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

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, Derek Dreyer

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.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program verification**; **Separation logic**.

Additional Key Words and Phrases: foreign-function interfaces, OCaml, C, program logics, multi-language semantics, garbage collection, separation logic, angelic non-determinism, transfinite step-indexing, Iris, Coq

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

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

*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.

2475-1421/2023/10-ART247
https://doi.org/10.1145/3622823

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.

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).

[1] OCaml'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.

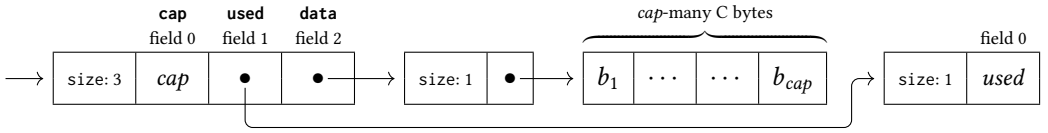

Fig. 1. C memory layout of the **buf**-record {cap, used, data} as exposed by the OCaml FFI

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: raw_bytes }
```

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

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

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

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

The operational semantics. At the operational semantics level, the challenge is that no existing multi-language semantics explains how to reconcile the key differences between both languages (*i.e.*, interacting with the OCaml garbage collector and runtime, registering "roots", executing callbacks, *etc.*). However, there are promising starting points. Patterson et al. [2022] combine a garbage collected language and a language with manual memory management. Unfortunately,

they do so using an elaboration semantics to a shared target language with garbage collection, which does not exist in the case of OCaml and C. Sammler et al. [2023] propose an approach to multi-language semantics that uses modular combinators to connect languages by composing their operational semantics. As such, their approach fits well with step one of the single-language recipe. Unfortunately, while they consider languages with significant differences (*e.g.*, different calling conventions and memory models), they do not consider garbage collection or a runtime.

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

1.2 Melocoton

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

(1) As a starting point, we take simplified versions of OCaml and C called λ_{C} and λ_{ML} (see §2.2) and—following the single-language recipe—we define their canonical operational semantics, $\rightarrow_{\mathsf{C}}$ and $\rightarrow_{\mathsf{ML}}$. 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 $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$, in the separation logic framework Iris [Jung et al. 2018, 2015].
(2) We extend the language-local program logics $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$ 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 $\mathbf{Iris}_{\mathsf{ML}}$) 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 $\rightarrow_{\mathsf{C}}$ and $\rightarrow_{\mathsf{ML}}$).
(3) Finally, for the combined language, $\lambda_{\mathsf{ML+C}}$, we develop an operational semantics $\longrightarrow\!\!\!\!\rightarrow_{\mathsf{ML+C}}$, 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, $\mathbf{Iris}_{\mathsf{ML+C}}$, which embeds $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$ and extends their reasoning principles to bridge between the two logics. We then prove that $\mathbf{Iris}_{\mathsf{ML+C}}$ is sound with respect to the joint semantics $\longrightarrow\!\!\!\!\rightarrow_{\mathsf{ML+C}}$.

In Melocoton, the verification of a mixed OCaml and C program can be done *almost entirely* in the language-local logics $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$. 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 $\mathbf{Iris}_{\mathsf{ML}}$ specification). In fact, even code interacting with the OCaml FFI—on the C side—can stay in the language-local logic $\mathbf{Iris}_{\mathsf{C}}$ (see §2.4). The only purpose of the "umbrella logic" $\mathbf{Iris}_{\mathsf{ML+C}}$ 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 $\mathbf{Iris}_{\mathsf{ML+C}}$ 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 list implementation that alternates between OCaml and C memory blocks, 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 $\mathbf{Iris}_{\mathsf{ML+C}}$, and extend it with reasoning principles for external calls. Melocoton is fully mechanized in Coq, and the Coq development can be found in the supplementary material [Guéneau et al. 2023]. The current development version of Melocoton and further information is available from the projet webpage at https://melocoton-project.github.io.

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.

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 raw_bytes and buf = { cap : int; used: int ref; data: raw_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.

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 the interface of Google's Snappy compression library [sna 2023]. For the purposes of the example, the implementation of the compression algorithm is not relevant.[2]

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 the that the two types, `char array` and `char *`, are not the same—they are not even part of the same language.

To circumvent this problem, we introduce *glue code*: C code that uses the OCaml FFI to mediate between `is_compressible` and `snappy_compress`. For now, we delay a discussion of the C implementation of the glue code (to §2.3) and focus on its OCaml interface (Lines 3-9). The interface consists of (1) a small buffer library (loosely inspired by OCaml's Bigarray library [oca 2023a]) and (2) wrappers of the compression functions operating on buffers. The interface declares an abstract

[2]In the Coq mechanization, we do not consider the actual Snappy compression algorithm, because its verification is besides the point of this paper. Instead, we verify a toy algorithm which can, in the worst case, double the size of the input.

λ_C

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

$c \in \mathit{Expr} ::= w \mid x \mid \ominus c \mid c \otimes c \mid \mathsf{malloc}(c) \mid \mathsf{free}(c, c) \mid *c \mid *c \leftarrow c \mid \mathsf{while}(c)\ c \mid \mathsf{call}\ \mathit{fn}\,\vec{c} \mid \cdots$

$p \in \mathit{Prog} ::= \varnothing \mid p, \mathit{fn}(\vec{x}) := c$

$\sigma \in \mathit{State} \triangleq \mathit{Addr} \xrightarrow{\text{fin}} \mathit{Cell}$ where $\mathit{cl} \in \mathit{Cell} \triangleq \{\star, \dagger\} \uplus \mathit{Val}$

λ_{ML}

$V \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \mathit{Loc}) \mid \mathsf{true} \mid \mathsf{false} \mid \langle\rangle \mid \langle V, V\rangle \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid \mathsf{rec}\ f\ x.\ e \mid \textcircled{1}$

$e \in \mathit{Expr} ::= V \mid x \mid e\ e \mid \ominus e \mid e \otimes e \mid \mathsf{alloc}\ e\ e \mid e.(e) \mid e.(e) \leftarrow e \mid \mathsf{length}\ e \mid \mathsf{call}\ \mathit{fn}\,\vec{e} \mid \cdots$

$\sigma \in \mathit{State} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathrm{list}(\mathit{Val}) \uplus \{\lightning\}$ $p \in \mathit{Prog} ::= \varnothing$

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

type `raw_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 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**$_{\mathsf{ML}}$.

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 and is uninitialized ($\star$) or has been freed already ($\dagger$). Memory is allocated with $\mathsf{malloc}(n)$ and has to be manually deallocated again with $\mathsf{free}(a, n)$. Executing $\mathsf{malloc}(n)$ 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

Iris$_C$

READ-C
$\{a \mapsto_C w\} *a \{w'.\ w = w' * a \mapsto_C w\}_C$

WRITE-C
$\{a \mapsto_C \_\} *a \leftarrow w' \{\_.\ a \mapsto_C w'\}_C$

ALLOC-C
$\{n \geq 0\}\ \mathrm{malloc}(n)\ \{a.\ \divideontimes_{0 \leq i < n}(a+i) \mapsto_C \star\}_C$

FREE-C
$\{\divideontimes_{0 \leq i < n}(a+i) \mapsto_C \_\}\ \mathrm{free}(a, n)\ \{\_.\ \mathrm{True}\}_C$

Iris$_{ML}$

READ-ML
$\{\ell \mapsto_{ML} \vec{V}\}\ \ell.(i)\ \{V'.\ V' = \vec{V}[i] * \ell \mapsto_{ML} \vec{V}\}_{ML}$

WRITE-ML
$\{\ell \mapsto_{ML} \vec{V}\}\ \ell.(i) \leftarrow V'\ \{\_.\ \ell \mapsto_{ML} \vec{V}[i := V']\}_{ML}$

ALLOC-ML
$\{n \geq 0\}\ \mathrm{alloc}\ n\ V\ \{\ell.\ \exists \vec{V}.\ \ell \mapsto_{ML} \vec{V} * |\vec{V}| = n * \forall i.\ \vec{V}[i] = V\}_{ML}$

Iris$_C$ and **Iris$_{ML}$**

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

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

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.

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 "$\{P\}\ \mathtt{call}\ fn\ \vec{w}\ @\ p, \Psi\ \{Q\}_C$" where $\mathtt{call}\ fn\ \vec{w}$ 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

$$\{n \geq 0\}\ \mathtt{call}\ \mathtt{snappy\_max\_compressed\_length}\ [n]\ \ @\ p_{\mathrm{lib}}, \varnothing\ \{w.\ \exists m.\ w = m * \mathrm{maxlen}(n, m)\}_C.$$

where p_{lib} is the surrounding compression library (and the interface is the empty interface $\varnothing$). 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").

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

$$\begin{array}{l}\left\{\mathrm{bf}(a_{in}, n_{in}, \overrightarrow{m_{in}}) * \mathrm{bf}(a_{out}, n_{max}, [\,]) * a_{outsz} \mapsto_C n_{max} * \mathrm{maxlen}(n_{in}, n_{max})\right\} \\ \quad \mathtt{call}\ \mathtt{snappy\_compress}\ [a_{in}, n_{in}, a_{out}, a_{outsz}]\ \ @\ p_{\mathrm{lib}}, \varnothing \\ \left\{w'.\ w' = 0 * \exists \overrightarrow{m_{out}}, n_{out}.\ \mathrm{bf}(a_{in}, n_{in}, \overrightarrow{m_{in}}) * \mathrm{bf}(a_{out}, n_{max}, \overrightarrow{m_{out}}) * \mathrm{cpr}(\overrightarrow{m_{in}}, \overrightarrow{m_{out}}) * a_{outsz} \mapsto_C |\overrightarrow{m_{out}}|\right\}_C\end{array}$$

where $\mathrm{bf}(a, n, [m_0, \ldots, m_{k-1}]) \triangleq (\divideontimes_{0 \leq i < k}(a+i) \mapsto_C m_i) * (\divideontimes_{k \leq i < n}(a+i) \mapsto_C \_) * k \leq n$ asserts that a stores a buffer of capacity n with the first k cells storing the integers $m_0, \ldots, m_{k-1}$, and we write $a \mapsto_C \_ \triangleq \exists cl \in Val \uplus \{\star\}.\ a \mapsto cl$ 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 $\mathrm{cpr}(\overrightarrow{m_{in}}, \overrightarrow{m_{out}})$), 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 "$(\!|\iota|\!)$" (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 $\texttt{alloc}\, n\, V$, but never has to be freed manually. Instead, conceptually, memory is garbage collected once no part of the program can access it anymore. Second, the memory of λ_{ML} is more complex, because it can store (lists of) arbitrary values, regardless of their shape or size (*e.g.*, $\langle\langle n, m\rangle, \texttt{rec}\, f\, x.\, e\rangle$, 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 $\texttt{length}\, \ell$, 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.*, $p = \varnothing$) and, instead, in λ_{ML} we execute single expressions e (which internally can contain let-bindings and mutual recursion).

There are three aspects of λ_{ML} that enable language interoperability. They are non-intrusive and minimal such that λ_{ML} does not even (need to) know which languages it is interacting with. First, there are foreign values "$(\!|\iota|\!)$", where ι is an abstract identifier. They can be used by other languages such as λ_{C} to "embed" their own data into λ_{ML}. We will use this feature of OCaml to embed the underlying C buffer into OCaml in our running example (in §2.5). Foreign values are abstract: they can be passed around via function calls, but there is no language construct in λ_{ML} to inspect their contents. Second, after executing an external function, locations in the heap of λ_{ML} can store data that is temporarily inaccessible from λ_{ML} (see §3.1). Whenever this is the case, the λ_{ML} heap stores $\lightning$ at the respective location and accesses in λ_{ML} will get stuck. Third, and most interestingly, λ_{ML} contains a language construct for *external function calls* $\texttt{call}\, fn\, \vec{V}$. The syntactic construct $\texttt{call}\, fn\, \vec{V}$ has no meaning in the language-local semantics of λ_{ML} (*i.e.*, it is stuck[3] in the semantics of λ_{ML}). As such, λ_{ML} does not have to include any rules in its semantics for executing code of other languages (*e.g.*, λ_{C}). Instead, we will assign meaning to external calls in the combined *multi-language* $\lambda_{\text{ML+C}}$ (see §3), where external calls in λ_{ML} are linked with their implementation in λ_{C}.

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

$$\{n > 0\}\ \texttt{buf\_alloc}(n)\ @\ \varnothing, \Psi_{buf}\{V.\, \text{buffer}_{\text{ML}}(V, n, [])\}_{\text{ML}}$$

[3]In general, "getting stuck" corresponds to reaching a "bad state" in the semantics (*e.g.*, a safety violation). By making $\texttt{call}\, fn\, \vec{V}$ stuck in the language-local semantics λ_{ML}, we ensure that one cannot obtain a closed proof of a λ_{ML}-program by only considering the λ_{ML}-side if the program has external calls. Instead, it is only be possible to verify such programs in the combined language $\lambda_{\text{ML+C}}$ by *additionally* considering the λ_{C} implementations of the external functions (see Theorem 4.2).

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

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

For example, in the case of the buffer library, the interface $\Psi_{buf} \triangleq \Psi_{\mathsf{buf\_alloc}} \sqcup \Psi_{\mathsf{buf\_free}} \sqcup \cdots \sqcup \Psi_{\mathsf{wrap\_max\_len}}$ 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:

$$\Psi_{\mathsf{buf\_alloc}}\ fn\ \vec{V}\ \Phi \triangleq \exists n.\ fn = \mathsf{buf\_alloc} \wedge \vec{V} = [n] \wedge n > 0 * (\forall V.\ \mathsf{buffer}_{\mathsf{ML}}(V, n, []) \mathrel{-\!\!*} \Phi V)$$

which we, more idiomatically, write as $\Psi_{\mathsf{buf\_alloc}} \triangleq \forall n.\ \langle n > 0 \rangle\ \mathsf{buf\_alloc}\ [n]\ \langle V.\ \mathsf{buffer}_{\mathsf{ML}}(V, n, []) \rangle$ (note the angle brackets!) to give single-function interfaces their familiar "Hoare triple reading".[4]

With the interface Ψ_{buf} for the library in hand, the verification of **is_compressible** in **Iris**$_{\mathsf{ML}}$ 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 [Guéneau et al. 2023].

The representation of OCaml values. In C, all OCaml values—regardless of their type, including **buf**, **raw_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` and `unit`. 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

[4] Formally, the notation is defined as $\forall \vec{x}.\ \langle P \rangle\ fn\ \vec{v}\ \langle v.\ \exists \vec{y}.\ Q \rangle \triangleq \lambda fn'\ \vec{v}'\ \Phi.\ fn' = fn * \exists \vec{x}.\ \vec{v}' = \vec{v} * P * (\forall v, \vec{y}.\ Q \mathrel{-\!\!*} \Phi\ v)$. The reader may cheerfully ignore this definition and stick with the intuition of Hoare triples as specifications.

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

Fig. 5. FFI glue code for **buf_alloc**

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.[5]

Manipulating OCaml values in C. As a function using the OCaml FFI, **buf_alloc** has to interact correctly with the OCaml runtime primitives (in **light blue**) and the OCaml garbage collector. We will ignore the garbage collector for now—and the primitives for working with it in Line 17 and **CAMLreturn** in Line 24—and come back to it below. Instead, we focus on how to operate on runtime values. For integers, converting between an integer n and the representation of n as a runtime value is simple: the runtime provides the primitives **Val_int** (read "integer to value", see Line 19) and **Int_val** (read "value to integer", see Line 21). These primitives actually do something—*e.g.*, **Val_int** converts n to $2n+1$ and **Int_val** right-shifts it back—because the OCaml runtime uses the least significant bit as a tag to distinguish integers from pointers.

Interacting with structured values is more subtle. Recall the runtime representation of the buffer record `{cap, used, data}` depicted in Fig. 1. To create it, **buf_alloc** proceeds as follows: it allocates a runtime block **r** of size one for the reference used using the primitive **caml_alloc** (in Line 18) and initializes it to zero using the primitive **Store_field** (in Line 19); it allocates a "custom" runtime block **bk** (in Line 20) using **caml_alloc_custom**; it allocates a C buffer of the right length and stores it in the custom block (in Line 21); it allocates a block for the **buf** record and stores the capacity (here **len**), the used reference, and the custom block in it (in Lines 22-23); and, finally, it returns the newly created runtime block (in Line 24).

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

One immediate consequence of this difference is that the buffer library needs to provide a **buf_free**-function to OCaml (Line 5), because otherwise the buffer will stay in memory forever. A more burdensome and subtle consequence is that we have to make sure that the GC does not

[5]In 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.

invalidate references to runtime blocks that we still want to use. That is, when we define an external function in C, the GC may execute whenever we make calls to (certain) OCaml FFI runtime functions (*e.g.*, `caml_alloc`). To prevent it from invalidating our local references (by deallocating or moving the runtime blocks that they point to), we have to "register" our local references with the GC as so-called "roots"(using `CAMLlocal` and `CAMLparam` in Line 17). Eventually, when we no longer need them, we can unregister our roots with the GC again (using `CAMLreturn` in Line 24).

2.4 An Interface for the OCaml FFI

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

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

The resources of the OCaml runtime. To define the runtime interface, we take the view that memory exposed by the OCaml FFI is an abstract *heap of runtime blocks* and then relate this heap to both its C and OCaml representations (*i.e.*, map it to pointers and integers in C and map it to structured values and references in OCaml). We will describe the interface of the OCaml runtime in more detail in §4.2. For now, we just take a glimpse at its values, resources, and rules.

The values of the OCaml runtime $v \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\gamma \in \mathit{Loc})$ 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 $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$, which says that γ currently stores a block of values $\vec{v}$. It, additionally, asserts that the tag[6] of the block is t and that the mutability of the block is $m \in \{\mathsf{imm}, \mathsf{mut}, \mathsf{fresh}\}$: 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 $\gamma \mapsto_{\mathrm{cstm}} w$, 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 $v \sim_{\mathrm{C}}^{\theta} w$:

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

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)

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

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 $\mathrm{GC}(\theta)$, 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 $\Psi_{\mathtt{FFI}}$):

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

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 $\mathrm{GC}(\theta)$ 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 $\lambda_{\mathtt{ML}}$ values across GC calls in §4.2.)

Using the interface $\Psi_{\mathtt{FFI}}$, we can verify glue code in $\mathbf{Iris}_C$. In particular, for **buf_alloc**, we prove:

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

where $\mathrm{buf}_{\mathrm{RT}}(\gamma, n, \vec{m}) \triangleq \exists \gamma_u, \gamma_d, a.\ \gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [n, \gamma_u, \gamma_d] * \gamma_u \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} [0] * \gamma_d \mapsto_{\mathrm{cstm}} a * \mathrm{bf}(a, n, \vec{m})$ 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 $\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [n, \gamma_u, \gamma_d]$ (with capacity n), (2) the reference for the **used** field $\gamma_u \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} [0]$, (3) the "custom block" $\gamma_d \mapsto_{\mathrm{cstm}} a$ underneath **data**, which stores the address of the underlying λ_C buffer, and (4) the underlying buffer $\mathrm{bf}(a, n, \vec{m})$.

2.5 View Reconciliation

We have sketched how to verify a mixed OCaml-and-C program from the OCaml side using $\mathbf{Iris}_{\mathtt{ML}}$ and from the C side using $\mathbf{Iris}_C$ (including glue code using the OCaml FFI via $\Psi_{\mathtt{FFI}}$). 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 $\mathbf{Iris}_{\mathtt{ML}}$ (in §2.2) and how to *prove a specification* for it in $\mathbf{Iris}_C$ (in §2.4). However, so far, the two specifications do not match up—the $\mathbf{Iris}_C$-specification uses the resources for the runtime and the $\mathbf{Iris}_{\mathtt{ML}}$-specification the resources for $\lambda_{\mathtt{ML}}$.

There is a fundamental challenge here. The two logics, $\mathbf{Iris}_C$ (extended with $\Psi_{\mathtt{FFI}}$) and $\mathbf{Iris}_{\mathtt{ML}}$, are about entirely different languages, and yet they express views on *the same underlying data*. 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. Each language-local logic models the language's own local account of the physical state, which is different on each side. Yet, whenever code on one side changes its underlying physical state, the changes become observable on the other side. For example, after **buf_alloc** allocates the block **r** for the **used**-reference (obtaining $\gamma_u \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} [0]$), the block becomes observable in OCaml as a reference (in the form of $\ell \mapsto_{\mathrm{ML}} [0]$) when **buf_alloc** returns. Subsequently, whenever the OCaml or C side mutates it (*e.g.*, in **wrap_compress**), the new value becomes also observable on the other side (*e.g.*, in **is_compressible**).

$$\mathrm{GC}(\theta) * \ell \mapsto_{\mathrm{ML}} \vec{V} \Rrightarrow \exists \vec{v}, \gamma.\ \mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} \vec{v} * \ell \sim_{\mathrm{ML}} \gamma * \vec{V} \sim_{\mathrm{ML}} \vec{v} \quad (\textsc{ml-to-ffi})$$

$$\mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v} \Rrightarrow \exists \ell.\ \mathrm{GC}(\theta) * \ell \mapsto_{\mathrm{ML}} \vec{V} * \ell \sim_{\mathrm{ML}} \gamma \quad (\textsc{ffi-to-ml})$$

Fig. 6. The view reconciliation rules

The fact that there are two ways of describing the same piece of data means we have to make sure that they stay "in sync" (*e.g.*, $\gamma_u \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} [0]$ and $\ell \mapsto_{\mathrm{ML}} [1]$ 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 on any piece of data—through the lens of *either* OCaml ($\ell \mapsto_{\mathrm{ML}} [0]$) *or* the runtime ($\gamma_u \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} [0]$), but not both. In the logic, we can transition between the two perspectives using the *view reconciliation principles*, depicted in Fig. 6. They allow us to use $\mathbf{Iris}_{\mathsf{ML}}$ resources when we are verifying C glue code in $\mathbf{Iris}_{\mathsf{C}}$. Concretely, the rule ML-TO-FFI allows us to turn $\ell \mapsto_{\mathrm{ML}} \vec{V}$ (whenever we own the GC resource $\mathrm{GC}(\theta)$) into the runtime block $\gamma \mapsto_{\mathrm{blk}[0|\mathrm{mut}]} \vec{v}$ 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 $\ell \sim_{\mathrm{ML}} \gamma$, 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 "$\sim_{\mathrm{ML}}$" is lifted to values as $\vec{V} \sim_{\mathrm{ML}} \vec{v}$ (*e.g.*, by relating false $\sim_{\mathrm{ML}}$ 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 $\ell \sim_{\mathrm{ML}} \gamma$ 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 $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$ 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**, $\mathrm{buffer}_{\mathrm{ML}}(V, n, \vec{m})$ and $\mathrm{buf}_{\mathrm{RT}}(\gamma, n, \vec{m})$. To do so, we prove the following view reconciliation rule:

$$\mathrm{GC}(\theta) * \mathrm{buf}_{\mathrm{RT}}(\gamma, n, \vec{m}) \Rrightarrow \exists V.\ \mathrm{GC}(\theta) * \mathrm{buffer}_{\mathrm{ML}}(V, n, \vec{m}) * V \sim_{\mathrm{ML}} \gamma$$

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

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

3 OPERATIONAL SEMANTICS

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

The language $\lambda_{\mathsf{ML+C}}$ and the semantics $\twoheadrightarrow_{\mathsf{ML+C}}$. We define the semantics $\twoheadrightarrow_{\mathsf{ML+C}}$ as a composition of smaller building blocks. Each building block is a *language* λ with an associated notion of expressions $e \in \mathit{Expr}$, values $v \in \mathit{Val}$, state $\sigma \in \mathit{State}$, functions $f \in \mathit{Func}$, programs $p \in \mathit{FnName} \xrightarrow{\text{fin}} \mathit{Func}$, and a small-step operational semantics $\twoheadrightarrow$. Two languages λ_A and λ_B interact through external calls $\texttt{call}\ \mathit{fn}\ \vec{v}$. Our basic building blocks are the semantics λ_{C} and λ_{ML}. Inspired by the approach of Sammler et al. [2023] (*i.e.*, defining multi-language semantics using modular combinators), we combine these building blocks using *combinators*.

In our case, to connect two languages λ_A and λ_B through their external calls, we introduce a language-generic linking combinator "$p_A \oplus p_B$" with its associated language $\lambda_{A \oplus B}$ (and semantics $\twoheadrightarrow_{A \oplus B}$). The linking combinator composes two languages with *the same* values and memory model: outgoing calls "$\texttt{call}\ \mathit{fn}\ \vec{v}$" from one side result in the execution of "$\mathit{fn}(\vec{v})$" in the other.

Of course, λ_{ML} and λ_{C} *differ* in their values and memory model (see Fig. 3). Thus, we cannot directly link a λ_{ML}-expression e and a λ_{C}-program p. To bridge the gap between them, we introduce a wrapping combinator "$[\cdot]_{\mathsf{FFI}}$" that embeds the language λ_{ML} into its own language $\lambda_{[\mathsf{ML}]_{\mathsf{FFI}}}$ (with semantics $\twoheadrightarrow_{[\mathsf{ML}]_{\mathsf{FFI}}}$). The wrapper takes a λ_{ML}-expression e and produces a program $[e]_{\mathsf{FFI}}$ with λ_{C}-values and the λ_{C}-memory model. However, the program $[e]_{\mathsf{FFI}}$ is *not* a syntactic λ_{C}-program; it is a program in the language $\lambda_{[\mathsf{ML}]_{\mathsf{FFI}}}$. The language $\lambda_{[\mathsf{ML}]_{\mathsf{FFI}}}$ uses λ_{C}-values and the λ_{C} memory model, but has a very different notion of expressions and semantics (see §3.1).

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

Modeling garbage collection. In the semantics of $\lambda_{\mathsf{ML+C}}$, we need to model the observable actions of the garbage collector (GC): when viewed from the C side, the OCaml GC can move objects in memory, as well as free memory that was previously storing OCaml values. To account for this behavior, our model of garbage collection is built around the following three key principles: First, we *decouple the identity of runtime values from their physical address in memory.* In the semantics, we distinguish between blocks in the "runtime heap" and their concrete physical addresses. This allows us to disentangle the concept of a "runtime value" (stable across runs of the GC) from its physical representation (which can change across GC runs). This principle was used previously by Hur and Dreyer [2011], albeit in proving compiler correctness in the presence of a GC, not in defining a multi-language semantics. Second, garbage collection is modeled as *a non-deterministic choice subject to reachability constraints*: the GC is modeled as non-deterministically changing or invalidating the physical addresses of blocks. This allows us to abstract over the concrete implementation of the OCaml GC. Non-determinism was also used by Moine et al. [2023], Wang et al. [2019], and Hur and Dreyer [2011] in their respective models of a GC. Third, we track the *registered FFI roots* in the

operational semantics. These roots are needed to make C accesses to runtime-managed memory sound: The OCaml FFI requires that C pointers to OCaml values are registered as "roots" with the GC (using FFI primitives). This registration ensures that the GC is aware of the values being used in C and, hence, does not erroneously deallocate them. Moreover, it allows the GC to update the C pointers when it decides to move the value they store in memory. In the semantics, we keep track of which pointers have been registered as roots and update them when appropriate.

Bridging language barriers with angelic non-determinism. The wrapper $[\![\cdot]\!]_{\mathrm{FFI}}$ 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.[7]

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 $p; s \longrightarrow\!\!\!\!\rightarrow S$, as steps in the operational semantics. Here, s is a configuration of the operational semantics (*i.e.*, a pair of an expression e and the state 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 $\rightarrow_{\mathrm{C}}$ and $\rightarrow_{\mathrm{ML}}$ use—as usual—only demonic non-determinism. Their semantics can be lifted in a generic way to multirelations $\longrightarrow\!\!\!\!\rightarrow_{\mathrm{C}}$ and $\longrightarrow\!\!\!\!\rightarrow_{\mathrm{ML}}$ (note the double arrow!). We define the semantics of the wrapper "$[\![\cdot]\!]_{\mathrm{FFI}}$" and linking combinator "$\oplus$" in terms of multirelations. (See the Coq development [Guéneau et al. 2023] for the full definition of the combinators.)

3.1 View Reconciliation in the Wrapper Semantics

We focus on the most interesting part of the semantics: the wrapper semantics and how it deals with the *view reconciliation problem* (see §2.5) at the level of the operational 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 $[\![e]\!]_{\mathrm{FFI}}$ 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 $[\![e]\!]_{\mathrm{FFI}}$ does not have to be a *syntactic* λ_{C}-program. Instead, since the linking combinator "$\oplus$" connects languages through external calls $\texttt{call}\ fn\ \vec{v}$, 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 $\longrightarrow\!\!\!\!\rightarrow_{[\![\mathrm{ML}]\!]_{\mathrm{FFI}}}$ by implementing the "functions" of $[\![e]\!]_{\mathrm{FFI}}$ (*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 $\rho \in \mathit{State}$ (in Fig. 7) has two execution modes: ML and C. While it is executing the wrapped expression e, its state is $\mathsf{ML}\ ((\zeta, \chi, \sigma, \mathit{rm}), \sigma)$ where σ is the current λ_{ML}-state and the remaining state is part of the *runtime state* (discussed shortly). The

[7] We do not use types to guide this choice as our wrapper—like the OCaml runtime—does not track type information and even with type information the choice would be unclear for functions with polymorphic types.

$$\zeta \in \mathit{BlockStore} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Block} \qquad b \in \mathit{Block} ::= \mathsf{Vals}(t, \mu, \vec{v}) \mid \mathsf{Custom}(w) \mid \mathsf{Closure}(\mathsf{rec}\, f\, x.\, e)$$
$$\chi \in \mathit{LocMap} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Loc} \uplus \mathit{ForeignId} \uplus \{\bullet\} \qquad \theta \in \mathit{AddrMap} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Addr}$$
$$\mathit{rm} \in \mathit{rootMap} \triangleq \mathit{Addr} \xrightarrow{\text{fin}} \mathit{Val} \qquad \mathit{rs} \in \mathit{rootSet} \triangleq \mathrm{set}(\mathit{Addr})$$
$$\rho \in \mathit{State} ::= \mathsf{ML}\,((\zeta, \chi, \sigma, \mathit{rm}), \sigma) \mid \mathsf{C}\,((\zeta, \chi, \theta, \mathit{rs}), \sigma)$$

Fig. 7. Runtime state of the wrapper.

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 $\mathsf{C}\,((\zeta, \chi, \theta, \mathit{rs}), \sigma)$ where σ is the current λ_{C}-state 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. Throughout the execution, the wrapper state thus switches between the ML and C views of the state. These two views represent the same data, *e.g.*, a reference on one side and a block on the other. When switching sides, it is the task of the wrapper to propagate changes made on one side into corresponding changes on the other side.

The runtime state (*i.e.*, ζ, χ, θ, σ, 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). In particular, it is stable under the action of the GC, as it describes an abstract view of blocks, independently from their physical representation. A block can be either a standard block $\mathsf{Vals}(t, \mu, \vec{v})$ (with tag t, mutability $\mu \in \{\mathsf{Mut}, \mathsf{Immut}\}$, storing values $\vec{v}$), a "custom" block $\mathsf{Custom}(w)$ (storing a λ_{C} value w), or a "closure" block $\mathsf{Closure}(\mathsf{rec}\, f\, x.\, e)$ 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 wrapper state has been carefully designed to move between the two views of the state with "just the right" amount of non-determinism: the wrapper needs to expose enough information about the runtime, but must not overspecify its behavior. For instance, we expose and track the mutability μ of standard blocks, encoding the runtime's expectation that one must not modify a block that is supposed to be immutable. (While the concrete implementation of the runtime will give some behavior to this operation, performing an update to an "immutable" block violates an implicit OCaml assumption that the compiler can depend on for optimizations.) But we are also careful to leave some flexibility in how blocks are used to allow compiler optimizations (*e.g.*, representationally identical, immutable values such as the λ_{ML} pairs $\langle 42, \mathsf{true}\rangle$ and $\langle 42, 1\rangle$ can reuse the same runtime blocks).

The garbage collector and registered roots. In $\lambda_{\mathsf{ML+C}}$, we do not fix a specific garbage collector implementation. Instead, the semantics of $\lambda_{\mathsf{ML+C}}$ should be sound with respect to all reasonable

garbage collectors. Thus, as outlined above, we model the GC using non-determinism. That is, 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 some runtime primitives such as **caml_alloc**). This choice over θ is made *demonically* (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:

$$\mathsf{closed}(\theta, \zeta) \triangleq \forall (\gamma \in \mathrm{dom}\,\theta)(b \in \mathit{Block}).\ \zeta(\gamma) = b \Rightarrow \forall \gamma' \in \mathsf{locs}(b).\ \gamma' \in \mathrm{dom}\,\theta \tag{GC1}$$

$$\mathsf{roots}(\theta, \mathit{rs}) \triangleq \forall a\, \gamma.\ \mathit{rm}(a) = \gamma \Rightarrow \gamma \in \mathrm{dom}\,\theta \tag{GC2}$$

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 $\mathsf{locs}(b)$ 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 λ_{ML+C}, it suffices to track those roots that have been explicitly registered with the GC through an FFI primitive (explained below). How the other GC roots—which we do not model in λ_{ML+C}—are determined is left as an implementation choice to the GC so long as it does not deallocate reachable blocks (GC1). In λ_{ML+C}, registered roots are λ_C-addresses that store (the λ_C-addresses) of runtime blocks γ, given by $\theta(\gamma)$. They are tracked in the roots map rm (and in the set rs when in C).[8] Their intended behavior is that (1) the block γ is not deallocated by the GC and (2) whenever the runtime *moves the block γ in memory*, it will also *update its address* stored in the root. The first part is addressed by (GC2). The second part happens whenever we move from ML to C: The ML wrapper state contains a λ_C-memory σ where all the registered roots have been removed, the "residual" λ_C-memory. When we return to λ_C, the wrapper uses the freshly picked address map θ to "add in" the λ_C-representation of the roots.

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

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

The runtime primitives. The runtime provides the following primitives, a large subset of the OCaml FFI [oca 2023b], operating on the runtime state: The primitive alloc (for **caml_alloc**) allocates a new runtime block γ by extending ζ (with the block) and χ (with $\bullet$), "calls the GC" on θ, and then extends the new map with the λ_C address. Similarly, alloc_custom (for **caml_alloc_custom**) extends these maps with a new custom block. The primitives Field (for **Field**) and Store_field (for **Store_field**) access and update a field of a block in ζ. The primitives read_custom (for reading from **Custom_contents**) and write_custom (for writing to **Custom_contents**) access and update

[8] During the λ_{ML}-execution (where we have rm) we know *which* runtime value is rooted and has to be materialized again when moving to λ_C; whereas during λ_C-execution (where we have rs) the value that is stored in a root can be changed.

$$\textsc{IntfImplement}\ \frac{\forall fn\,\vec{v}\,P.\ \{\Psi\ fn\ \vec{v}\ P\}\ \texttt{call}\ fn\ \vec{v}\ @\ p, \Pi\ \{v'.\ Pv'\} \qquad \Psi\ fn \text{ is False for all } fn \notin \mathrm{dom}\ p}{\Pi \models p : \Psi}$$

$$\textsc{IntfConseq}\ \frac{\Psi \sqsubseteq \Psi' \qquad \Psi \models p : \Pi \qquad \Pi' \sqsubseteq \Pi}{\Psi' \models p : \Pi'} \qquad \textsc{Link}\ \frac{\Pi \models_{\mathsf{ML+C}} p_1 : \Psi \qquad \Psi \models_{\mathsf{ML+C}} p_2 : \Pi \qquad \mathrm{dom}(p_1)\#\mathrm{dom}(p_2)}{\varnothing \models_{\mathsf{ML+C}} p_1 \oplus p_2 : \Psi \sqcup \Pi}$$

$$\textsc{EmbedML}\ \frac{\{\mathrm{True}\}\ e\ @\ \varnothing, \Pi\ \{V.\ \exists n.\ V = n * \phi(n)\}_{\mathrm{ML}} \qquad \Pi \text{ is } \bot \text{ for primitives}}{[\Pi]_{\mathrm{FFI}} \models_{\mathsf{ML+C}} [e]_{\mathrm{FFI}} : \Psi^{\Pi}_{\mathrm{FFI}} \sqcup \textsc{main}(\phi)} \qquad \textsc{EmbedC}\ \frac{\Psi \models_{\mathsf{C}} p : \Pi}{\Psi \models_{\mathsf{ML+C}} p : \Pi}$$

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

the contents of a custom block in ζ. The primitive isblock (for `Is_block`) checks whether a runtime value is a block, length (for `Wosize_val`) reads the length of the block, and read_tag (for `Tag_val`) reads the tag of the block. The primitives registerroot and unregisterroot can be used for registering and unregistering roots in the map rm. These primitives model what the runtime does when the macros `CAMLparam`, `CAMLlocal`, and `CAMLreturn` are executed. The primitive callback can be used to trigger the execution of a callback $\mathrm{rec}\ f\ x.\ e$ in λ_{ML} (see also §4.1).

4 PROGRAM LOGIC

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

4.1 Composing Proofs in $\mathbf{Iris}_{\mathsf{ML+C}}$

In $\mathbf{Iris}_{\mathsf{ML+C}}$, we primarily reason at the level of interfaces $\Psi, \Pi \in \mathit{Intf}(\mathit{Val})$. We write $\Psi \models p : \Pi$ to mean that the program p implements interface Π under the "assumption" of interface Ψ. We can prove (also in $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$) 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 $\Psi \sqsubseteq \Psi' \triangleq \forall fn\,\vec{v}\,P.\ \Psi\ fn\ \vec{v}\ P \mathrel{-\!\!*} \Psi'\ fn\ \vec{v}\ P$. Using interfaces, we can succinctly state how proofs in the language-local logics (*e.g.*, from §2) can be connected through $\mathbf{Iris}_{\mathsf{ML+C}}$:

Theorem 4.1 (Connecting $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$). *Let e be a λ_{ML}-expression, p a λ_{C}-program (where $\mathrm{dom}\ p$ does not contain FFI primitives), Π a λ_{ML}-interface, and ϕ a pure predicate on integers. If*

(1) $\{\mathrm{True}\}\ e\ @\ \varnothing, \Pi\ \{V.\ \exists n.\ V = n * \phi(n)\}_{\mathrm{ML}}$ *in* $\mathbf{Iris}_{\mathsf{ML}}$ *and*
(2) $\Psi^{\Pi}_{\mathrm{FFI}} \models_{\mathsf{C}} p : [\Pi]_{\mathrm{FFI}}$ *in* $\boldsymbol{Iris}_{\mathsf{C}}$,

then we have $\varnothing \models_{\mathsf{ML+C}} [e]_{\mathrm{FFI}} \oplus p : \textsc{main}(\phi)$ *in* $\mathbf{Iris}_{\mathsf{ML+C}}$.

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 $\mathbf{Iris}_{\mathsf{ML}}$) 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

$$\{\mathsf{GC}(\theta)\}\ \texttt{call alloc\_custom}\ []\ \{w.\ \exists \theta', v', w'.\ \mathsf{GC}(\theta') * \gamma \mapsto_{\mathrm{cstm}} w' * \gamma \sim_{\mathrm{C}}^{\theta'} w\}\quad (\textsc{AllocCustom})$$

$$\{\mathsf{GC}(\theta) * \gamma \sim_{\mathrm{C}}^{\theta} w * a \mapsto_{\mathrm{C}} w\}\ \texttt{call registerroot}\ [a]\ \{\mathsf{GC}(\theta) * a \mapsto_{\mathrm{root}} \gamma\}\quad (\textsc{RegRoot})$$

$$\{\mathsf{GC}(\theta) * a \mapsto_{\mathrm{root}} \gamma\}\ \texttt{call unregisterroot}\ [a]\ \{\exists w.\ \mathsf{GC}(\theta) * \gamma \sim_{\mathrm{C}}^{\theta} w * a \mapsto_{\mathrm{C}} w\}\quad (\textsc{UnregRoot})$$

$$\frac{\{P\}\ (\mathrm{rec}\ f\ x.\ e')\ V\ @\ \varnothing, \Pi\ \{V'.\ Q(V')\}_{\mathrm{ML}}}{\begin{array}{l}\{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{clos}} \mathrm{rec}\ f\ x.\ e' * \triangleright P * \gamma \sim_{\mathrm{C}}^{\theta} w_f * V \sim_{\mathrm{ML}} v * v \sim_{\mathrm{C}}^{\theta} w\} \\ \quad \texttt{call callback}\ [w_f, w]\ @\ [e]_{\mathrm{FFI}}, [\Pi]_{\mathrm{FFI}} \\ \{w.\ \exists \theta'\ V'\ v'\ w'.\ \mathsf{GC}(\theta') * Q(V') * V' \sim_{\mathrm{ML}} v' * v' \sim_{\mathrm{C}}^{\theta'} w'\}\end{array}}\quad (\textsc{ExecCallback})$$

Fig. 9. **Iris**$_{\mathsf{ML+C}}$ Hoare triples for selected runtime primitives.

interface over λ_{ML}-values, but we want to use it in **Iris**$_{\mathsf{C}}$, we convert it into an interface on λ_{C}-values using the interface wrapper $[\cdot]_{\mathrm{FFI}}$ (discussed below). To use runtime primitives, p is proven against the runtime interface $\Psi_{\mathrm{FFI}}^{\Pi}$. Here, Ψ_{FFI} 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 $[e]_{\mathrm{FFI}} \oplus p$ implements the interface $\textsc{main}(\phi) \triangleq \langle \mathsf{atInit} \rangle\ \mathsf{main}\ []\ \langle w.\ \exists n.\ w = \hat{n} \wedge \phi(n) \rangle$. Here, atInit is a resource that signals program start and main is a "primitive" of the wrapper that will trigger the execution of e.

The proof of Theorem 4.1 is straightforward using the key properties of **Iris**$_{\mathsf{ML+C}}$ (in Fig. 8). We can embed proofs into **Iris**$_{\mathsf{ML+C}}$ from **Iris**$_{\mathsf{C}}$ (EmbedC) and **Iris**$_{\mathsf{ML}}$ (EmbedML). To embed an **Iris**$_{\mathsf{ML}}$ 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 $[e]_{\mathrm{FFI}}$ implements the runtime interface $\Psi_{\mathrm{FFI}}^{\Pi}$ and the main interface $\textsc{main}(\phi)$—assuming the wrapped interface $[\Pi]_{\mathrm{FFI}}$. After embedding proofs into **Iris**$_{\mathsf{ML+C}}$, 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 $\varnothing$) and the linked program implements $\Psi \sqcup \Pi$. In the case of Theorem 4.1, the wrapped program $[e]_{\mathrm{FFI}}$ assumes $[\Pi]_{\mathrm{FFI}}$ and implements $\Psi_{\mathrm{FFI}}^{\Pi}$ whereas p assumes $\Psi_{\mathrm{FFI}}^{\Pi}$ and implements $[\Pi]_{\mathrm{FFI}}$. Thus, they cancel out.

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

$$\begin{array}{rcl}[\Pi]_{\mathrm{FFI}}\ fn\ \vec{w}\ P & \triangleq & \exists \theta\ \vec{V}\ \vec{v}\ Q.\ \mathsf{GC}(\theta) * \vec{V} \sim_{\mathrm{ML}} \vec{v} * \vec{v} \sim_{\mathrm{C}}^{\theta} \vec{w} * \Pi\ fn\ \vec{V}\ Q * \\ & & (\forall \theta'\ V'\ v'\ w'.\ Q(V') * \mathsf{GC}(\theta') * V' \sim_{\mathrm{ML}} v' * v' \sim_{\mathrm{C}}^{\theta'} w' \mathrel{-\!\!*} P(w'))\end{array}$$

While its formal definition is quite a mouthful, its high-level intuition is comparatively simple: The wrapper $[\Pi]_{\mathrm{FFI}}$ relates the λ_{C}-values to λ_{ML}-values through the runtime representation. Concretely, it (1) gives us access to the GC resource $\mathsf{GC}(\theta)$ when entering the function on the λ_{C}-side (and demands it back with an updated map θ' when existing 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 $\twoheadrightarrow_{\mathsf{ML+C}}$. At the program logic level, we prove their specifications in **Iris**$_{\mathsf{ML+C}}$ (against the underlying semantics $\twoheadrightarrow_{\mathsf{ML+C}}$). 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 $\Psi_{\mathtt{alloc}}$ (from §2.4). The rule RegRoot 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).

Unregistering the root (UnregRoot) gives back ownership of the underlying λ_C points-to. While the root is registered, one is prevented from freeing the underlying memory. (Note that the program logic permits roots to remain registered indefinitely, and thus does not guarantee the absence of memory leaks.)

The rule ExecCallback allows executing a callback $\text{rec}\, f\, x.\, e'$ in the runtime. It uses a special points-to assertion "$\mapsto_{\text{clos}}$" 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 $\mathbf{Iris}_{\mathsf{ML}+\mathsf{C}}$ shortly (in §4.3).

View reconciliation. One of the main challenges, also at the level of the program logic, is *view reconciliation*. That is, we have to soundly combine the resources $a \mapsto_C cl$ of $\mathbf{Iris}_C$, $\ell \mapsto_{\text{ML}} \vec{V}$ of $\mathbf{Iris}_{\mathsf{ML}}$, and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$ of the runtime. Fortunately, the resources $a \mapsto_C cl$ and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$ never overlap, because the operational semantics separates the runtime-managed memory ζ (whose resources are of the form $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$) from the manually-managed C memory σ (whose resources are of the form $a \mapsto_C cl$). Thus, we focus on the interaction of $\ell \mapsto_{\text{ML}} \vec{V}$ and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$. The resources $\ell \mapsto_{\text{ML}} \vec{V}$ and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$ can in fact overlap (see §2.5), as they can describe different views on the same piece of data. For example, 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.

As far as the user of $\mathbf{Iris}_{\mathsf{ML}+\mathsf{C}}$ is concerned, these two views are mediated by the view reconciliation rules (Fig. 6). While these rules are in some sense very natural, justifying their soundness requires care: The crux is that there is an inherent disconnect between the physical state of the operational semantics (*i.e.*, σ and ζ) and the logical state in terms of points-to assertions (*i.e.*, $\ell \mapsto_{\text{ML}} \vec{V}$ and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$). In the operational semantics, there is only one view of the λ_{ML}-heap at any given time: either as a runtime heap ζ or a λ_{ML}-heap σ.[9] (This is essential to reuse the language-local semantics, which are only defined in terms of the language-specific heap.) The logic, however, allows more fine-grained views, where $\ell' \mapsto_{\text{ML}} \vec{V}$ and $\gamma' \mapsto_{\text{blk}[t|m]} \vec{v}$ can exist at the same time for different locations ℓ' and γ' (both during the execution in λ_{ML} and in λ_C). The coexistence of runtime points-to assertions and λ_{ML}-points-to assertions is not only convenient (*e.g.*, it enables the view reconciliation rules in Fig. 6), it is essential for core reasoning principles of separation logic such as framing (*e.g.*, the language-local logic $\mathbf{Iris}_{\mathsf{ML}}$ can frame its resources $\ell \mapsto_{\text{ML}} \vec{V}$ around external calls).

Our solution to the view reconciliation challenge is to adjust the connection between the physical representation and the logical representation when we cross language boundaries. Concretely, whenever we are in λ_{ML}, the resource $\ell \mapsto_{\text{ML}} \vec{V}$ is connected directly to the physical memory (*i.e.*, $\sigma(\ell) = V$). When we transition to λ_C, we connect it instead to a runtime block in the runtime memory ζ using ghost state. When $\ell \mapsto_{\text{ML}} \vec{V}$ is connected to ζ, we can obtain the runtime resource $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$. To ensure that there are never two overlapping views exposed, we maintain as an invariant (not in the Iris sense) that the runtime blocks backing $\ell \mapsto_{\text{ML}} \vec{V}$ and $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$ must

[9] In Fig. 7, the runtime state in the case ML $((\zeta, \chi, \sigma, rm), \sigma)$ contains both "ζ" and "σ". The heap σ is the important one: it represents the entire λ_{ML} memory, mapping locations to their λ_{ML} values. The heap ζ contains additional runtime state (*e.g.*, the runtime representation of immutable OCaml values such as integer pairs, or runtime blocks that have been allocated using the FFI and have not been exposed to λ_{ML}).

$$
\begin{array}{lll}
 & \mathsf{wp}\ e\ @\ p, \Psi\{\Phi\} & \triangleq \ \forall \sigma.\ \mathrm{SI}(\sigma) \Rrightarrow \mathsf{wp}'\ (e, \sigma)\ @\ p, \Psi\{\Phi\} \\
\text{(WP1)} & \mathsf{wp}'\ (v, \sigma)\ @\ p, \Psi\{\Phi\} & \triangleq \ \mathrm{SI}(\sigma) * \Phi(v) \\
\text{(WP2)} & \mathsf{wp}'\ (K[\texttt{call}\ fn\ \vec{v}], \sigma)\ @\ p, \Psi\{\Phi\} & \triangleq \\
 & \quad \mathrm{SI}(\sigma) * fn \notin \mathrm{dom}(p) * \exists \Phi'.\ \Psi\ fn\ \vec{v}\ \Phi' * \triangleright \forall v'. \Phi'(v') \mathrel{-\!\!*} \mathsf{wp}\ K[v']\ @\ p, \Psi\{\Phi\} & \\
\text{(WP3)} & \mathsf{wp}'\ (e, \sigma)\ @\ p, \Psi\{\Phi\} & \triangleq \qquad \textit{(e not a value nor a call)} \\
 & \quad \exists S.\ p; (e, \sigma) \longrightarrow\!\!\!\!\rightarrow S * \forall e' \sigma'.\ (e', \sigma') \in S \Rrightarrow \triangleright \Rrightarrow \mathrm{SI}(\sigma') * \mathsf{wp}\ e'\ @\ p, \Psi\{\Phi\} &
\end{array}
$$

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

always be disjoint. We maintain this invariant in the model of the GC resource:

$$\mathrm{GC}(\theta) \triangleq \exists \zeta_{phys}, \sigma_{virt}, \zeta_{virt}, \zeta_{ml}.\ \mathtt{SI}_{\mathtt{ML}}(\sigma_{virt}) * \boxed{\bullet(\zeta_{virt})} * \zeta_{phys} = \zeta_{virt} \uplus \zeta_{ml} * \mathrm{repr}(\sigma_{virt}, \zeta_{ml}) * \cdots$$

The physical view that the operational semantics has at this point on the $\lambda_{\mathtt{ML}}$-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} that backs up the resource $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$ (through $\boxed{\bullet(\zeta_{virt})}$) and the part ζ_{ml} that backs up the resource $\ell \mapsto_{\mathrm{ML}} \vec{V}$. We cannot directly back $\ell \mapsto_{\mathrm{ML}} \vec{V}$ with a runtime heap, since "$\mapsto_{\mathrm{ML}}$" is a resource from $\mathbf{Iris}_{\mathtt{ML}}$ which—as usual—is backed by a $\lambda_{\mathtt{ML}}$-heap (through $\mathtt{SI}_{\mathtt{ML}}(\cdot)$, the "*state interpretation*" in the terminiology of Iris). However, what we can do is maintain a separate, *virtual* $\lambda_{\mathtt{ML}}$-heap σ_{virt} (not present in the underlying physical memory) which (1) backs the resource $\mapsto_{\mathrm{ML}}$ in the form of $\mathtt{SI}_{\mathtt{ML}}(\sigma_{virt})$ and (2) is faithfully represented by the runtime heap ζ_{virt} (denoted $\mathrm{repr}(\sigma_{virt}, \zeta_{ml})$).

The way the view reconciliation rules (see Fig. 6) are proven sound 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 $\lambda_{\mathtt{ML}}$-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 $\mathbf{Iris}_{\mathtt{ML+C}}$

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

Theorem 4.2 (Adequacy of $\mathbf{Iris}_{\mathtt{ML+C}}$). *Let p be a $\lambda_{\mathtt{ML+C}}$-program and ϕ a pure predicate on integers. If $\varnothing \models p : \mathrm{main}(\phi)$, then $p; (\texttt{call main []}, \varnothing) \longrightarrow\!\!\!\!\rightarrow^*_{\mathtt{ML+C}} \{(\dot{n}, h) \mid \phi(n)\}$.*

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

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

At first glance, this definition may seem odd, because it may seem 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 $p; s \longrightarrow\!\!\!\!\rightarrow S$, the outer choice over the set of states S is *angelic* and, thus, when we reason about all executions, we need to *resolve* this choice (hence "$\exists S'$"). The inner choice over the next state $s \in S$ is *demonic* and, hence, when we reason about all executions, we need to *accept* this choice (hence $\forall s' \in S'$).

Angelic non-determinism and step-indexing. Proving Theorem 4.2 is non-trivial. What makes it challenging is the interaction of angelic non-determinism (contained in the semantics) and *step-indexing* (underlying the model of Iris). To explain this interaction, we first take a step back and discuss the model of Hoare triples in $\mathbf{Iris}_{\mathtt{ML+C}}$. 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 (*i.e.*, $\{P\}\ e\ @\ p, \Psi\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathrm{wp}\ e\ @\ p, \Psi\{\Phi\})$). A simplified version of the weakest precondition $\mathrm{wp}\ e\ @\ p, \Psi\{\Phi\}$ 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 function calls in the way explained in §2.2. Second, the definition is *step-indexed* which reveals itself through the modality "$\triangleright$" that appears in the cases (WP2) and (WP3). Step-indexing enables powerful recursive reasoning in Iris and, hence, $\mathbf{Iris}_{\mathtt{ML+C}}$. We make use of it in our examples (in §5) and when reasoning about callbacks (see ExecCallback). Third, the weakest precondition uses multi-relations $p; s \longrightarrow\!\!\!\!\!\rightarrow S$ instead of "ordinary" relations $p; s \to s'$ in (WP3). This is only the case for $\mathbf{Iris}_{\mathtt{ML+C}}$; the logics $\mathbf{Iris}_{\mathtt{C}}$ and $\mathbf{Iris}_{\mathtt{ML}}$ use weakest preconditions with ordinary relations as usual. We use multi-relations here to deal with the semantics $\longrightarrow\!\!\!\!\!\rightarrow_{\mathtt{ML+C}}$.

Combining step-indexing and angelic non-determinism, as done in the weakest precondition (see Fig. 10), makes proving adequacy nontrivial. The reason is a circularity that arises in the model: (1) the number of steps taken by the program might depend on angelic choices, (2) the witnesses for the angelic choices proven in the logic may themselves depend (implicitly) on an underlying step-index from the model, and (3) the step-index only ensures adequacy if it is greater than the number of steps the program takes. More concretely, to prove Theorem 4.2, we have to take the angelic choices over S in (WP3) *inside of* $\mathbf{Iris}_{\mathtt{ML+C}}$ and "replay" them *outside of* $\mathbf{Iris}_{\mathtt{ML+C}}$ to resolve the angelic choices in $p; s \longrightarrow\!\!\!\!\!\rightarrow^* S$. However, in traditional step-indexed logics such as Iris, the meaning of existential quantification is weaker than what one would intuitively expect. 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 $p; s \longrightarrow\!\!\!\!\!\rightarrow^* S$, because of the aforementioned dependence on the step-index.

To establish adequacy (*i.e.*, to "replay" the angelic choices outside of $\mathbf{Iris}_{\mathtt{ML+C}}$), 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 provides the kind of "witness extraction" from existential quantifiers that we need in this proof. More specifically, when we use transfinite step-indexing (*i.e.*, ordinals as step-indices), we can prove that the angelic choices cannot *truly* depend on the step-index, which breaks the dependency cycle outlined above. The details of this construction can be found in the supplementary material [Guéneau et al. 2023].

Language generality. Above, we have discussed how to compose $\mathbf{Iris}_{\mathtt{ML+C}}$-proofs and how to use adequacy to obtain assurances for a combined $\lambda_{\mathtt{ML}}$ and $\lambda_{\mathtt{C}}$ program. In the accompanying Coq development [Guéneau et al. 2023], we prove many of these results in a language-generic form for arbitrary languages in the sense of §3. Concretely, we define the weakest precondition $\mathrm{wp}\ e\ @\ p, \Psi\{\Phi\}$ in a language-generic form (*i.e.*, parameterized over the concrete language under consideration) for languages with external calls and angelic and demonic non-determinism. We then prove interface implementation (IntfImplement), the rule of consequence (IntfConseq), and linking (Link) parameterized over the language. Moreover, we prove adequacy for the weakest precondition $\mathrm{wp}\ e\ @\ p, \Psi\{\Phi\}$ parameterized over the language—taking care of the interaction of angelic non-determinism and step-indexing. For Theorem 4.2, we instantiate the language-generic version with $\lambda_{\mathtt{ML+C}}$.

From the language-generic adequacy statement, we additionally derive standard language-specific adequacy statements for $\mathbf{Iris}_{\mathsf{C}}$ and $\mathbf{Iris}_{\mathsf{ML}}$—provided the program under consideration does not contain any external function calls (*i.e.*, it can be verified against the empty interface $\varnothing$). (We do so using a generic lifting from standard relations $p; s \to s'$ to multi-relations $p; s \twoheadrightarrow S$.) For example, for $\mathbf{Iris}_{\mathsf{ML}}$, we derive the following adequacy statement (and analogously for $\mathbf{Iris}_{\mathsf{C}}$):

Corollary 4.3 (Adequacy of $\mathbf{Iris}_{\mathsf{ML}}$). *Let e be a λ_{ML} expression and ϕ a pure predicate on integers. If $\{\mathsf{True}\}\ e\ @\ \varnothing, \varnothing\ \{V.\ \exists n.\ V = n * \phi(n)\}_{\mathsf{ML}}$ in $\mathbf{Iris}_{\mathsf{ML}}$, then* $\mathsf{safe}(e, \phi)$.

Here, the predicate $\mathsf{safe}(e, \phi)$ guarantees that e is never stuck and that, if e terminates, the resulting value is an integer satisfying ϕ.

5 CASE STUDIES

We have applied Melocoton to several interesting case studies. In §5.1 we start with three examples that illustrate how Melocoton can be used for *program verification*. Afterwards, in §5.2, we show how Melocoton can, *additionally*, be used to prove type safety of external functions implemented in C.

5.1 Program verification

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:

$$\frac{\forall k.\ \{P(k)\}\ F(k)\ @\ \varnothing, \Psi_{buf}\,\{u.\ u = f(k) * P(k+1)\}_{\mathsf{ML}}}{\begin{array}{l}\langle 0 \le i \le j < n * i \le |\vec{m}| * \mathsf{buffer}_{\mathsf{ML}}(V, n, \vec{m}) * P(i)\rangle\ \mathsf{buf\_upd}\ [i, j, F, V] \\ \langle V'.\ V' = \langle\rangle * \mathsf{buffer}_{\mathsf{ML}}(V, n, \vec{m}[i := f(i), \ldots, j := f(j)]) * P(j+1)\rangle\end{array}}$$

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 $P(\_) \triangleq \ell \mapsto_{\mathsf{ML}} \vec{n}$ (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. (OCaml exposes an `Obj` module providing some escape hatches that expose the runtime representation of values, but it is undocumented and breaks type safety so we do not consider it here.) As such, this example demonstrates that the FFI can be used to add new functionality to λ_{ML}.

Zig-Zag Lists. As another example, we have verified a "zig-zag list"—a linked list implementation whose cons-operation allocates cells on the C heap to store the head and tail of the list (OCaml values), and then exposes this "cons-cell" to OCaml by embedding it into a custom block. To make this work, the fields of the cons-cell are registered with the GC as *global roots*, ensuring that they are not accidentally deallocated by the GC. We provide C implementations of external functions

to work with such lists in OCaml (*e.g.*, **head**, **cons**). We use the term "zig-zag list" because, when traversing such list, one follows pointer chains that alternate between the OCaml and C heaps. This example demonstrates that Melocoton supports data structures spanning both heaps, and that the values of each language can be stored in the other's heap.

5.2 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 $fn : A_1 \to \cdots \to A_n \to B$ to every external function *fn* in it. Then, we interpret the contexts using the interfaces Ψ of $\mathbf{Iris}_{\mathsf{ML}}$ such that each external function $fn \in \Sigma$ is assigned (the semantic interpretation of) its type as a specification in Ψ. Finally, we use $\mathbf{Iris}_{\mathsf{C}}$ to *validate* the assumed types of external functions, which means we prove in $\mathbf{Iris}_{\mathsf{C}}$ 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)) =
  let l = ref (fun _ -> assert false) in
  l := (fun x -> f (fun y -> callk l y) x);
  (fun x -> callk l x)
```

```
value callk(value l, value x) {
  value f = Field(l, 0);
  return caml_callback(f, 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 $\forall \alpha\beta.\ ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$.

Event listeners. As our last example, we have verified a small library of "event listeners". It demonstrates a tricky use of higher-order state in 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 of OCaml's type system, can capture the listener to which it is attached. This makes proving type safety of the library tricky, because we can easily run into circularity issues when naively attempting to define an interpretation of **'a listener** (*e.g.*, a listener is safe if its callbacks are safe, and callbacks are safe if the listeners they capture are safe). We resolve these circularity

issues as usual in logical relations, by using *step-indexing*. Concretely, analogous to our standard interpretation of reference types [Timany et al. 2022, Figures 5], we make use of Iris's impredicative invariants [Jung et al. 2018, §7.1], which internally are modelled using step-indexing. As mentioned in §4.3, for the semantics $\twoheadrightarrow_{\mathtt{ML+C}}$, this is only sound because we use a transfinitely step-indexed version of Iris. Ultimately, we prove the functions (`create`, `listen`, `notify`) safe at type: $\forall\alpha.\ \exists \mathit{listener}.\ (\mathtt{unit} \to \mathit{listener}) \times ((\alpha \to \mathtt{unit}) \to \mathit{listener} \to \mathtt{unit}) \times (\alpha \to \mathit{listener} \to \mathtt{unit})$.

6 RELATED WORK

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

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

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

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

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

Verified compilers. Mates et al. [2019] (building on the work of Patterson et al. [2017]) verify a compiler from a stateful language with closures to one without closures in a syntactic multi-language [Matthews and Findler 2007; Perconti and Ahmed 2014] using a logical relation for contextual equivalence. Hur and Dreyer [2011] verify a one-pass compiler from an ML-like language to an assembly-like language, via a cross-language logical relation where garbage collection is axiomatized. Their approach supports linking with manually verified assembly-level code, but only if that code can be proven behaviorally equivalent to some ML-level module. Compositional

CompCert [Stewart et al. 2015] and the line of work it inspired [Gu et al. 2015; Koenig and Shao 2021; Song et al. 2020] show how to extend the CompCert compiler with cross-language linking. The composition of languages based on external calls in Melocoton is inspired by Compositional CompCert's interaction semantics, but extends it with DimSum-style wrappers [Sammler et al. 2023] to handle linking of languages with different memory models. DimSum [Sammler et al. 2023] provides a "decentralized" approach for reasoning about multi-language programs via combinators for linking and language translation, which inspired the definition of the operational semantics in Melocoton (see §3). DimSum only considers the interaction of relatively low-level languages, none of which have such a rich FFI and runtime as OCaml.

Formal models of garbage collection using nondeterminism. Our operational model of garbage collection (§3) uses nondeterminism, which has also been used before in the literature. Hur and Dreyer [2011] similarly decouple GC-managed values from their physical representation, albeit in a program equivalence setting without interactions between the different languages. They model GC-managed values as stored in *logical memories* in which pointers are never moved or deallocated. Logical memories are related to physical memory using a *lookup table*; running the GC is modeled as non-deterministically changing the lookup table. These closely match the block store and address map of our runtime semantics, respectively (Fig. 7). The idea of modeling the effect of the GC via non-determinism (subjet to constraints) can also be found in other, more recent work [Moine et al. 2023; Wang et al. 2019]. None of the above work studies the combination of a GC with an FFI and, hence, they do not consider the ability to register user-declared roots.

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. For instance, we do not model direct pointer-accesses to the contents of runtime blocks as if they were normal C memory. Instead, all modifications go through runtime primitives provided by the FFI. Direct accesses are primarily used for efficiency—to enable exchanging data such as strings or byte arrays without making copies. We also do not currently model features like exceptions or 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.

ACKNOWLEDGMENTS

We would like to thank Jacques-Henri Jourdan for many insightful discussions, and the anonymous reviewers for their helpful feedback. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation and in part by Google PhD Fellowships for the third and fourth authors.

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

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 Emílio 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

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C (Coq development). https://doi.org/10.5281/zenodo.8331210 Project webpage: https://melocoton-project.github.io.

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI*. ACM, 1095–1109. https://doi.org/10.1145/3453483.3454097

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320.

C. E. Martin, S. A. Curtis, and I. Rewitzky. 2007. Modelling Angelic and Demonic Nondeterminism with Multirelations. *Sci. Comput. Program.* 65, 2 (mar 2007), 140–158. https://doi.org/10.1016/j.scico.2006.01.007

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *PPDP*. ACM, 16:1–16:15. https://doi.org/10.1145/3354166.3354181

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 841–856. https://doi.org/10.1145/3519939.3523704

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *POPL*. ACM, 3–10. https://doi.org/10.1145/1190216.1190220

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (aug 2020), 29 pages. https://doi.org/10.1145/3408978

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL (2023), 718–747. https://doi.org/10.1145/3571218

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

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP*. ACM, 166–178. https://doi.org/10.1145/2784731.2784764

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement through restraint: bringing down the cost of verification. In *ICFP*. ACM, 89–102. https://doi.org/10.1145/2951913.2951940

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* 31 (2021), e25.

https://doi.org/10.1017/S095679682100023X

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *PLDI*. ACM, 609–624. https://doi.org/10.1145/3519939.3523703

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: reasonably mixing a functional language with assembly. In *PLDI*. ACM, 495–509. https://doi.org/10.1145/3062341.3062347

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS, Vol. 8410)*. Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR (LNCS, Vol. 12167)*. 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. In *Proceedings of the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2023)*. Association for Computing Machinery.

Ingrid Rewitzky. 2003. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*. LNCS, Vol. 2929. Springer, 256–271. https://doi.org/10.1007/978-3-540-24615-2_12

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 80–95. https://doi.org/10.1145/3453483.3454031

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. ACM, 275–287. https://doi.org/10.1145/2676726.2676985

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. (2022). https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf (Under submission).

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*. ACM, 675–690. https://doi.org/10.1145/2660193.2660201

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 171:1–171:30. https://doi.org/10.1145/3360597

Received 2023-04-14; accepted 2023-08-27