iris-Wasm is a tiny subset
 1097 of JavaScript. The memory model of the host is a minor extension of WebAssembly's memory
 1098 model and hence, there is no view reconciliation challenge. In contrast, Melocoton shows how to
 1099 scale verification to more complex FFIs that require integrating program logics with very different
 1100 memory models and how to deal with the problems that arise such as view reconciliation (§2.5).

1101 **Cito.** Cito [Pit-Claudel et al. 2020; Wang et al. 2014] is a C-like language with a verified compiler
 1102 formalized in Coq, which supports linking with functions from other languages via axiomatic
 1103 specifications built into its operational semantics. While these axiomatic specifications follow the
 1104 style of open simulations [Hur et al. 2012] like the interfaces Ψ presented in this paper, they are not
 1105 phrased using a program logic, but stated using abstract data types and pure pre- and postconditions.
 1106 Cito uses program logics to reason about *individual* languages, but it does not tackle the problem
 1107 of building a *multi-language* program logic like $\text{Iris}_{\text{ML+C}}$.

1108 **Cogent.** Cheung et al. [2022] show how to extend the compiler correctness theorem of Co-
 1109 gent [O'Connor et al. 2016, 2021] with manually verified external C functions. Cheung et al. require
 1110 the C code to uphold the invariants guaranteed by Cogent's linear type system and focus on extend-
 1111 ing the correctness proof of the Cogent compiler using these invariants. In contrast, we assume
 1112 correctness of the OCaml compiler and runtime and instead focus on building a program logic for
 1113 verifying OCaml-and-C programs that maintain the (more complex) OCaml runtime invariants.

1114 **Semantic soundness for language interoperability.** Patterson et al. [2022] prove type safety
 1115 of the interaction between (among others) a MiniML-style garbage collected language and an L^3 -
 1116 style language with manual memory management. Instead of Melocoton's source-level reasoning,
 1117 they compile both languages to a common target language and build logical relations that relate
 1118 source-level types with target-level terms. This approach shifts reasoning to the target language.
 1119 For example, they relate values from different languages using target-level conversion functions
 1120 instead of Melocoton's source-level relations like $V \sim_{\text{ML}} v$ and $v \sim_{\text{C}} w$.

1121 **Verified compilers.** Mates et al. [2019] (building on the work of Patterson et al. [2017]) verify
 1122 a compiler from a stateful language with closures to one without closures in a syntactic multi-
 1123 language [Matthews and Findler 2007; Perconti and Ahmed 2014] using a logical relation for
 1124

1127

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

- 2023a. The OCaml Bigarray library. <https://v2.ocaml.org/api/Bigarray.html>
- 2023b. The OCaml manual – Chapter 22: Interfacing C with OCaml. <https://v2.ocaml.org/manual/intfc.html>
2023. The Snappy compression library. <https://github.com/google/snappy>
- Anonymous. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C (Coq development). Submitted as supplementary material with the paper.
- Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *JAR* 61, 1-4 (2018), 367–422. <https://doi.org/10.1007/s10817-018-9457-5>
- Louis Cheung, Liam O’Connor, and Christine Rizkallah. 2022. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) (CPP 2022). Association for Computing Machinery, New York, NY, USA, 13–26. <https://doi.org/10.1145/3497775.3503686>
- Paulo Emilio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proc. ACM Program. Lang.* 5, POPL, Article 33 (jan 2021), 28 pages. <https://doi.org/10.1145/3434314>
- Michael Furr and Jeffrey S. Foster. 2005. Checking type safety of foreign function calls. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 62–72. <https://doi.org/10.1145/1065010.1065019>
- Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608. <https://doi.org/10.1145/2676726.2676975>
- Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. 59–72. <https://doi.org/10.1145/2103656.2103666>
- Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation between ML and Assembly. In *POPL*. Association for Computing Machinery, New York, NY, USA, 133–146. <https://doi.org/10.1145/1926385.1926402>

- 1177 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the
1178 Rust Programming Language. *Proc. ACM Program. Lang.*, 2, POPL, Article 66 (dec 2017), 34 pages. <https://doi.org/10.1145/3158154>
- 1180 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the
1181 ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.
<https://doi.org/10.1017/S0956796818000151>
- 1182 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris:
1183 Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. <https://doi.org/10.1145/2676726.2676980>
- 1184 J er mie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI*. ACM, 1095–1109.
1185 <https://doi.org/10.1145/3453483.3454097>
- 1186 Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320.
- 1187 C. E. Martin, S. A. Curtis, and I. Rewitzky. 2007. Modelling Angelic and Demonic Nondeterminism with Multirelations. *Sci.*
1188 *Comput. Program.* 65, 2 (mar 2007), 140–158. <https://doi.org/10.1016/j.scico.2006.01.007>
- 1189 Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with
1190 Mutable State. In *PPDP*. ACM, 16:1–16:15. <https://doi.org/10.1145/3354166.3354181>
- 1191 Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: A Semantic Foundation
1192 for Functional Verification of Rust Programs with Unsafe Code. In *Proceedings of the 43rd ACM SIGPLAN International*
1193 *Conference on Programming Language Design and Implementation (San Diego, CA, USA) (PLDI 2022)*. Association for
1194 Computing Machinery, New York, NY, USA, 841–856. <https://doi.org/10.1145/3519939.3523704>
- 1195 Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *POPL*. ACM, 3–10.
1196 <https://doi.org/10.1145/1190216.1190220>
- 1197 Glen M elvel, Jacques-Henri Jourdan, and Fran ois Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml.
1198 *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (aug 2020), 29 pages. <https://doi.org/10.1145/3408978>
- 1199 Guillaume Munch-Maccagnoni and Gabriel Scherer. 2022. Boxroot, fast movable GC roots for a better FFI. In *ML Family*
1200 *Workshop*. Beno t Montagu, Ljubljana, Slovenia. <https://hal.inria.fr/hal-03910313>
- 1201 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a
1202 compositionally verified compiler for a higher-order imperative language. In *ICFP*. ACM, 166–178. <https://doi.org/10.1145/2784731.2784764>
- 1203 Liam O’Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas
1204 Sewell, and Gerwin Klein. 2016. Refinement through restraint: bringing down the cost of verification. In *ICFP*. ACM,
1205 89–102. <https://doi.org/10.1145/2951913.2951940>
- 1206 Liam O’Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas
1207 Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* 31 (2021), e25.
1208 <https://doi.org/10.1017/S095679682100023X>
- 1209 Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability.
1210 In *PLDI*. ACM, 609–624. <https://doi.org/10.1145/3519939.3523703>
- 1211 Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: reasonably mixing a functional
1212 language with assembly. In *PLDI*. ACM, 495–509. <https://doi.org/10.1145/3062341.3062347>
- 1213 James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS,*
1214 *Vol. 8410)*. Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8
- 1215 Cl ement Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction
1216 of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR (LNCS,*
1217 *Vol. 12167)*. 119–137. https://doi.org/10.1007/978-3-030-51054-1_7
- 1218 Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal.
1219 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. In *Proceedings of the 44th ACM SIGPLAN*
1220 *International Conference on Programming Language Design and Implementation (PLDI 2023)*. Association for Computing
1221 Machinery.
- 1222 Ingrid Rewitzky. 2003. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*.
1223 LNCS, Vol. 2929. Springer, 256–271. https://doi.org/10.1007/978-3-540-24615-2_12
- 1224 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC:
1225 Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI (Virtual, Canada) (PLDI*
1226 *2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. <https://doi.org/10.1145/3453483.3454036>
- 1227 Michael Sammler, Simon Spies, Youngju Song, Emanuele D’Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer.
1228 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7,
1229 POPL, Article 27 (jan 2023), 31 pages. <https://doi.org/10.1145/3571220>

- 1226 Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert
1227 with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31.
1228 <https://doi.org/10.1145/3371091>
- 1229 Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual
1230 Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. <https://doi.org/10.1145/3571232>
- 1231 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021.
1232 Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM*
1233 *SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021)*.
1234 Association for Computing Machinery, New York, NY, USA, 80–95. <https://doi.org/10.1145/3453483.3454031>
- 1235 Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. ACM,
1236 275–287. <https://doi.org/10.1145/2676726.2676985>
- 1237 Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. (2022).
1238 <https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf> (Under submission).
- 1239 Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data
1240 abstraction. In *OOPSLA*. ACM, 675–690. <https://doi.org/10.1145/2660193.2660201>
- 1241
- 1242
- 1243
- 1244
- 1245
- 1246
- 1247
- 1248
- 1249
- 1250
- 1251
- 1252
- 1253
- 1254
- 1255
- 1256
- 1257
- 1258
- 1259
- 1260
- 1261
- 1262
- 1263
- 1264
- 1265
- 1266
- 1267
- 1268
- 1269
- 1270
- 1271
- 1272
- 1273
- 1274